Федеральное государственное бюджетное учреждение науки
Институт системного программирования им. В. П. Иванникова
Российской академии наук

На правах рукописи

Шимчик Никита Владимирович

Исследование и разработка методов поиска уязвимостей в программах на C и C++ на основе статического анализа помеченных данных

Специальность 2.3.5 —
«Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
кандидат физико-математических наук
Игнатьев Валерий Николаевич

Москва — 2024

Оглавление

Стр.

Стр.

Введение

При разработке крупных программных продуктов неизбежно появление значительного количества ошибок в коде. Не все из них оказываются обнаружены и исправлены на этапе разработки из-за сложности в их обнаружении. Это характерно и для так называемых уязвимостей — дефектов исходного кода, которые могут быть использованы злоумышленником для вмешательства в работу программы.

Уязвимости особенно опасны для программ, осуществляющих обмен данными по сети, или же обрабатывающих файлы, получаемые из недоверенных источников, потому что в этом случае уязвимости могут эксплуатироваться даже без прямого доступа к компьютеру с уязвимой программой.

Также опасны уязвимости в библиотеках, поскольку они попадают во все использующие её программы. Показательным примером является уязвимость CVE-2014-0160 (Heartbleed), появившаяся в криптографической библиотеке OpenSSL в 2011 году и обнаруженная только в 2014, из-за чего ей могло быть подвержено до 66% web-сервисов. Суть этой уязвимости заключалась в том, что обработчик сообщений типа heartbeat не проверял корректность размера буфера, указанного в полученных по сети данных, что позволяло выйти за его границы и получить несанкционированный доступ к информации из памяти процесса.

Таких последствий можно было избежать, если бы библиотека проверялась анализатором кода, способным выявлять потенциальные уязвимости вида «данные, полученные из недоверенного источника, нельзя использовать в критических функциях без предварительной проверки их корректности». Под *потенциальными уязвимостями* здесь и далее будут пониматься такие типы дефектов, которые могут приводить к появлению уязвимостей (например, `CWE-120`[1]). Термин *уязвимость* будет использоваться, когда возможность использования злоумышленником уже продемонстрирована (например, `CVE-2014-0160`[2]).

Ключевыми характеристиками любого анализатора являются точность, полнота и масштабируемость анализа. Точность означает процент истинных

[1] CWE-120: Buffer Copy without Checking Size of Input

[2] CVE-2014-0160: Heartbleed

срабатываний от общего числа предупреждений, выдаваемых инструментом. Полнота означает процент обнаруживаемых инструментом ошибок от их общего количества в анализируемой программе. Масштабируемость тем выше, чем меньше время анализа и требуемые ресурсы (например, память) зависят от размера анализируемого проекта.

Методы поиска потенциальных уязвимостей в программах можно разделить на две группы: основанные на динамическом и статическом анализе.

Методы *динамического анализа* исследуют программу в процессе её выполнения. Они показывают высокую точность при анализе полной программы, поскольку для любого предупреждения инструмента известны конкретные входные данные, на которых оно достигается. Масштабируемость и полнота анализа оказываются относительно низкими даже в случае направленного динамического анализа [14], поскольку чем более специфичные условия требуются для реализации определённого пути выполнения программы, тем сложнее найти соответствующие ему входные данные — в общем случае это алгоритмически неразрешимая задача.

Статический анализ не требует запуска анализируемой программы, а вместо этого исследует её модель. Преимуществом его использования является возможность обнаруживать ошибки и потенциальные уязвимости на ранних этапах разработки за счёт того, что статический анализ можно выполнять даже не имея компилируемой программы — тем самым снижается стоимость исправления ошибок и уязвимостей.

Среди его методов можно выделить статическое символьное выполнение, как обеспечивающее высокую точность за счёт проверок выполнимости получаемых в ходе анализа условий. Недостатком подхода является накапливающаяся сложность условий на длинных межпроцедурных путях, в результате чего теряется либо масштабируемость, либо полнота. Также проблемой для статического анализа является анализ циклов, рекурсивных вызовов и другие сложности, связанные с моделированием поведения программы.

Другой разновидностью статического анализа являются методы анализа потоков данных — они моделируют только интересующие зависимости по данным в программе, за счёт чего достигают более высокой масштабируемости и полноты, ценой снижения точности. Примером такого анализа является зада-

ча IFDS[3] [15], которая сводится к проверке достижимости на межпроцедурном графе, отображающем зависимости по данным в программе.

Поскольку большинство уязвимостей так или иначе связано с использованием данных в программе, они часто могут быть обобщены в виде единой задачи *анализа помеченных данных*. В данной задаче в программе выделяются критические функции, в которые не должны попадать определённые данные без их предварительной проверки — обозначим эти данные словом *помеченные*. Само определение помеченности зависит от конкретной уязвимости: например, это могут быть полученные извне данные, которые не должны использоваться при работе с памятью без проверки их корректности, или наоборот — непубличные данные, которые не должны «утекать» за пределы программы.

Задача анализа помеченных данных решается как статическими, так и динамическими методами, с упомянутыми выше достоинствами и недостатками. В данной работе приоритет отдаётся полноте анализа, которая важна для нахождения наибольшего количества уязвимостей, а потому используется алгоритм решения задачи IFDS, позволяющий обнаруживать даже длинные межпроцедурные зависимости по данным в реальных проектах. Тем не менее, без дополнительных алгоритмов и методов такой подход неприменим на практике. Проблема точности в первую очередь вытекает из отсутствия в решении задачи IFDS чувствительности к путям, а также необходимости снятия помеченности с данных. Проблемы полноты являются общими для большинства видов статического анализа и вызваны в частности необходимостью моделирования поведения внешних функций, а также косвенных и виртуальных вызовов. Проблема масштабируемости связана в первую очередь с затратами по памяти для решения IFDS на больших программах, а также избыточным количеством итераций алгоритма, особенно в случае глобальных переменных.

Для того, чтобы анализатор был применим для поиска реальных уязвимостей, он должен устранить перечисленные выше недостатки, а также решать общие для большинства видов статического анализа сложности, такие как необходимость анализа псевдонимов, циклов и рекурсивных вызовов.

Целью данной работы является разработка методов и алгоритмов поиска уязвимостей на основе статического анализа помеченных данных с низким процентом пропущенных и ложных срабатываний, а также масштабируемостью на проекты в миллионы строк кода.

[3]Interprocedural Finite Distributive Subset problem

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

1. Разработать методы и алгоритмы, повышающие полноту анализа помеченных данных за счёт разрешения косвенных вызовов и автоматического поиска источников помеченных данных.
2. Разработать методы и алгоритмы повышения точности анализа помеченных данных за счёт проверки консистентности путей и снятия помеченности с целочисленных переменных.
3. Разработать алгоритмы, повышающие масштабируемость анализа помеченных данных за счёт направленного распространения помеченности через глобальные переменные.
4. Реализовать разработанные методы и алгоритмы и провести их испытание на реальных программных проектах и тестовых наборах с искусственно созданными уязвимостями, чтобы оценить полноту, точность и масштабируемость анализа.

Научная новизна:

1. Алгоритм анализа косвенных вызовов на основе решения задачи IFDS.
2. Алгоритм направленного распространения помеченности через глобальные переменные, на основе анализа графа вызовов и использований переменных.
3. Чувствительный к потоку управления алгоритм снятия помеченности на основе проверок значений целочисленных переменных.
4. Метод уточнения результатов нечувствительного к путям анализа при помощи дополнительного обхода расширенного суперграфа для проверки консистентности путей распространения помеченных данных по выбранным критериям.

Теоретическая и практическая значимость. Теоретическая значимость данной работы заключается в разработанных алгоритмах, повышающих точность, полноту и масштабируемость статического анализа помеченных данных на основе решения задачи IFDS, которые могут быть использованы для дальнейшего улучшения данного вида анализа. В частности, на их основе может быть разработан метод анализа, устраняющий основной недостаток подхода на основе IFDS — отсутствие чувствительности к путям.

Практическая значимость заключается в реализованном автором инструменте Irbis, предназначенном для поиска уязвимостей в программах на языках

C и C++. Этот инструмент встраивается в инфраструктуру статического анализатора Svace, разрабатываемого в ИСП РАН, и способен обнаруживать такие реальные уязвимости как Heartbleed, показывая хорошие результаты как на специальных тестовых наборах, так и на реальных проектах. Разработанные в нём алгоритмы и подходы могут быть использованы для реализации аналогичных инструментов, основанных на открытой компиляторной инфраструктуре LLVM или её аналогах.

Методология и методы исследования. В данной работе использовались методы анализа потоков данных, теории компиляции, теории графов.

Основные положения, выносимые на защиту:

1. Алгоритм анализа косвенных вызовов, позволяющий найти информацию о возможных кандидатах для вызова по указателю на основе междпроцедурного анализа потоков данных IFDS.
2. Алгоритм снятия помеченности с целочисленных переменных, значение которых было проверено, применимый в нечувствительном к путям анализе.
3. Алгоритм направленного распространения помеченности через глобальные переменные, позволяющий не исследовать функции, которые не зависят от значения этих переменных.
4. Метод проверки консистентности путей на основе обхода расширенного суперграфа.

Апробация работы. Основные результаты работы докладывались на:

1. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 4 декабря 2023.
2. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 2 декабря 2022.
3. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 3 декабря 2021.
4. Конференция “Ломоносовские чтения”, секция “Вычислительная математика и кибернетика”. Москва. 23 октября 2020.
5. Международная конференция Spring/Summer Young Researchers’ Colloquium on Software Engineering. Саратов, Россия. 30 мая 2019.
6. Конференция “Ломоносовские чтения”, секция “Вычислительная математика и кибернетика”. Москва. 18 апреля 2018.

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

Публикации. Основные результаты по теме диссертации изложены в 8 печатных изданиях, 4 из которых изданы в журналах, рекомендованных ВАК, 1 — в периодических научных журналах, индексируемых Web of Science и Scopus, 4 — в тезисах докладов. Зарегистрированы 5 программ для ЭВМ.

В работах [1; 2] автором был реализован анализ помеченных данных на основе символьного выполнения, им был полностью написан раздел 4 и отдельные части разделов 1 и 5.

В работах [3—6] все результаты были получены лично автором.

В работах [7; 8] автор принимал определяющее участие в разработке и реализации алгоритмов.

Объем и структура работы. Диссертация состоит из введения, 5 глав и заключения. Полный объём диссертации составляет 108 страниц, включая 5 рисунков, 9 таблиц и 7 листингов кода. Список литературы содержит 90 наименований.

Глава 1. Обзор существующих методов поиска уязвимостей

Данная глава посвящена обзору существующих методов и инструментов поиска уязвимостей в программах, а также вводит основные понятия, использующиеся в последующих главах.

В разделе 1.1 вводятся основные термины анализа помеченных данных, как одного из основных подходов к поиску уязвимостей.

В разделе 1.2 приводится обобщённая классификация существующих методов поиска уязвимостей.

В разделе 1.3 более подробно рассматривается анализ помеченных данных на основе решения задачи IFDS.

В разделе 1.4 подводятся итоги главы.

1.1 Основные термины анализа помеченных данных

Многие типы потенциальных уязвимостей можно формализовать путём сведения их к задаче анализа помеченных данных. Это задача межпроцедурного анализа потоков данных, в которой в программе выделяется следующий набор специальных инструкций:

1. **Истоки** — инструкции, после выполнения которых какие-то данные считаются помеченными.
2. **Передаточные функции** — инструкции, после выполнения которых помеченность распространяется с одних данных на другие.
3. **Санитайзеры** — инструкции, которые снимают помеченность с данных.
4. **Стоки** — инструкции, попадание помеченных данных в которые считается недопустимым поведением.

В случае динамического анализа помеченность может приписываться к адресам ячеек памяти в программе, но в статическом анализе её принято описывать через *пути доступа*: их можно понимать как инструкции относительно того, как можно найти ячейку памяти. Путь доступа состоит из базового значения (некоторого указателя или значения в программе) и последовательности

байтовых смещений и разыменований относительно него. Примеры путей доступа: `var`, `*(p + 10)`, `this->my_array[1]`. На основе путей доступа строятся помеченные факты анализа данных: они состоят из пути доступа, описывающего местонахождение помеченных данных, а также дополнительных полей, зависящих от конкретной задачи. Отдельно выделяется помеченный факт с пустым путём доступа, которым обозначается отсутствие помеченности.

Путём распространения помеченности или *трассой* предупреждения, соответствующими уязвимости, будет называться последовательность пар $\langle N, D \rangle$, где N — вершина графа потока управления (ГПУ), а D — помеченный факт в этой вершине. Эта последовательность упорядочена от истока к стоку помеченности, а в каждой паре наличие помеченных данных зависит от наличия помеченности в предыдущей. Как правило, пользователю демонстрируется не вся трасса, а некоторая её подпоследовательность, содержащая только важные вершины: например, в которых происходит вызов другой функции, возврат из текущей или меняется путь доступа к помеченным данным. Таких точек, в дополнение к истоку и стоку, должно быть достаточно, чтобы пользователь мог понять, откуда берутся помеченные данные и как они достигают точки, в которой уязвимость может быть реализована.

Какие именно инструкции в программе считаются истоками, передаточными функциями и стоками, зависит от конкретного типа уязвимости, который планируется искать. Например, для задания класса уязвимостей «SQL инъекция» можно ввести следующие определения:

- Истоками являются вызовы функций, считывающих данные по сети или из файла — например, вызов `read(file, buffer, size)`, помечающий данные по указателю `buffer`.
- Передаточными функциями являются арифметические операции, операторы копирования, конкатенации строк и т.п.
- Санитайзером могла бы считаться функция, удаляющая из строки все специальные символы.
- Стоками являются функции, осуществляющие SQL запрос к базе данных. Если значение аргумента, содержащего текст запроса, окажется помеченным, то можно сделать вывод о возможности SQL инъекции.

Рассмотрим модельный пример уязвимого кода в Листинге 1.1:

Листинг 1.1: Пример кода, содержащего уязвимость «SQL инъекция»

```
1 database *db;
2
3 void handle_find_user(Request req, Response &res) {
4   string user = req.get_param_value("user");
5
6   string request = std::format(
7     "SELECT id FROM Users WHERE name = '{}'", user);
8
9   auto result = db->execute(request);
10  // ...
11 }
```

Вызов метода `httplib::Request::get_param_value` на строке 4 является истоком помеченности, т.к. он записывает в переменную `user` данные, полученные из одноимённого параметра обрабатываемого HTTP запроса.

Передаточная функция `std::format` распространяет помеченность на `request`, поскольку содержимое этой строки зависит от помеченной переменной `user`.

Стоком помеченности является вызов `database::execute`, выполняющий содержащийся в переменной `request` запрос к базе данных.

Функция `handle_find_user` должна обрабатывать HTTP запрос, получая имя пользователя из его параметров и выполняя SQL запрос к базе данных, возвращающий идентификатор этого пользователя. Однако из-за того, что значение переменной `user` не проверяется, а сам запрос формируется с помощью подстановки строки в текст запроса, этот код содержит уязвимость и эту функцию можно заставить выполнить почти любое действие над базой данных, отправив запрос с подобранным значением `user`, содержащим символ `'`, SQL команду и специальные символы, которые нужны, чтобы запрос оставался корректным.

Примеры других классов потенциальных уязвимостей, которые можно свести к анализу помеченных данных:

1. Выход за границы буфера в результате использования слишком большого или отрицательного индекса массива. Помеченными в данном примере также считаются данные, получаемые из недоверенного источника.

2. Утечку чувствительных данных, при которой помеченными считаются данные, которые разрешено обрабатывать в самой программе, но запрещено передавать или сохранять в файлы.
3. Ошибки использования освобождённой памяти, при которой помеченными считаются значения, указывающие на уже освобождённые области памяти.
4. Использование константных паролей, при которых помеченными считаются данные, использующиеся в качестве паролей или ключей шифрования (распространение помеченности проводится в обратном направлении — от использования в функциях безопасности к инициализации переменной константным значением).

Каждый из этих классов характеризуется собственным набором истоков, стоков и передаточных функций.

1.1.1 Основные термины задачи IFDS

В данном разделе даётся краткое описание схемы решения задачи анализа помеченных данных через её сведение к задаче IFDS, необходимые термины и краткая характеристика. Более подробное описание дано в разделе 1.3.1.

Для решения задачи анализа помеченных данных и поиска уязвимостей может быть использована схема, предложенная авторами инструмента Flowdroid [16], которая предполагает сведение задачи анализа помеченных данных (в их примере — обнаружения утечек чувствительных данных) к задаче IFDS (Interprocedural Finite Distributive Subset) о достижимости по межпроцедурно реализуемым путям на расширенном суперграфе. *Суперграфом* называется межпроцедурный ориентированный граф, множество вершин которого состоит из вершин графов потока управления (ГПУ) всех функций в программе с выделенными вершинами входа и выхода из функции, а рёбра бывают трёх видов:

1. Внутрипроцедурные рёбра, совпадающие с рёбрами из исходного ГПУ функции.
2. Внутрипроцедурные рёбра, соединяющие вершину вызова функции с соответствующей ей вершиной возврата — специальной пустой

вершиной, в которую возвращается управление после завершения вызываемой функции.

3. Межпроцедурные рёбра вызова функции, соединяющие вершину вызова и вершину входа в вызываемую функцию, а также возврата из функции, соединяющие вершину выхода из функции и вершину возврата.

Расширенным суперграфом называется межпроцедурный ориентированный граф с вершинами $\langle N, D \rangle$, где N — вершина суперграфа, а D — помеченный факт. Также он иногда называется *графом распространения помеченности*. В этом графе ребро соединяет вершины $\langle N_1, D_1 \rangle$ и $\langle N_2, D_2 \rangle$ тогда и только тогда, если выполняются два требования:

1. В суперграфе программы существует внутрипроцедурное или межпроцедурное ребро между N_1 и N_2.
2. Из наличия помеченных данных D_1 до выполнения инструкции в N_1 следует наличие помеченных данных D_2 после её выполнения и перехода к N_2.

На практике распространение помеченности можно упрощённо визуализировать в виде ориентированного мультиграфа с пометками на рёбрах, вершины которого совпадают с вершинами ГПУ, а ребро из N_1 в N_2 с пометкой D существует тогда и только тогда, когда в расширенном суперграфе присутствует ребро из некоторой вершины $\langle N_1, D_1 \rangle$ в $\langle N_2, D \rangle$, как показано на Рисунке 1.1. Этот способ подходит для случаев, когда значение D_1 не важно или понятно из контекста.

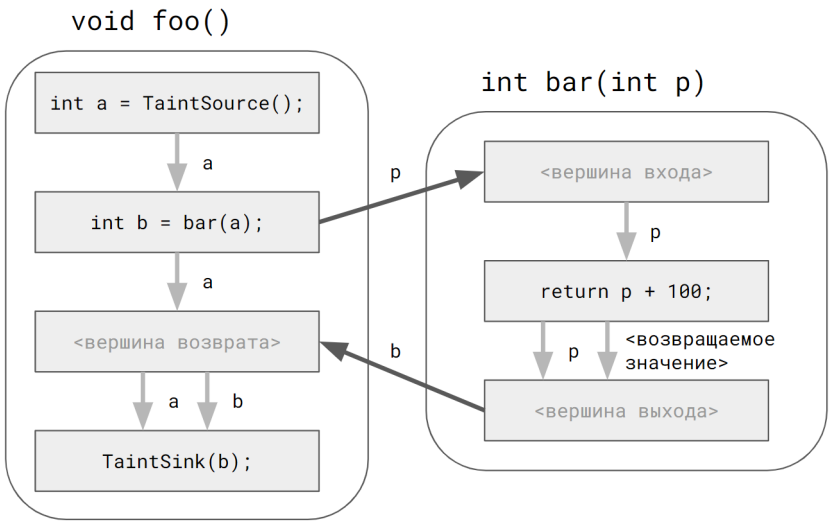

Рисунок 1.1 — Упрощённое представление расширенного суперграфа

Межпроцедурно реализуемым называется такой путь в расширенном суперграфе, в котором для любого межпроцедурного ребра вызова функции из вершины $\langle N_1, D_1 \rangle$ и соответствующего ему межпроцедурного ребра возврата из функции в вершину $\langle N_2, D_2 \rangle$ верно, что в ГПУ существует ребро из N_1 в N_2. Другими словами, возврат из функции должен происходить не по любому из межпроцедурных рёбер, а должен переводить в вершину ГПУ, следующую за той, из которой эта функция была ранее вызвана. На Рисунке 1.2 путь, проходящий через вершины $N_{A1} \to N_{B1} \to N_{B2} \to N_{A2}$, является межпроцедурно реализуемым, а путь через $N_{A1} \to N_{B1} \to N_{B2} \to N_{C2}$ — нет.

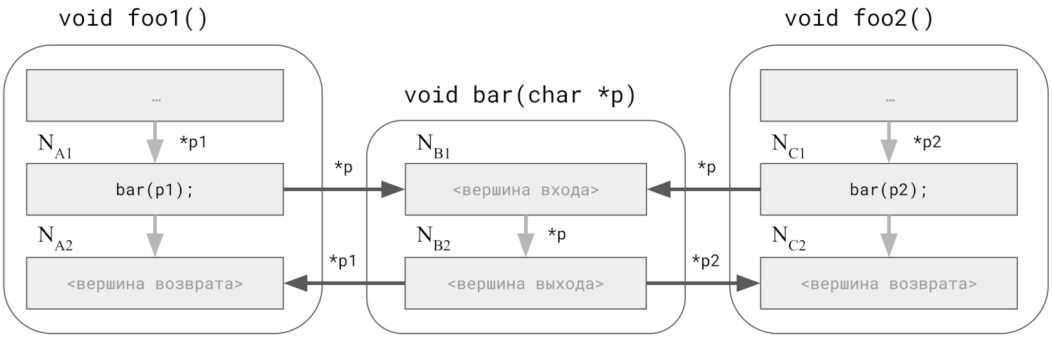

Рисунок 1.2 — Пример межпроцедурных рёбер, ведущих в разные вершины ГПУ

Один из способов соблюдения этого требования при обходе расширенного суперграфа состоит в добавлении понятия контекста вызова, который отображает, с каким помеченным фактом был осуществлён вход в текущую функцию и который не изменяется на внутрипроцедурных рёбрах. Анализ начинается от истока помеченных данных с пустым контекстом и продолжается вдоль путей их распространения. При межпроцедурном переходе в вызываемую функцию, контекст заменяется на совпадающий с помеченным фактом в точке входа в эту функцию. При выходе из функции осуществляется возврат по тем выходящим межпроцедурным рёбрам, которые соответствуют входящим, ведущим в данный контекст. Также для ускорения дальнейшего анализа создаётся ребро резюме, напрямую соединяющее входную и выходную вершины данной функции. Если выход из функции осуществляется с пустым контекстом, то возврат из неё происходит по всем возможным выходным рёбрам, а контекст не изменяется.

Такой подход позволяет эффективно исследовать любые межпроцедурные пути в программе, отслеживая значения только тех переменных, которые могут хранить помеченные значения. Каждая вершина этого графа посещается

только один раз, независимо от того, по какому количеству путей помеченные данные могут попадать в эту точку. Недостатком этого подхода является то, что анализировать программу нужно целиком — то есть максимальный размер анализируемой программы ограничен объёмом доступной оперативной памяти. Это ограничение можно ослабить, если для каждого истока помеченности запускать отдельный анализ и выставить ограничения на глубину проводимого анализа.

Для сравнения, одним из основных подходов к межпроцедурному статическому анализу является анализ на основе резюме функций: он состоит из набора внутрипроцедурных анализов, результатом которых является построение резюме, описывающих поведение функции — в данном примере, каким помеченным фактам на входе соответствуют какие помеченные факты на выходе из неё. Последовательность анализа выбирается на основе графа вызовов функций так, чтобы к моменту анализа вызывающей функции резюме для вызываемых функций уже было готово. Такой подход хорошо масштабируется на проекты любого размера и позволяет контролировать время анализа, выставляя ограничения на время анализа одной функции и размер её резюме. В контексте анализа помеченных данных недостатком такого подхода является то, что во время анализа отдельной функции не известно, через какие её аргументы будут передаваться помеченные данные, потому либо в резюме будут сохраняться все возможные зависимости по данным, либо часть межпроцедурных путей распространения помеченности будет теряться.

Отсутствие чувствительности к путям является одним из основных недостатков: алгоритм решения задачи IFDS предполагает, что в каждой точке программы распространение помеченности не зависит от значения других переменных и пути, по которому выполняется программа. Также из-за того, что алгоритм осуществляет межпроцедурный анализ без его разбиения на анализ отдельных независимых функций, ему может требоваться больше памяти для хранения всей текущей информации, чем для методов анализа, основанных на резюме.

Из этого можно сделать вывод о том, что поиск потенциальных уязвимостей на основе задачи IFDS подходит для тех случаев, когда уязвимость можно описать в терминах наличия зависимости по данным между «истоками» и «стоками» помеченности и важно исследовать максимальное число путей в программе. Однако для качественного анализа недостаточно просто реали-

зовать этот алгоритм — требуются дополнительные алгоритмы и эвристики, обсуждаемые в Главах 2–4, т.к. без них IFDS плохо применим для больших проектов и может выдавать большое количество предупреждений, не указывающих на потенциальные уязвимости.

1.2 Общая классификация методов анализа

Для поиска уязвимостей могут успешно применяться совершенно различные подходы. Основные преимущества и недостатки основных из них были рассмотрены во введении. Несмотря на то, что в данной работе рассматривается алгоритм на основе статического анализа помеченных данных, далее будут также рассмотрены особенности динамического анализа и методов тестирования с целью обоснования перспективности подхода на основе статического анализа исходного кода. Кроме того, в работе рассматривается метод валидации критического пути проявления ошибки, обнаруженного с помощью статического анализа помеченных данных, другими инструментами, что требует описания их ограничений и возможностей.

Ключевым фактором классификации подходов является принцип анализа. Их можно разделить на основанные на:

– тестировании;
– динамическом анализе программы;
– статическом анализе исходного кода.

Тестирование предполагает запуск программы или её компонентов на различных входных данных или в различных тестовых ситуациях и проверку соответствия наблюдаемого поведения ожидаемому. Входные данные для тестирования могут выбираться как вручную, так и генерироваться автоматически в случае фаззинга.

Для динамического анализа также необходим запуск программы на различных входных данных, однако часто используются дополнительные средства, например инструментация кода анализируемой программы, подмена используемых программных библиотек и другое. Например, в инструменте Valgrind [17] используется теневая память, хранящая дополнительную информацию о содержимом физической памяти программы для поиска возможных утечек и

повреждений. В инструментах динамического символьного исполнения, таких как KLEE [18], могут использоваться символьные данные, обобщающие целые наборы конкретных данных в программе.

Формальная верификация [19] — другая техника поиска ошибок в программах, позволяющая в полуавтоматическом режиме не только обнаруживать ошибки в программах, но и доказывать корректность программы, т.е. полное отсутствие ошибок заданного типа. Данная техника анализа программы требует значительного участия квалифицированного специалиста (человеко-годы на несколько тысяч строк [20]) и аппаратных ресурсов. По этой причине применение в индустрии ограничено очень критичными к безопасности сферами. Несмотря на то, что с помощью формальной верификации могут быть найдены уязвимости, она не рассматривается в работе, поскольку разрабатывается для решения других задач.

Основной особенностью статического анализа по сравнению с динамическим является отсутствие необходимости запуска анализируемой программы. Кроме того, для него не актуальна проблема покрытия кода, т.к. работая с моделью программы, он анализирует все существующие пути выполнения. Чаще всего статический анализ требует наличия исходного кода, в том числе и потому что в этом случае пользователю можно продемонстрировать конкретное место ошибки — однако существуют и статические анализаторы двоичного кода. При статическом анализе становятся актуальными проблемы точности (ложных срабатываний, false positive, FP), полноты (пропущенных ошибок, false negative, FN) и масштабируемости. Предупреждения, выданные в результате статического анализа, далее используются человеком с целью их проверки и исправления найденных ошибок в коде. Идеальной проверкой является подготовка такого входа программы, на котором при запуске проявляется найденная ошибка. Для автоматизации решения этой задачи используются комбинированные подходы на основе направленного воспроизведения пути.

1.2.1 Методы, основанные на тестировании

Тестирование является одним из основных методов поиска ошибок и общей оценки качества проекта на этапе разработки. Оно заключается в под-

готовке тестовых примеров (входных данных), запуске программы на них и анализе результатов выполнения. Проверяться может как отсутствие критических ошибок (например, программа не должна аварийно завершаться ни на каких входных данных), так и соответствие результатов выполнения отдельных функций или программы в целом ожидаемым.

Одной из целей тестирования, вне зависимости от используемого сценария, является выявление максимального количества реальных ошибок, не обязательно являющихся уязвимостями, с минимальным участием пользователя, а также максимизация покрытия кода. В случае системного тестирования можно говорить об отсутствии ложных срабатываний: каждая найденная ошибка может быть воспроизведена, поскольку известны параметры запуска тестируемой системы — хотя возможны и сложно воспроизводимые ошибки.

В зависимости от доступности исходного кода программы, тестирование может производиться по стратегии чёрного или белого ящика. Отдельно стоит выделить технику под названием фаззинг [21], которая позволяет автоматически генерировать наборы входных данных, как правило обеспечивая большее покрытие кода, чем подготавливаемые вручную тесты.

Параметры запуска и входные данные при фаззинге могут быть сгенерированы на основе случайных значений, могут являться изменёнными корректными значениями [22], могут вычисляться специальными методами для проверки крайних случаев, таких как, например, 0 или максимальное значение. Современные фаззеры могут использовать статический анализ для поиска «интересных» параметров запуска [23], а также динамический анализ (инструментирование) для оценки покрытия кода.

Фаззинг подходит для поиска уязвимостей, поскольку он генерирует «необычные» входные данные, о которых мог не подумать разработчик при написании тестов, однако необходимость частого запуска программы, а также сложность подбора входных данных, вызывающих ошибку, приводит к известным ограничениям.

Тем не менее, фаззинг в настоящее время — наиболее распространённая техника автоматического тестирования [24], в том числе для поиска уязвимостей и позволяет обнаруживать большое количество новых уязвимостей.

В рамках данной работы фаззинг представляет теоретический интерес с точки зрения возможности использования в двухэтапном анализе для верификации предупреждений, полученных при помощи статического анализа.

Основным недостатком с этой точки зрения является то, что в отличие от статического анализа, подобные эксперименты требуют участие человека для настройки среды для каждой анализируемой программы — особенно в случае программ, получающих данные не из файлов или стандартного потока ввода, а по сети.

1.2.2 Методы на основе статического и динамического символьного выполнения

Все большее применение для поиска уязвимостей находят подходы, основанные на символьном выполнении [25—28]. В классических ранних методах пути выполнения программы анализируются по очереди. Для каждого пути проверяется, что все допустимые входные данные не нарушают спецификаций. Для этого с помощью внешних решателей предикатов (логических формул) проверяется условие нарушения спецификации при выполнении заданного пути. В результате возможно построение примера входных данных для воспроизведения ошибок. Существующие детекторы ошибок KLEE [18] и S2E [29], использующие символьное исполнение, обеспечивают частичную верификацию путей.

При разработке инструментов символьного выполнения необходимо решить две сложные проблемы.

1. Экспоненциальный рост количества путей относительно количества ветвлений в программе, который приводит к сложности масштабирования. Данная задача хорошо изучена, однако все ещё не до конца решена в современных инструментах [18; 20; 23; 29; 30].
2. Проблема проверки достижимости пути, которая обычно сводится к проверке выполнимости предикатов, построенных при анализе. Выполнимость логических формул проверяется с помощью решателей [31], которые достаточны для практического применения [32], однако, тем не менее, существуют «узкие места». Известные решения основаны на кешировании [18], упрощении формул [23]. Однако не существует алгоритма для общего случая.

Для поиска уязвимостей на основе символьного исполнения может применяться комбинированный подход: большое количество предупреждений статического анализатора автоматически проверяется с помощью символьного выполнения или динамического анализа. Второй этап необходим для генерации входных данных, воспроизводящих уязвимость.

Инструменты, основанные на классическом статическом символьном выполнении (Calysto [33] и Saturn [34]) не получили широкого распространения на практике, так как предикат, построенный сразу для всех возможных путей, даже с отсечением циклов, зачастую приводит к слишком ресурсоёмким вычислениям в решателе.

Инструмент Svace [35—37] также содержит подсистему статического символьного выполнения и для преодоления проблем масштабируемости использует подход на основе резюме функций, сокращение модели программы и различные эвристики, снижающие сложность получающихся формул.

Динамическое символьное выполнение состоит в интерпретации программ, данные которых параметризованы символьными значениями. Такие инструменты могут обрабатывать либо по одному пути за раз, либо все пути за один запуск. В первом случае анализируемая программа выполняется с использованием конкретных значений на входе. Параллельно конкретному выполнению поддерживается актуальное символьное состояние. После завершения конкретного выполнения условие пути изменяется с помощью отрицания одного из условий ветвления для покрытия другого пути и решатель вычисляет новые входные значения. Несмотря на простоту реализации и невысокую ресурсоёмкость подход имеет низкую эффективность из-за необходимости многократно обрабатывать одни и те же инструкции программы до достижения целевого ветвления. Наиболее известными инструментами, реализующими данный метод, являются CUTE [38], DART [39], Sage [23].

В алгоритмах, обрабатывающих все пути в одном запуске, в каждом ветвлении происходит клонирование контекста анализа, создание нового потока конкретного выполнения и продолжение обработки теперь уже независимых, отдельных путей. За счёт клонирования исключается необходимость повторной обработки инструкций, однако требуется большое количество памяти для хранения всех состояний. В связи с невозможностью обработки всех путей выполнения программы из-за их неограниченного количества, используются две стратегии обхода путей: 1) максимизация покрытия и 2) поиск входных

данных, позволяющих достигнуть заданной точки программы. Максимизация покрытия используется для автоматической генерации тестов и реализована, например в KLEE [18], S2E [29], Mayhem [30]. Целевой поиск может быть применён для автоматизации проверки предупреждений статического анализатора [40], воспроизведения аварийных завершений программы [41; 42] и локализации ошибок [43].

В данной работе предложен подход проверки выданных предупреждений с помощью статического символьного анализа с использованием инструментов KLEE и Svace. Валидация предупреждения сводится к проверке достижимости пути с целью преодоления нечувствительности к путям статического анализа помеченных данных в форме IFDS задачи. В связи с этим необходимо учитывать ограничения указанных инструментов. Часть из них связана с ограничениями реализации, например отсутствие поддержки вызовов по указателю в Svace, а другие — с ограничениями самого подхода, например, невозможность символьного анализа библиотечных функций без исходного кода.

Современные инструменты на основе символьного выполнения имеют ряд непреодолимых теоретических ограничений, что влияет на полноту результатов при их использовании.

- невозможность гарантированного обнаружения достижимого пути к целевой инструкции (даже если он существует);
- невозможность гарантированного вычисления любого предиката, построенного в процессе анализа из-за его высокой сложности.

В существующих реализациях ограничений гораздо больше: проблемы отсечения путей, связанные с байпасами и попаданиями в локальные минимумы целевых функций, ограничения при моделировании окружения, отсутствие операций со строками и вычислений с плавающей точкой. Таким образом, валидация предупреждений с помощью инструментов символьного анализа — это поиск компромисса между долей ложных предупреждений (точность), количеством пропущенных предупреждений (полнота) и ресурсоёмкостью (масштабируемость) анализа.

1.3 Анализ помеченных данных

Анализ помеченных данных – это одна из форм анализа потоков данных [44; 45], широко применяемая в области компьютерной безопасности. Он может проводиться на основе как динамического, так и статического анализа программы и основывается на распространении пометок по всем возможным путям передачи «интересных» данных в программе от некоторого истока (например, функции чтения пользовательских данных из формы) до стока (например, функции исполнения SQL запроса).

Основной недостаток динамического анализа состоит в замедлении анализируемой программы [46; 47]. Так инструмент libdft [48] увеличивает время выполнения программы до 6 раз [49], а в худших случаях возможно 20–30 кратное замедление [46; 47]. Это существенно ограничивают сферу применения динамического анализа на практике. Для борьбы с существенными накладными расходами предложено множество техник: распараллеливание логики продвижения пометок [50—53], использование специальной аппаратуры [52; 53] или общей памяти [50; 51]. Однако лучших результатов удаётся достигнуть за счёт разделения анализа на две стадии: сбор трассы при выполнении и последующая обработка сохранённых данных [54—57].

Статический анализ помеченных данных нацелен на поиск пути выполнения между предварительно заданными истоками и стоками без запуска программы [16; 58; 59]. При этом статический анализ может выполняться как на основе бинарного представления программы, так и исходного кода. Статический анализ имеет более низкую точность чем динамический, особенно при отсутствии исходного кода программы. Обычно статический анализ указывает точное место возникновения и проявления дефекта, а также предоставляет последовательность диагностических точек, содержащих путь распространения данных, но не может восстановить входные данные для воспроизведения ошибки.

При статическом анализе помеченных данных просматриваются все возможные (в том числе недостижимые) пути выполнения программы. Это находит применение при поиске уязвимостей для приложений Android [16; 60; 61], поиска эксплойтов [59], исследовании уязвимостей в бинарном коде [58]. Инструмент FlowDroid [16] моделирует работу приложений Android, выполняя

поиск потоков данных между истоками и стоками. Для повышения полноты и точности результатов в нем реализован ленивый анализ псевдонимов.

В работе [61] предложено обобщение задачи в виде IFDS/IDE решателя, который формулирует статический анализ помеченных данных в виде задачи достижимости на графе. IFDS и IDE — это обобщённые формы формализации межпроцедурной задачи анализа потоков данных с дистрибутивными передаточными функциями над конечными доменами фактов, предложенные в работе [45] с последующим развитием в [61]. Большое количество задач, традиционно решаемых на основе анализа потоков данных, таких как «живые переменные», «достигающие определения», «распространение констант» имеют дистрибутивные передаточные функции и могут быть формализованы в форме IFDS/IDE. Такое представление позволяет сводить задачу анализа потоков данных к задаче достижимости на графе и решать за полиномиальное время. Подробное рассмотрение данного класса задач содержится далее в разделе 1.3.1.

Другой подход основан на статическом символьном выполнении и производит однократный обход графа развёртки функции, интерпретируя выполнение каждой встреченной инструкции в графе [62]. Обход функций выполняется в топологическом порядке по графу вызовов, чтобы анализ всех вызываемых функций был завершён к началу рассмотрения вызывающей. Проблемными местами такого подхода является анализ циклов и рекурсивных вызовов. Также данный подход может испытывать проблемы с полнотой анализа на длинных межпроцедурных путях из-за количества и сложности накапливаемых символьных формул. Сравнение реализаций подходов на основе IFDS и символьного выполнения, реализованных в одном инструменте, приведено в работах [1; 2].

Таким образом, статический анализ помеченных данных в форме IFDS задачи является перспективным подходом для поиска уязвимостей с точки зрения максимизации полноты анализа, в то же время уступая по точности подходам на основе динамического анализа или символьного выполнения.

1.3.1 Анализ помеченных данных в форме IFDS задачи

Класс задач IFDS[1], изначально предложенный в работе [45], состоит из межпроцедурных задач анализа потоков данных с конечным множеством фактов анализа и дистрибутивными передаточными функциями. Дистрибутивность передаточных функций означает, что для любого факта d, подмножества фактов анализа потоков данных D и передаточной функции $F_{<N_1,N_2>}(D)$ на ребре из вершины программы N_1 в N_2 верно, что $F_{<N_1,N_2>}(d \sqcap D) = F_{<N_1,N_2>}(d) \sqcap F_{<N_1,N_2>}(D)$, где $\sqcap$ — оператор слияния, в качестве которого может использоваться либо объединение, либо пересечение множеств. На практике из этого следует, что все факты могут распространяться независимо друг от друга. Для анализа помеченных данных в качестве оператора слияния обычно используется объединение множеств фактов анализа.

В изначальной работе, посвящённой задаче IFDS, был предложен обобщённый межпроцедурный, чувствительный к полям и контексту вызова алгоритм решения IFDS задачи, имеющий сложность $O(ED^3)$ в общем случае и $O(ED)$ для локально-разделимых задач, где E – число рёбер в межпроцедурном графе потока управления, а D – мощность множества фактов. При этом экспериментальные оценки показывают, что эффективность существенно зависит от D.

Идея подхода IFDS заключается в сведении широкого круга задач анализа программы к задаче достижимости на графе. Для этого на основе межпроцедурного графа потока управления G строится расширенный суперграф G', вершины которого представляют собой декартово произведение $N \times D$, где N — множество вершин в исходном графе, а D – множество фактов анализа потоков данных, включающее в себя, в частности, специальный пустой факт 0. В этом суперграфе вершина (n, d) достижима из начальной $(n_0, 0)$ тогда и только тогда, когда факт анализа потоков данных d верен в вершине n.

Наличие ребра между вершинами в графе определяется следующим образом: между вершинами (n, d) и (n', d') есть ребро тогда и только тогда, когда ребро $n, n' \in G$ и $d' = F(d)$. Такое определение графа G', с учётом дистрибутивности передаточных функций, гарантирует тот факт, что существование корректного пути из начальной вершины $(n_0, 0)$ в вершину (n, d), равносильно тому, что факт d верен в вершине n.

[1]Interprocedural, Finite, Distributive, Subset problems

Следует отдельно отметить, что «корректность» пути является существенным требованием, поскольку при соединении отдельных внутрипроцедурных графов потоков выполнения в единый межпроцедурный граф появляется большое количество некорректных путей. Минимальным требованием, обеспечивающим чувствительность к контексту, является то, что в любом корректном пути возврат из функции должен переходить в ту точку, из которой на этом пути ранее произошёл вызов этой функции. Подробнее этот вопрос рассматривается в работе [16].

Оригинальный алгоритм требовал построения графа G' в явном виде, однако он может содержать до $O(ED)$ вершин и $O(ED^2)$ рёбер, что делает невозможным его построение для реальных программ при сколько-нибудь большой мощности множества D. Однако, на практике граф G' чаще всего оказывается несвязным и из начальной вершины доступна лишь незначительная его часть. Поэтому впоследствии были предложены улучшения оригинального алгоритма [63], не требующие более построения G' явном виде.

Для задачи распространения помеченных данных необходимо продвижение пометок по базовым блокам и рёбрам расширенного межпроцедурного графа потока управления. Помимо рёбер между функциями и базовыми блоками, такой граф содержит специальные ребра из точки вызова c и точки возврата r: $c \to r$ (call to return), $c \to$ «начало функции» (call to start), «выход из функции» $\to r$ (exit to return). При этом пометки могут продвигаться от источников к стокам и наоборот. Будем говорить, что прямой анализ начинается от источников, обратный — от стоков. Правила распространения помеченных данных задаются передаточными функциями. Анализ для каждого источника (в обратном анализе — стока) проводится независимо.

Алгоритм анализа работает с парами ⟨инструкция, факт⟩, означающими, что в данной точке программы значение, описывающееся в факте анализа, является помеченным. Правила распространения помеченных данных задаются передаточными функциями, отображающими факт анализа данных в множество фактов. Для эффективной поддержки детекторов различных ошибок, имеющих различное соотношение количества источников и стоков, необходим как прямой (от источника к стоку), так и обратный (от стока к источнику) анализ.

Поскольку предполагается, что факты независимы, для кеширования и повторного использования результатов анализа возможно составление резюме

методов. Это значительно ускоряет анализатор, так как это избавляет от необходимости повторно рассматривать не только метод, для которого построено резюме, но и все вызываемые им (в т.ч. транзитивно) методы.

Схема анализа может быть существенно адаптирована для конкретных применений в зависимости от поставленных целей и анализируемых языков. Так, например, в случае языка программирования C необходимо выполнение анализа псевдонимов, без которого результаты базового алгоритма очень неточные.

1.3.2 Инструменты анализа помеченных данных

Широко известным примером инструмента анализа помеченных данных на основе IFDS является FlowDroid [16] — чувствительный к потоку управления, контексту вызова и полям классов, нечувствительный к пути выполнения статический анализатор, обеспечивающий высокую точность для анализа Android приложений на языке Java. Результатом его работы является набор потенциально опасных потоков данных, которые могут быть проанализированы аналитиком для установления истинности или разработки атаки.

Для практической применимости к анализу Android, в котором приложения преимущественно реализованы в виде обработчиков асинхронных событий, FlowDroid использует подсистему моделирования всех возможных последовательностей событий. Истоки и стоки для приложений вычисляются автоматически на основе манифеста приложения и задекларированных форм пользовательского интерфейса. Для повышения полноты анализа в инструменте реализован алгоритм поиска псевдонимов и подсистема девиртуализации. Для поиска псевдонимов используется сочетание обратного анализа для вычисления псевдонимов помеченных переменных и последующий прямой анализ, учитывающий инструкцию активации, для распространения новых пометок. Факты представляют собой пути доступа ограниченной длины, использующие только имена полей в качестве идентификаторов смещений. Это не позволяет различать элементы массива. Обратный анализ поиска псевдонимов в FlowDroid никогда не возвращает управление прямому с целью борьбы с ложными срабатываниями, возникающими когда обратный анализ попадает в

контексты, не просмотренные прямым проходом. Вместо этого, если найден псевдоним, запускается ещё один прямой проход с помеченным псевдонимом, а задача отображения необходимых фактов в контекст вызывающей функции решается при прямом анализе. Построение межпроцедурного ГПУ производится «на лету», и, таким образом, вычисление фактов производится только для тех путей доступа, которые на самом деле помечены. Для моделирования библиотечных функций возможен как их анализ, так и текстовые аннотации, задающие правила создания и распространения фактов.

К основным проблемам инструмента можно отнести следующее. Существенная часть ложных срабатываний и ненайденных ошибок возникает из-за ограничений построения графа вызовов. Также язык Java поддерживает вызовы через «отражение» (reflections), которые сложны для анализа в статике. Кроме того, часть виртуальных вызовов не может быть полно и корректно обработана без профилирования.

Известно несколько инструментов статического анализа помеченных данных, нацеленных на конкретные типы ошибок, различающиеся по точности, полноте и масштабируемости. Одним из активно развивающихся инструментов для языков C и C++ является PhASAR.

PhASAR [64] — инфраструктура статического анализа на основе LLVM [65], которая позволяет пользователю задавать и решать некоторые задачи анализа потоков данных с использованием графа вызовов, анализа псевдонимов. В множество поддерживаемых алгоритмов входит IFDS/IDE, применимый для реализации анализа помеченных данных. К сожалению, готового анализатора, демонстрирующего поиск ошибок не реализовано. Вместо него приводится образец, демонстрирующий тривиальный сценарий и не имеющий практической значимости. На данный момент PhASAR может быть использован при поиске уязвимостей только в качестве инфраструктуры для работы с LLVM-биткодом с рядом существенных ограничений. Например, для линковки файлов отдельных модулей компиляции предлагается использовать сторонний проект wllvm [66], который является простой обёрткой вокруг компилятора Clang и не может гарантировать получение результата для произвольной системы сборки. Задача объявления и построения множеств истоков, стоков, санитайзеров, а также моделирования библиотечных функций в данном инструменте не решается. Однако наиболее существенным ограничением является использование LLVM Value вместо путей доступа для задания фак-

тов алгоритма табуляции. Автор не нашёл возможности использования LLVM Value для моделирования отдельных элементов массива, а также для вычисления отношений (пересечение, вложенность, независимость) между различными LLVM Value, что является критичным для реализации анализа помеченных данных. Таким образом, открытая инфраструктура PhASAR может являться перспективным инструментом для реализации инструмента поиска уязвимостей, однако на данный момент требует решения множества прикладных задач, начиная с разработки самого детектора.

Анализатор CHEX [67] реализует поиск уязвимостей, связанных с перехватом компонентов Android на основе распространения помеченных данных между публичными интерфейсами и значимыми истоками-стоками. Вместо анализа вызовов системных функций Android, для работы требуется их модель, что уступает подходу FlowDroid, где моделирование необходимо в основном для повышения точности анализа. Кроме того, отсутствие анализа псевдонимов, не позволяет обеспечить сравнимых при практическом применении результатов.

Другой инструмент LeakMiner [68] использует практически тот же набор алгоритмов, что и FlowDroid, однако не поддерживает чувствительность к контексту вызова, что отрицательно сказывается на результатах, но обеспечивает более высокую производительность.

Инструмент AndroidLeaks [69] способен обрабатывать событийную модель ОС Android, однако страдает из-за нечувствительности к полям объектов, помечая объект целиком, если на самом деле помечается лишь одно из его полей.

Средство SCanDroid [70] осуществляет поиск утечек данных в межкомпонентных взаимодействиях. SCanDroid консервативно допускает любые пары источника и обработчика события, не противоречащие стандарту языка. Игнорирование девиртуализации приводит к большому количеству ложных срабатываний из-за анализа недостижимых путей в графе вызовов.

Инструмент SparseDroid [71] решает задачу анализа Android приложений, схожую с инструментом FlowDroid, но нацелен на повышение эффективности анализа за счёт учёта «разреженности» графа распространения помеченных данных.

Современные исследования IFDS чаще всего посвящены повышению масштабируемости анализа, а не его точности или полноты. В частности, можно отметить работы [72—74], посвящённые разреженному анализу помеченных дан-

ных и работу [75], в которой помимо прочего исследуется зависимость скорости анализа от его направления.

1.4 Выводы

Тестирование и динамический анализ кода являются важными методами с точки зрения поиска уязвимостей в программах, в том числе за счёт того, что работают напрямую с анализируемой программой, а потому могут обеспечить более высокую точность анализа. Тем не менее, с их помощью сложнее обеспечить полноту и масштабируемость на реальных проектах, особенно если уязвимости находятся на редких путях выполнения программы.

Среди методов статического анализа выделяется статическое символьное выполнение. Оно использует формулы над символьными значениями для описания зависимостей между данными в программах, возникающих на путях выполнения программы, что позволяет достичь высокую по меркам статического анализа точность. Среди недостатков символьного выполнения можно назвать сложность моделирования циклов и рекурсивных вызовов, а также сложность проверки разрешимости формул на длинных межпроцедурных путях, что влияет на полноту анализа на реальных проектах.

Анализ помеченных данных на основе IFDS является распространённым подходом, используемым для поиска уязвимостей — самым известным примером является FlowDroid, позволяющий обнаруживать утечки данных в приложения на Android. Данный подход сводится к решению задачи о достижимости на межпроцедурном графе распространения помеченности и обеспечивает высокую полноту и скорость анализа за счёт того, что анализируются только значения помеченных переменных. Ключевыми ограничениями точности являются отсутствие чувствительности к путям и предположение о независимости наличия помеченности от значений других переменных.

При переносе использованного в Flowdroid подхода с языка Java на более низкоуровневый язык C ухудшается масштабируемость из-за того, что в путях доступа используются не именованные поля классов, а произвольные байтовые смещения. Это значительно увеличивает множество фактов анализа, а следовательно, и максимальный размер расширенного суперграфа. Из-за

этого изменения для практического применения инструмента становятся необходимы ограничения анализа.

Существующие реализации данного подхода для языков C и C++ чаще всего в качестве фактов анализа используют не пути доступа, а значения из инфраструктуры LLVM. При таком подходе множество фактов анализа становится ограничено ценой того, что в анализе можно описать помеченность только тех значений, которые в явном виде фигурировали в коде текущей функции.

Таким образом, в настоящее время автору не известно о существовании инструмента статического анализа помеченных данных для языков C и C++, обладающего чувствительностью к потоку управления, контексту вызова, полям объектов и имеющего прикладную значимость, т.е. способного анализировать реальные проекты.

Глава 2. Методы повышения точности статического анализа помеченных данных

Данная глава посвящена описанию предлагаемых в данной работе алгоритмов и методов, позволяющих повысить общую точность статического анализа помеченных данных на основе IFDS. В данной и последующих главах подразумевается следующая общая схема работы предлагаемого анализатора, изображённая на Рисунке 2.1.

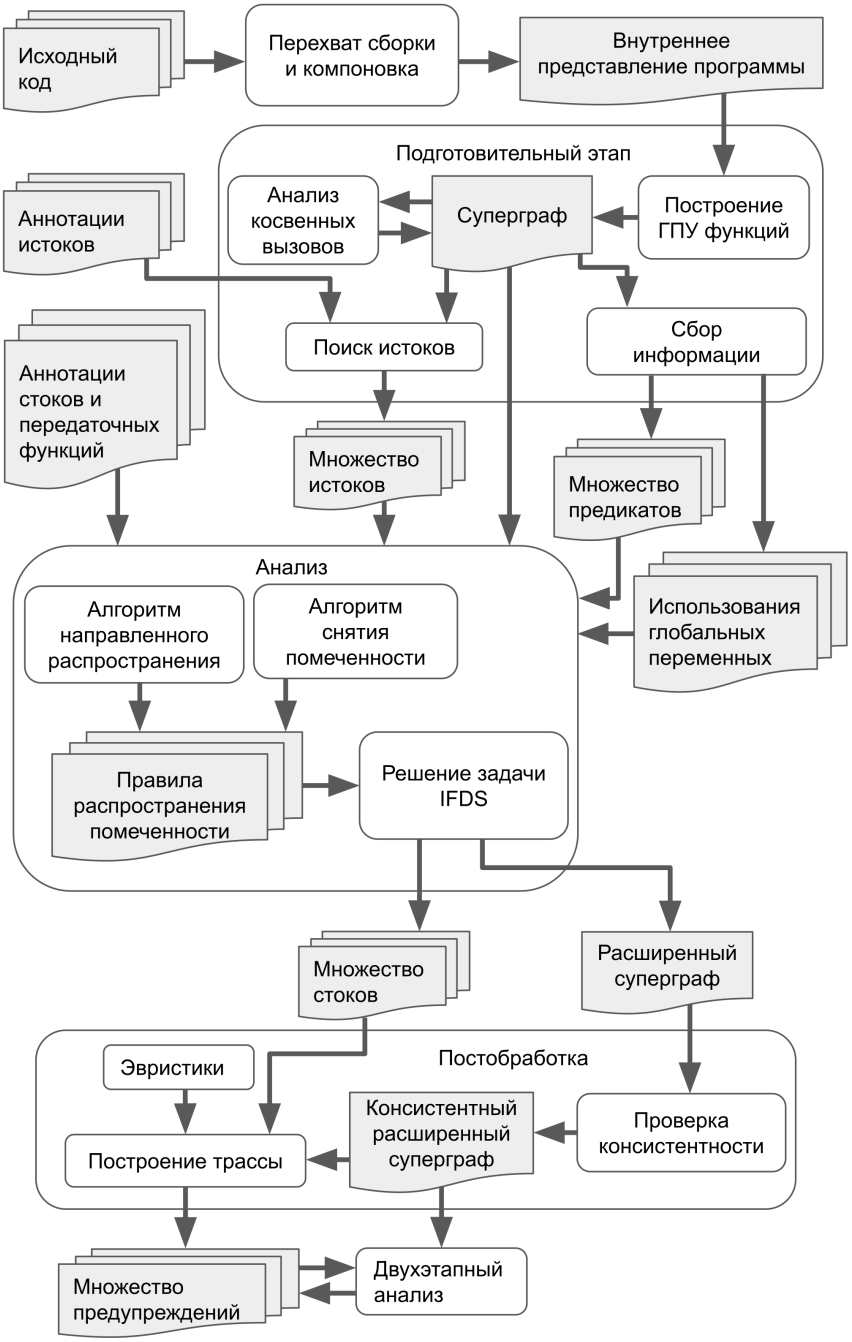

Рисунок 2.1 — Общая схема работы анализатора

Исходный код анализируемой программы переводится сначала во внутреннее представление, а затем в суперграф — межпроцедурный граф, объединяющий графы потоков управления всех функций в программе, с добавленными рёбрами вызова функции и возврата из неё.

На подготовительном этапе выполняется анализ косвенных и виртуальных вызовов, который обнаруживает и добавляет в суперграф функции, которые могут быть кандидатами таких вызовов. Алгоритм анализа косвенных вызовов на основе IFDS предлагается в разделе 3.1 данной работы. Также на подготовительном этапе происходит поиск истоков в программе, от которых будет выполняться анализ помеченных данных — в том числе эвристических, обсуждаемых в разделе 3.2.2. На этом же этапе осуществляется сбор множества предикатов и использований глобальных переменных, которые необходимы для алгоритмов из разделов 2.2 и 4.1 соответственно.

После этого выполняется анализ помеченных данных через решение задачи IFDS по отдельности для каждого найденного истока. Обычные правила построения расширенного суперграфа модифицируются алгоритмом снятия помеченности с целочисленных переменных (раздел 2.2) и алгоритмом направленного распространения помеченности через глобальные переменные (раздел 4.1). Результатом этого этапа анализа является расширенный суперграф, вершинами которого являются пары ⟨вершина суперграфа, факт помеченных данных⟩, описывающий распространение помеченных данных от выбранного истока, а также множество достигнутых стоков помеченности.

На этапе постобработки для каждой пары ⟨исток, сток⟩ осуществляется чувствительный к путям проход по расширенному суперграфу, описанный в разделе 2.1.2, позволяющий получить модифицированную версию расширенного суперграфа, в которой любой путь от истока ведёт к выбранному стоку и является консистентным по выбранным критериям. Из этого графа выбирается один путь распространения помеченности, на основе которого строится трасса, демонстрируемая пользователю. Дополнительные эвристики позволяют уточнить тип предупреждения и отсеять часть предупреждений, похожих на характерные шаблоны ложных срабатываний.

Результаты анализа могут использоваться как напрямую пользователем, так и передаваться на вход следующему анализатору из двухэтапного анализа, описываемого в разделе 2.1.1. Такой подход позволяет повысить точность анализа, отсеяв часть ложных срабатываний инструментом на основе статиче-

ского символьного выполнения, без необходимости выполнения такого анализа для всего проекта целиком.

Данная глава посвящена описанию предлагаемых алгоритмов и методов повышения точности статического анализа помеченных данных. Точность можно оценить вручную, просматривая результаты работы инструмента, или автоматизированно при анализе специальных тестовых наборов с заранее известными потенциальными уязвимостями.

В разделе 2.1 предложены способы уточнения выдаваемых предупреждений при двухэтапного анализа с использованием сторонних инструментов на основе символьного выполнения и собственного прохода по расширенному суперграфу, позволяющего получить консистентный расширенный суперграф, в котором любой путь от истока к стоку является консистентным по заданным критериям.

Для многих видов потенциальных уязвимостей требуется критерий того, что данные проверены и больше не считаются помеченными. В разделе 2.2 предлагается алгоритм снятия помеченности для частного случая — проверки границ значений целочисленных переменных, полученных из недоверенного источника. Этот алгоритм опирается на множество предикатов для целочисленных переменных, выделяемых из программы и использует понятие доминаторов, что позволяет не требовать от алгоритма чувствительности к путям.

Даже верно обнаруженные зависимости между истоками и стоками помеченности не всегда означают наличие потенциальной уязвимости в программе. В разделе 2.3 предлагаются эвристики, которые отсеивают или выделяют отдельные шаблоны предупреждений, характерные для ложных срабатываний.

Вопросам полноты посвящена глава 3.

Проблемой для статического анализа является сложность анализа вызовов, в которых вызываемая функция известна только на этапе выполнения программы: например, виртуальных вызовов и вызовов по указателю. В разделе 3.1 предлагается алгоритм нахождения кандидатов для косвенных вызовов на основе решения задачи IFDS.

Ещё одной проблемой является необходимость описания поведения функций, реализация которых находится за пределами анализируемого проекта является проблемой, с которой сталкивается любой статический анализатор кода. В разделе 3.2 предлагается как формат для их описания с точки зрения

истоков, стоков и передаточных функций, так и эвристики, позволяющие автоматически распознавать два шаблона проектно-специфичных истоков: чтение из внешнего источника и освобождение памяти.

Вопросам масштабируемости анализа посвящена Глава 4.

Распространение помеченности через глобальные переменные может занимать значительную часть общего времени анализа. В разделе 4.1 предложен алгоритм направленного распространения помеченности для таких случаев.

Необходимость анализа программы целиком накладывает сильные ограничения на максимальный размер анализируемой программы. В разделе 4.2 описаны некоторые практические ограничения, которые оказались полезны для решения этой проблемы.

Глава 5 посвящена реализации описанных методов в инструменте статического анализа Irbis.

В разделе 5.1 приводится общая схема работы инструмента и его место в инфраструктуре Svace. В разделе 5.2 перечислен список реализованных детекторов, а также тегов, использующихся для уточнения предупреждений. В разделе 5.3 приведены некоторые особенности реализации инструмента. В разделе 5.4 приведена общая оценка результатов работы инструмента на тестовом наборе Juliet Test Suite и нескольких реальных проектах, а также сравнение с несколькими статическими анализаторами, реализующими анализ помеченных данных.

В сумме предложенные улучшения позволили разработать анализатор, демонстрирующий 100% точность на тестовом наборе Juliet Test Suite и более 50% истинных срабатываний на реальных проектах, позволяющий анализировать проекты на более чем 3 миллиона строк кода за меньше чем 6 часов и способный находить такие уязвимости как Heartbleed на реальных проектах. Для синтетическом проекте Juliet Test Suite, на подмножестве тестов, принадлежащих к одному из 9 поддерживаемых анализатором классов потенциальных уязвимостей, связанных с некорректной работой с памятью и содержащих чтение данных из внешнего источника, инструмент показал 100% полноту и точность, что превзошло результаты сравнимых инструментов статического анализа помеченных данных.

2.1 Проблема отсутствия чувствительности к путям

В данном разделе рассматривается проблема отсутствия у алгоритма решения задачи IFDS чувствительности к путям: это означает, что распространение помеченности зависит только от текущего помеченного факта D и точки суперграфа N и не различает одинаковые факты, пришедшие по разным путям выполнения программы. Это ограничение является существенным для обеспечения полиномиального времени работы алгоритма, однако оно на практике это является упрощением, которое может снижать точность анализа. В частности, данный алгоритм предполагает, что все помеченные данные могут распространяться независимо друг от друга, а значения непомеченных переменных на них не влияют.

На практике часть найденных путей распространения могут быть недостижимыми либо из-за несовместности условий на пути выполнения программы (выполнение программы ни при каких входных данных не может пройти по найденному пути), либо из-за того, что в зависимости от пройденного пути выполнения структура памяти в текущей точке может отличаться (выполнить программу по этому пути возможно, но помеченность не достигнет стока). Модельный пример ложного срабатывания, вызванного отсутствием чувствительности к путям, привёден в Листинге 2.1:

Листинг 2.1: Модельный пример ложного срабатывания

```
1 void foo(char *tainted, bool condition) {
2   if (condition)
3     tainted = TaintSource();
4
5   if (!condition)
6     TaintSink(tainted);
7 }
```

В расширенном суперграфе для этого участка кода будет путь, соединяющий исток помеченных данных на строке 3 и сток помеченных данных на строке 6, следовательно алгоритм IFDS сообщит о наличии здесь ошибки. Тем не менее, чувствительный к путям анализ мог бы определить, что этот путь нереализуем из-за значения переменной `condition`, а значит срабатывание является ложным.

Хотя преимущество чувствительного к путям анализа с точки зрения точности заметно, это не означает, что им можно полностью заменить алгоритм IFDS. В данном примере алгоритм IFDS отслеживает только наличие помеченности в переменной `tainted` в нескольких точках программы, в то время как алгоритмам на основе символьного выполнения требуется отслеживать возможные значения всех переменных и зависимости между ними в каждой точке этой функции. При увеличении масштаба это означает, что чувствительный к путям анализ хуже подходит для обнаружения длинных межпроцедурных путей распространения помеченных данных, поскольку время, необходимое для полного анализа реальной программы, окажется слишком велико. На практике эту проблему обходят, устанавливая ограничения на количество отслеживаемых значений и/или время анализа отдельной функции — однако это также уменьшает вероятность обнаружения сложных ошибок.

Для решения этой проблемы можно использовать двухэтапный анализ, который должен объединять лучшие стороны двух разновидностей анализа:

1. Анализ на основе IFDS, позволяющий обеспечить высокую полноту анализа и обнаружение большого количества пар ⟨ Исток, Сток ⟩. Результатами его работы могут быть сами пары, путь распространения помеченности между ними, путь распространения помеченности между ними или расширенный суперграф, описывающий все такие пути.
2. Чувствительный к путям анализ, который проверяет результаты работы первого анализатора, отсекая часть ложных срабатываний за счёт проверки реализуемости предикатов путей в программе. Это может быть как разновидность статического анализа (например, статическое символьное выполнение), так и направленный динамический анализ.

При правильной реализации, интеграция двух видов анализа позволяет сначала выделить из всего межпроцедурного графа управления программы те пути, которые ведут от истока к стоку помеченности, а затем подтвердить их реализуемость. Это позволяет ограничить подмножество путей в программе, нуждающихся в полноценном исследовании анализатором со второго этапа.

При выборе анализаторов для двух этапов следует учитывать следующие вопросы:

1. Такие расширения как анализ косвенных и виртуальных вызовов должны быть реализованы в обоих анализаторах, потому что иначе

срабатывания, их содержащие, либо не будут найдены, либо всегда будут помечаться как «неподтверждённые».

2. Во втором этапе должны быть возможности для ограничения множества просматриваемых путей на основе информации, полученной на первом этапе — иначе двухэтапный анализ будет менее эффективным. В случае динамического анализа, даже наличие инструментов для направленного анализа (например KATCH [14] для KLEE) не гарантирует возможность повторения искомого пути из-за сложности проверки выполнимости формул и наличия в них не только символьных, но и конкретных значений.
3. Идеальным является вариант, при котором два этапа пользуются одним внутренним представлением (например, биткодом LLVM) и одной системой перехвата сборки, чтобы упростить передачу информации между ними.

Также главными критериями для анализатора первого этапа являются полнота и масштабируемость. С точки зрения полноты требуется, чтобы любое предупреждение, выдаваемое анализатором второго этапа, могло быть выдано и анализатором первого этапа — в противном случае полнота общего анализатора может оказаться ниже, чем полнота каждого из анализаторов по отдельности. Масштабируемость требуется по той причине, что на первом этапе происходит анализ всей программы, а не избранных её частей.

Анализатор второго этапа занимается проверкой ранее найденных путей, потому для него менее важна масштабируемость, а ключевым критерием является точность. Более формально, полнота общего инструмента будет определяться минимумом полноты двух составляющих его анализаторов, а точность будет определяться максимумом точности двух инструментов.

Было опробовано три варианта анализатора для второго этапа:

1. KLEE — инструмент динамического символьного анализа, использующий представление программ в биткоде LLVM. Сама идея комбинирования статического и динамического анализа не нова и рассматривалась в различных работах, например [76—78]. Не удалось получить удовлетворительные результаты покрытия из-за проблем с поиском данных, на которых искомый путь будет достигнут — эту проблему не решают до конца даже специализированные расширения для направленного поиска типа KATCH, поскольку условия могут зависеть не только

от символьных, но и от конкретных значений переменных, как `flag` в Листинге 2.1. Ещё одной проблемой стала необходимость создания заглушек для вызовов библиотечных функций, реализация которых отсутствует в файле с биткодом программы.

2. Svace — имеющаяся в инструменте инфраструктура статического символьного анализа позволила реализовать проверку реализуемости путей, найденных IFDS анализатором. Эта проверка позволила отсечь большое количество срабатываний, однако подход не позволял понять причины, по которым срабатывание не было подтверждено: из-за нереализуемости пути, отсутствия поддержки косвенных/виртуальных вызовов или из-за внутренних ограничений анализатора на размер резюме или времени анализа одной функции.
3. Был реализован дополнительный проход по расширенному суперграфу в рамках самого инструмента. Он происходит после завершения основного анализа и позволяет проверять консистентность путей между некоторым истоком и стоком с точки зрения конкретного критерия: например, консистентность выбора кандидатов для косвенных и виртуальных вызовов. Такой подход позволяет осуществлять простые проверки, требующие наличия чувствительности к путям, и при этом может использовать всю информацию, полученную в ходе анализа. Именно этот вариант используется в инструменте сегодня.

2.1.1 Двухэтапный анализ со статическим символьным выполнением

Одним из опробованных инструментов стал статический анализатор Svace [35], разрабатываемый ИСП РАН. Для него был создан модельный детектор на основе статического символьного выполнения, реализующий поиск реализуемых путей между вызовами *taint_variable()* и *check_taint()* с идентичными аргументами.

Схема работы детектора выглядит следующим образом:

1. Для каждой функции, содержащей вызов taint_variable либо check_tainted создаётся аннотация — эта аннотация содержит усло-

вие внутрипроцедурной достижимости соответствующего истока или стока помеченности.

2. Аналогичные аннотации создаются для каждой вызывающей функции — в них записывается конъюнкция условия достижимости точки вызова и условия, содержащегося в аннотации вызываемой функции. При переходе из контекста вызываемой функции в вызываемую происходит замена формальных параметров функции на фактические аргументы вызова.
3. Если в функции есть вызов и taint_variable и check_tainted (прямой, или посредством аннотированных функций), то строится конъюнкция условий на путях между ними и условий, указанных в аннотациях, после чего суммарное условие используется в запросе к SMT решателю. Если SMT решатель не может доказать, что условие на пути от истока к стоку не является реализуемым — то это срабатывание исходного инструмента считается подтверждённым.

При запуске двухэтапного анализа выявилось, что он способен подтвердить бóльшую часть тестов, написанных для проверки функционала основного анализатора, за исключением тестов на виртуальные и косвенные вызовы, из-за их ограниченной поддержки в Svace — по этой причине во всех последующих запусках двухэтапного анализа анализ виртуальных и косвенных вызовов отключён.

В тестовом наборе Juliet Test Suite [79] не оказалось тестов, для прохождения которых требуется чувствительность к путям, потому использование описанного подхода не смогло показать улучшение точности анализа. Для демонстрации возможностей символьного анализа была добавлена ещё одна специальная функция — sanitize(value, source_i), которая вызывается на каждом участке трассы предупреждения и которой передаётся значение во внутреннем представления LLVM, соответствующее текущему помеченному пути доступа, а также глобальная переменная, соответствующая текущему истока помеченности. В детектор была добавлено условие, что после вызова sanitize() переменная source_i перестаёт быть помеченной, если value не может принимать произвольно большие (по модулю, в случае знаковых типов) значения.

После такого изменения двухэтапный анализ показал уменьшение процента ложных срабатываний на данном тестовом наборе с 59% до 0% по сравнению с одним только первым этапом, без потери истинных срабатываний. При этом

15% из пройденных тестов не обнаруживались имеющимися на тот момент детекторами самого анализатора Svace, что позволяет говорить о том, что данный подход может показывать лучшие результаты, чем оба анализатора по отдельности.

Проверка на библиотеках с реальными уязвимостями, обнаруживаемыми на первом этапе, показала, что сложные пути не удаётся подтвердить из-за ограничений на суммарный размер аннотаций и время работы SMT решателя. После изменения указанных ограничений удалось достичь снижения количества срабатываний на libSSL на 15% (34% при использовании функции sanitize()) с сохранением единственного истинного. На библиотеке libTIFF не удалось подтвердить истинное срабатывание из-за того, что оно требует поддержки косвенных вызовов на втором этапе.

Проведённые испытания показали возможность объединения подхода на основе решения задачи IFDS и статического символьного анализа при помощи инструментирования внутреннего представления анализируемой программы. Этот подход позволяет уменьшить количество ложных срабатываний первого анализатора, а также найти срабатывания, которые не обнаруживались вторым анализатором. Ключевыми проблемами оказались различия в графе вызовов программы из-за недостаточной или отсутствующей поддержки виртуальных и косвенных вызовов во втором анализаторе, а также имеющиеся ограничения на размер аннотаций и запросы к SMT решателю — по этой причине этот подход в инструменте на данный момент больше не поддерживается.

Подробности этого подхода изложены в работе [3].

2.1.2 Двухэтапный анализ с обходом расширенного суперграфа

Поскольку использование стороннего инструмента для выполнения второго этапа анализа означает потерю части результатов из-за различий в реализации анализа косвенных вызовов, внутренних ограничений инструмента и других причин, был предложен метод обхода расширенного суперграфа, получаемого в ходе анализа IFDS.

В отличие от инструментов статического и динамического выполнения, этот подход не отслеживает возможные значения переменных и не проверя-

ет совместность условий на путях — вместо этого формулируются конкретные критерии, которые должны соблюдаться вдоль всего пути выполнения программы, ведущего от истока к стоку помеченности. Примерами таких критериев являются:

1. Совместность классов-наследников, выбираемых в виртуальных вызовах вдоль пути выполнения программы.
2. Совместность кандидатов для косвенных вызовов.
3. В случае анализа не отдельной программы, а объединённого файла биткода всего проекта — совместность файлов, содержащих реализации функций, встречаемых вдоль пути выполнения.

Поскольку эти критерии требуют знание о двух и более посещённых точках в пути выполнения, их проверку невозможно осуществить в рамках алгоритма IFDS из-за отсутствия чувствительности к путям.

Вместо этого между этапом построения расширенного суперграфа и шагом выделения конкретной трассы, по которой будет выдано предупреждения, добавляется дополнительный этап. На этом этапе строится консистентный граф, в вершинах которого к уже имеющейся паре $\langle \mathrm{N}, \mathrm{D} \rangle$, где N — вершина Графа потока управления (ГПУ), D — помеченный факт, добавляется новый элемент C — множество ограничений, накладываемых при прохождении текущего пути. Данный граф строится путём обхода расширенного суперграфа в ширину, в обратном направлении, начиная с каждого достигнутого стока по отдельности. По умолчанию C содержит специальное значение Σ, обозначающее все возможные ограничения, а рёбра, соединяющие вершины консистентного графа, совпадают с аналогичными рёбрами исходного графа.

На межпроцедурных рёбрах, переход по которым требует консистентности с остальными рёбрами в пути выполнения программы, вычисляется множество ограничений C_e. Для виртуальных вызовов это множество классов-наследников, использующих эту реализацию вызванной функции. Для косвенных вызовов это множество базовых блоков, в которых в переменную мог быть записан адрес вызванной функции. Для полнопроектного анализа это множество программ из проекта, в которых существует эта реализация функции.

Если C_0 — это множество ограничений в начальной вершине ребра, то в конечной вершине множество ограничений будет $C_1 = C_e \cap C_0$. Если C_1 является пустым множеством, то текущий путь считается неконсистентным и отбрасывается. После окончания обхода расширенного суперграфа получается либо граф,

в котором нет вершины, соответствующей истоку — в этом случае предупреждение не выдаётся, либо граф, в котором любой путь от истока ведёт к стоку является консистентным. В случае если в графе появилось больше одной вершины, соответствующей одному истоку, но с разными значениями C, то они заменяются на единую вершину с $C = \Sigma$.

Использование такого подхода гарантирует, что из расширенного суперграфа, описывающего все возможные пути распространения помеченности от конкретного истока, при создании предупреждения всегда будет выбран путь, являющийся консистентным с точки зрения выбранного критерия. При этом алгоритм позволяет строить консистентный для нескольких критериев одновременно — в этом случае в качестве значения C используется кортеж множеств ограничений, соответствующих каждому из критериев. Все операции над ними выполняются независимо и требуется, чтобы путь объявляется неконсистентным, если хотя бы одно из множеств пусто.

В худшем случае консистентный граф может иметь в $|\Sigma|$ раз больше вершин, чем расширенный суперграф из-за того, что одна и та же вершина может быть посещена по разным путям — это являлось бы существенной проблемой, поскольку именно хранение расширенного суперграфа составляет основную часть затрат анализатора по памяти. На практике анализатор не показывает существенного замедления или увеличения потребления памяти по нескольким причинам:

– Обход в обратном направлении от конкретного стока означает, что консистентный граф строится не для всех возможных путей распространения помеченности от истока, а только для путей, соединяющих конкретный исток и сток.
– Множество ограничений C остаётся неизменным в подавляющем большинстве рёбер расширенного суперграфа, что означает малое количество дополнительных ветвлений при построении консистентного графа. В тривиальном случае, если во всех вершинах $C = \Sigma$, общее количество вершин консистентного графа будет меньше или равно количеству вершин расширенного суперграфа.

Влияние такого анализа на результаты показано в Таблице 1: ни на одном из проектов не было замечено увеличение общего времени анализа и потребления памяти, из-за того что основную сложность для инструмента представляет решение задачи IFDS, а данный метод на него не влияет и работает отдельным

этапом. Поле «Неконсистентных предупреждений» показывает количество срабатываний, для которых без двухэтапного анализа из расширенного суперграфа могла быть выделена неконсистентная трасса.

Таблица 1 — Алгоритм проверки консистентности путей

Проект	Всего предупреждений	Неконсист. предупреждений	Время анализа	Память
OpenSSL	1137	41	3 ч. 17 м.	3 Гб.
LibTIFF	283	1	7 м.	2 Гб.
binutils	1316	25	38 м.	10 Гб.

Таким образом, хотя в процентном отношении использование данного подхода повлияло на небольшое количество срабатываний инструмента, оно не повлекло дополнительных расходов даже в случае анализа проекта binutils, состоящего более чем из 3 миллионов строк кода, а значит позволяет добавлять отдельные проверки, требующие чувствительности к путям, без дополнительных издержек.

Подробности этого подхода изложены в работе [7].

2.2 Санитизация помеченных данных

В данном разделе рассматривается вопрос необходимости снятия помеченности с данных. Наличие зависимости по данным от истока к стоку не всегда автоматически означает наличие уязвимости в программе. Может оказаться, что помеченная переменная в точке стока не может принимать те значения, на которых уязвимость проявляется, или же значения переменных, фигурирующих в условиях, каким-то образом зависят от значения помеченной переменной, из-за чего уязвимость не реализуется. Для того чтобы иметь возможность избавиться от таких предупреждений, необходимо предусмотреть возможность снятия помеченности.

Снятие помеченности (санитизация) описывает действия, которые следует выполнить, чтобы ранее помеченные данные можно было безопасно использовать в стоках помеченности. Существуют как минимум два её вида:

1. Изменяющее данные — например функция, удаляющая специальные символы из SQL запроса или изменяющая значение указателя или индекса массива на допустимые. В используемом в данной работе подходе такой тип санитайзеров моделируется отсутствием передаточной функции для аргументов, с которых стирается помеченность.
2. Проверяющее данные — такая функция проверяет допустимость значения переменной и возвращает `true` или `false`. Такой тип санитайзеров сложно поддержать в рамках IFDS из-за предположения о том, что наличие помеченности не зависит от значения других переменных.

В данной работе был предложен Алгоритм 1, позволяющий реализовать частный случай санитизации — снятие помеченности с целочисленных переменных в случае, если их значение было ограничено сверху и снизу путём сравнения с константой. Это наиболее распространённый способ проверки чисел, полученных из недоверенного источника, а предлагаемый алгоритм не требует чувствительности к путям, а потому подходит для использования в решении задачи IFDS. Этот алгоритм предназначается для детекторов, которые используют помеченность класса «ненадёжные входные данные» по классификации из раздела 5.2.

Он основан на том, что большинство логических формул в популярных низкоуровневых внутренних представлениях программ, включая LLVM, сводятся ограниченному набору операций сравнения и условным переходам между базовыми блоками: даже явная поддержка логических операторов `&&` и `||` не требуется, поскольку в них правый операнд вычисляется или не вычисляется в зависимости от результата проверки левого и на практике они реализуются при помощи инструкции условного перехода. Благодаря использованию дерева доминаторов[80], этот алгоритм подходит для использования в нечувствительном к путям анализе, поскольку пометки добавляются только если они верны независимо от конкретного пути выполнения. Этот подход позволяет отсечь распространённый класс ложных срабатываний, связанный с использованием помеченного значения индекса массива.

Результаты применения данного алгоритма на проектах Juliet Test Suite и OpenSSL, содержащем уязвимость Heartbleed, приведены в Таблице 2. Более подробно о выбранном подмножестве тестов и методологии тестирования можно узнать в разделе 5.4.

Алгоритм 1 Алгоритм снятия помеченности с целочисленных переменных

1. Найти все вершины ГПУ, в которых от результата целочисленного сравнения зависит выбор базового блока для перехода.
2. Из вершин с шага 1 выбрать те, в которых осуществляется сравнение целочисленной переменной X и константы C — без ограничения общности будем считать, что в сравнении переменная всегда находится слева.
3. Назначить веткам выполнения, соответствующим истинности и ложности условия, пометки «X ограничена сверху» ($\overline{X}$) и «X ограничена снизу» ($\underline{X}$), в зависимости от операции сравнения и знаковости:
   - `X == C`: $\overline{X}$ и $\underline{X}$ в истинной ветке;
   - `X != C`: $\overline{X}$ и $\underline{X}$ в ложной ветке;
   - `X < C` или `X <= C`: $\overline{X}$ в истинной ветке (и $\underline{X}$ для беззнакового сравнения) и $\underline{X}$ в ложной;
   - `X > C` или `X >= C`: $\overline{X}$ в ложной ветке (и $\underline{X}$ для беззнакового сравнения) и $\underline{X}$ в истинной.
4. Для каждой из веток назначать выбранные пометки базовым блокам, начиная с того, в который осуществляется переход, вдоль переходов в ГПУ, пока выполняются все условия:
   - текущий блок доминируется базовым блоком, содержащим сравнение, а также базовым блоком, с которого начинается выбранная ветка
   - в ГПУ нет пути из базового блока, с которого начинается альтернативная ветка, который бы не проходил через базовый блок условия
   - в текущем базовом блоке не изменяется значение переменной X
5. После выполнения предыдущих шагов для всех операций сравнения в текущей функции, определить базовые блоки, содержащие одновременно $\overline{X}$ и $\underline{X}$: они сохраняются и при последующем анализе при попадании помеченной переменной X в такой базовый блок, помеченность с неё будет сниматься.

Также как и двухэтапный анализ, алгоритм снятия помеченных переменных позволил избавиться от всех ложных срабатываний на тестовом наборе Juliet Test Suite, не потеряв истинные срабатывания. На OpenSSL он позволил отсеять 12 ложных срабатываний (1,4% от общего числа предупреждений) и уменьшить время анализа на 10 минут (12%), при увеличении использования памяти на 130 Мб (5%).

Таблица 2 — Алгоритм снятия помеченности с целочисленных переменных

Проект	Всего предний	Δ TP	Δ FP	Время	Память
Juliet Test Suite	5546			16м 57с	1,35 Гб
после:	4258	0	-1288	16м 27с	1,47 Гб
OpenSSL	834			1ч 22м	2,84 Гб
после:	822	0	-12	1ч 12м	2,97 Гб

2.3 Известные причины ложных срабатываний

В данном разделе описываются характерные примеры ложных срабатываний, замеченные при анализе реальных проектов, а также примеры эвристик, запускаемых для проверки результатов работы алгоритма IFDS, обнаруживающих конкретные шаблоны кода и убирающих соответствующие срабатывания или помечающие их специальным тегом, показывающим пользователю, что эти предупреждения следует проверять в последнюю очередь.

Рассмотрим в качестве примера строку `result = tainted & 0xFF;`. Результат арифметических операций взятия остатка от деления, побитового «И» и некоторых других, применённых к помеченной переменной, формально считается помеченным, однако не может превышать некоторое известное число, из-за чего его часто используют без дополнительных проверок. При этом анализатор не может считать, что результат такой операции не является помеченным, поскольку считывание многобайтовых чисел часто осуществляется макросами, состоящими из таких операций и битовых смещений.

Рассмотрим строку `c = (unsigned char) tainted;`. Значение переменной `c` зависит от помеченной переменной, но из-за приведения к однобайтовому целому типу также сильно ограничено с точки зрения множества возможных значений. Например, его можно безопасно использовать в качестве индекса массива из 256 элементов, что часто встречается в макросах вида `isalpha()`.

Ещё одним примером является `if (buf = safe_alloc(tainted))`. Значение переменной `tainted` используется для указания размера выделяемого в памяти буфера. Если буфер выделился успешно, то хотя значение `tainted` и остаётся помеченным, но оно соответствует размеру буфера `buf`, а потому мо-

жет использоваться в качестве его индекса. Такой подход часто встречается при получении и обработке пакетов данных неизвестного размера — и хотя выделение произвольно больших блоков памяти может являться потенциальной уязвимостью, дальнейшие использования `tainted` с большей вероятностью являются корректными.

Информация о размере буфера в C и C++ часто передаётся межпроцедурно вместе с указателем на буфер через другие аргументы вызова функции или поля того же класса, например `void fill(char *buffer, size_t length);`. В этом случае анализатор считает, что размер буфера неизвестен, в то время как человек предположит, что он равен значению переменных с соответствующим названием.

В большинстве описанных случаев анализатор не может однозначно сделать вывод об отсутствии помеченности в данном шаблоне, поскольку это привело бы к потере части истинных. С практической точки зрения наиболее корректным является подход, при котором на этапе формирования множества предупреждений происходит определение наличия одного из известных шаблонов ложных срабатываний и добавления к типу предупреждения соответствующего ему тега. Теги предназначены для пользователя инструмента и позволяют группировать результаты по степени достоверности: при анализе нового проекта в первую очередь просматриваются предупреждения без тегов, соответствующих шаблонам ложных срабатываний, а при достаточно полной проверке проекта аналитику имеет смысл проверять все возможные срабатывания. Подробнее о тегах, используемых в инструменте, рассказывается в разделе 5.2.1.

В этой главе были описаны способы проверки результатов анализа при помощи двухэтапного анализа с использованием стороннего анализатора или отдельного прохода с обходом расширенного суперграфа. Оба варианта позволяют проводить проверки, требующие чувствительности к путям, что невозможно в рамках алгоритма IFDS — при этом второй этап анализирует не всю программу целиком, а только те её части, которые соответствуют уже обнаруженным путям распространения помеченности от истока к стоку.

Также в данной главе был предложен алгоритм снятия помеченности с целочисленных переменных, не требующий чувствительности к путям, а потому пригодный к использованию в алгоритме IFDS. Применение данного алгоритма позволило на тестовом наборе Juliet Test Suite избавиться от всех ложных сраба-

тываний для детекторов из класса «ненадёжные входные данные», не потеряв истинные срабатывания. На реальных проектах этот алгоритм также позволил убрать не менее 1,4% ложных срабатываний.

Результаты второй главы опубликованы в работах [3—8].

Глава 3. Методы повышения полноты статического анализа помеченных данных

Эта глава посвящена описанию предлагаемых методов повышения полноты статического анализа помеченных данных. Под полнотой понимается процент обнаруженных потенциальных уязвимостей от их общего количества в проекте. Поскольку на реальных проектах их общее количество неизвестно, эту характеристику можно достоверно оценить только через сравнение с результатами работы других анализаторов. На тестовых наборах с заранее известными внедрёнными уязвимостями это значение можно посчитать достаточно точно.

В данной главе рассматривается две основных причины снижения полноты анализа, которые характерны не только для анализа помеченных данных, но и для большинства статических анализаторов:

1. В косвенных и виртуальных вызовах недоступна информация о вызываемой функции, поскольку в общем случае она определяется только на этапе выполнения программы.
2. Во внутреннем представлении программы отсутствуют реализации вызываемых библиотечных функций, потому что они будут добавлены компоновщиком при последующей сборке проекта, в момент запуска программы, динамически во время её работы или иным способом.

Для решения первой проблемы в данной работе предлагается алгоритм на основе решения задачи IFDS, позволяющий обнаруживать связи по данным между взятием адреса функции и его использованием в вызове по указателю. Это позволяет исследовать межпроцедурные пути в программе с косвенными вызовами, не добавляя потенциальных кандидатов для вызова, адреса которых не могут использоваться в конкретной точке вызова.

Для решения второй проблемы принято использовать спецификации, представляющие собой упрощённое описание поведения вызываемой функции — с точки зрения анализа помеченных данных это означает информацию о том, что происходит с помеченностью её аргументов после её выполнения: в каких функциях она появляется, в каких копируется или стирается, а в какие аргументы функций помеченные данные не должны попадать. В данной работе предлагается как формат описания поведения функций, подходящий для анализа помеченных данных на основе решения задачи IFDS, так и эври-

стические алгоритмы, позволяющие автоматически обнаруживать некоторые шаблоны функций, которые могут являться истоками помеченности.

3.1 Анализ косвенных вызовов

Помимо вызова внешних функций, реализация которых отсутствует в файле внутреннего представления программы, статический анализ сталкивается со сложностями при анализе вызовов, в которых вызываемая функция неизвестна на этапе компиляции — при этом возможна ситуация, что при выполнении одной и той же инструкции программы более одного раза каждый раз будет вызываться разная функция. Для языков C и C++ наиболее характерными примерами являются косвенные вызовы (вызовы по указателю на функцию) и виртуальные вызовы соответственно.

При консервативном подходе такие вызовы могут трактоваться так же, как и вызовы библиотечных функций без известной реализации — с точки зрения анализа помеченных данных это означает, что помеченные факты передаются сразу в точку возврата, не попадая в вызываемую функцию. Это приводит к снижению полноты анализа, поскольку возможные стоки помеченности в вызываемой функции не будут достигнуты, а часть путей распространения помеченных данных в вызывающей функции окажется не исследована, если вызываемая функция копировала помеченные данные между аргументами.

Для решения этой проблемы используется анализ виртуальных и косвенных вызовов, который определяет набор функций-кандидатов для каждого такого вызова. В межпроцедурном графе добавляются рёбра, соединяющие вызов с каждым из возможных кандидатов — то есть делается предположение, что в ходе выполнения программы может быть вызван любой из них. Важным вопросом является как можно более точное определение множества кандидатов для вызова. Для виртуальных вызовов это множество обычно получается напрямую из иерархии классов [81]: подставляются все реализации выбранного метода из классов-наследников. Это множество можно дополнительно уточнить при помощи анализа типов [82] в тех случаях, когда возможно извлечь допол-

нительную информации о том, объекты какого класса могут содержаться в переменной на практике.

Для вызовов по указателю тривиальным решением можно назвать подстановку в качестве кандидатов всех возможных функций в программе, подходящих по количеству и типам параметров [83]. Ключевым недостатком этого подхода является высокое количество ложных срабатываний из-за того, что большое количество не имеющих ничего общего семантически функций имеют одинаковые типы параметров. Это особенно характерно для функций с 0–1 параметрами, в случае которых количество возможных классов эквивалентности не превышает количество типов данных в программе. Более того — тривиальный подход не гарантирует даже наибольшую полноту из-за того, что несмотря на то что преобразование между типами указателей на функции приводит к неопределённому поведению в языке C, оно используется на практике, особенно при написании программ, предназначенных для конкретной архитектуры.

В целом этот подход плохо подходит для практического использования в анализе помеченных данных, поскольку если один или более аргумент косвенного вызова окажется помеченным, то каждый дополнительный кандидат приводит к значительному увеличению расширенного суперграфа — а в случае неверно определённых кандидатов новые вершины приводят к ложным срабатываниям анализатора.

Во время практической реализации данного метода и его оценки на реальных проектах наблюдались показатели до нескольких сотен кандидатов, не имеющих никакого отношения к изначальному косвенному вызову, что приводило к большому количеству ложных срабатываний и затратам времени на бесполезное распространение помеченности. Для уменьшения количества кандидатов можно исключить те функции, которые используются только в прямых вызовах и адрес которых никогда не брался, однако это не решает проблемы того, что в языке C сигнатуры функций могут быть похожи: например, `read(buf, length)` и `write(buf, length)` функции часто совпадают по типам параметров и часто встречаются в анализе помеченных данных. С точки зрения полноты оказалось, что в отдельных случаях у вызова функции и возможного кандидата может не совпадать даже количество параметров — например, такое встретилось во время анализа кода, реализующего обработку событий.

Предлагаемый в данной работе метод получения множества кандидатов для вызова функции по указателю сводится к решению задачи IFDS:

1. Фактом такого анализа является текущий путь доступа, а также то, адрес какой функции может в нём храниться.
2. Истоками являются инструкции, использующие адрес функции не для осуществления прямого вызова, а также инструкции, использующие значения глобальных переменных, в инициализаторе которых фигурирует адрес какой-либо функции.
3. Передаточные функции совпадают с теми, которые используются в основном анализе помеченных данных.
4. Стоками являются вызовы функций по указателю.

При достижении стока, функция из помеченного факта сохраняется как потенциальный кандидат для данного вызова и после окончания анализа добавляется в граф вызовов, как показано в Алгоритме 2.

Поскольку результат этого анализа сам зависит от графа вызовов, для нахождения всех возможных кандидатов необходимо запускать больше одной его итерации. В этом случае для каждого анализируемого истока стоит записывать все посещённые в ходе анализа косвенные вызовы и достигнутые выходы из функции. Если в ходе анализа для одного из записанных косвенных вызовов был найден новый кандидат, или же для покидаемой функции было добавлено новое ребро возврата — анализ для этого истока может дать новые результаты, а потому должен быть повторён. Все остальные истоки на новой итерации следует пропустить.

Поскольку решение этой задачи IFDS может быть сравнимо по сложности с основным анализом и только косвенно влияет на результаты анализа, с практической точки зрения имеет смысл ограничивать глубину и время, выделяемое на решение данной задачи. С учётом этого и необходимости распространять адреса функций через всю программу, была разработана дополнительная эвристика: она предполагает, что если адрес функции F был записан в некоторое поле структуры, то любое считывание из этого поля в программе может вернуть адрес этой функции — без необходимости поиска пути, ведущего от записи к чтению. Это предположение является интуитивно верным для большинства программ и совпадает с тем, как программист может выяснять, адреса каких функций могут содержаться в данном поле структуры. Если эта эвристика включена, то при достижении вершины суперграфа, осуществляющей запись

Алгоритм 2 Вычисление множества кандидатов для вызовов по указателю на основе IFDS

Входные данные: Суперграф G

Выходные данные: Множество $Candidates$ пар вида $\langle N,F\rangle$, в которых N — вершина вызова по указателю, а F — возможный кандидат вызова

```
Candidates ← ∅
for all F ∈ {функции из G} do
    Sources ← ∅
    for all U ∈ {все упоминания функции F} do
        if U используется в инструкции, входящей в суперграф then
            N ← ⟨вершина суперграфа, содержащая U⟩
            if N это взятие адреса F then
                D ←путь доступа к U в N
                Sources ← Sources ∪ {⟨N, D⟩}
            end if
        else if U в инициализаторе глобальной переменной T then
            D ←путь доступа к U, использующий в качестве основы T
            Sources ← Sources ∪ FIND_USES(T, D)
                ▷ Рекурсивная функция, выполняющая аналогичные шаги для
                  упоминаний T и возвращающая множество пар ⟨вершина су-
                  перграфа, путь доступа⟩, которые могут содержать адрес
                  функции F
        end if
    end for
    Sinks ← SOLVE_IFDS(G, Sources)
                ▷ Решение задачи IFDS с истоками из множества Sources и ин-
                  струкциями вызова по указателю в качестве стоков
    for all N ∈ Sinks do
        Candidates ← Candidates ∪ {⟨N, F⟩}
    end for
end for
```

в поле некоторой структуры, распространение факта с адресом функции продолжается во всех вершинах, считывающих значение этого поля.

На Juliet Test Suite этот алгоритм позволил пройти все тесты на косвенные вызовы, не оказав существенного влияния на время анализа. Для сравнения, тривиальный алгоритм на данном проекте не применим, поскольку каждый вызов по указателю получает несколько сотен кандидатов. В зависимости от настроек анализа, это приведёт либо к большому количеству ложных срабатываний и сильному замедлению, либо, если вызовы с большим количеством кандидатов игнорируются, будет неотличимо от запуска без анализа косвенных вызовов. На OpenSSL он продемонстрировал результаты, показанные в Таблице 3.

Таблица 3 — Результаты анализа косвенных вызовов на OpenSSL

Алгоритм	Функций	1-3 кандидата	4+ кандидата	Время
Тривиальный	18	352	810	<1с
IFDS	350	293	57	8м 17с

Алгоритм на основе IFDS позволил найти хотя бы одного кандидата для вызова по указателю в 30% случаев по сравнению с тривиальным алгоритмом, но качество подставляемых кандидатов оказалось значительно выше, что заметно и по количеству вызовов, для которых подставилось 4 и более кандидатов, так и при ручной проверке случайного набора кандидатов, во время которой не было выявлено ложных срабатываний алгоритма.

Влияние алгоритма анализа косвенных вызовов на общую полноту и масштабируемость анализа можно увидеть в Таблице 4.

Таблица 4 — Влияние алгоритма анализа косвенных вызовов на анализ помеченных данных на OpenSSL

Алгоритм	Предупреждений	Время	Память
Игнорирование	57	8 мин.	2.05 Гб.
Тривиальный	49	53 мин.	2.92 Гб.
IFDS	60	33 мин.	2.65 Гб.

При стандартных настройках запуска, использование тривиального алгоритма анализа косвенных вызовов приводит к увеличению продолжительности анализа более чем в 6 раз, при этом 10 прежних срабатываний исчезло и появилось 2 новых, одно из которых является ложным, так как при вызове по

указателю был подставлен неправильный кандидат. Этот результат объясняется тем, что анализатор тратит много времени на анализ заведомо невозможных путей, а также тем, что срабатывания с путями доступа, смещения в которых не соответствуют смещениям полей в используемых структурах автоматически подавляются инструментом.

Использование анализа косвенных вызовов на основе IFDS также увеличило продолжительности анализа за счёт того, что было исследовано больше путей, но позволило найти 3 дополнительных предупреждения.

3.2 Спецификации для внешних функций

Общей для большинства статических анализаторов проблемой является то, что результат выполнения исполняемой программы определяется не только её собственным кодом, но и реализациями внешних библиотек, а также операционной системой, взаимодействие с которой осуществляется через системные вызовы.

С точки зрения анализатора это выглядит как вызов функции, у которой отсутствует реализация, а потому результат её выполнения и воздействие на переданные аргументы неизвестны. Хотя можно делать некоторые общие предположения о том, что функция всегда перезаписывает переданные ей аргументы и возвращает некоторое неизвестное значение — для полноты анализа крайне рекомендуется составление спецификаций хотя бы для наиболее часто используемых функций: например, входящих в библиотеку стандартных шаблонов или конкретные библиотеки, использование которых ожидается в анализируемых проектах. Часть таких спецификаций должны быть созданы разработчиками анализатора, в то время как другие, более специфичные для конкретных проектов, могут создаваться самими пользователями анализатора.

Для анализа помеченных данных необходимо по крайней мере три разновидности спецификаций:

- *истоки* описывают пути доступа, которые могут содержать помеченные данные после вызова этой функции;
- *передаточные функции* описывают зависимости между помеченностью аргументов функции и возвращаемого значения до и после её вызова;

– *стоки* описывают пути доступа, наличие помеченных данных в которых в момент вызова является потенциальной уязвимостью.

Санитизаторы могут как выделяться в отдельную сущность, так и отсутствовать, если описывать их как передаточную функцию, стирающую помеченность со своих аргументов.

В разделе 3.2.1 предлагается собственный формат для их задания.

При этом даже имея неограниченные ресурсы, невозможно создать спецификации для всех функций из всех возможных существующих и будущих библиотек, которые могут использоваться в анализируемых проектах. Эту проблему можно сгладить, если научить анализатор самостоятельно определять возможное поведение неизвестных внешних функций. В разделе 3.2.2 описываются эвристики, поддерживающие определение нескольких шаблонов истоков помеченности.

Также в инструмент добавлена возможность просматривать аргументы внешних функций, через которые чаще всего стиралась помеченность. Такие функции являются наиболее подходящими кандидатами для ручной спецификации пользователем в качестве передаточных функций или стоков, поскольку заранее известно, в каком количестве эти функции встречались на пути распространения помеченных данных.

3.2.1 Формат спецификаций для описания истоков, стоков и пропагаторов

Автором работы предлагается собственный формат для описания истоков, передаточных функций и стоков помеченности в текстовом виде, на основе широко известного формата JSON [84], который одновременно является человекочитаемым и имеет большое количество готовых библиотек для обработки программными средствами — а потому в него можно вносить как ручные правки, так и относительно несложно написать конвертеры для других форматов. Описание функций использует регулярные выражения и учитывает такие особенности, как перегрузка имён в языке C++.

Этот формат позволяет задать спецификации для вызовов функций, внешних или внутренних для анализируемой программы, а также для парамет-

ров функций, являющихся истоками помеченных данных. Задание поведения арифметических операций, операций копирования памяти и других обычных инструкций в программе не предусматривается и должно быть реализовано в самом детекторе, что не является проблемой, поскольку множество инструкций, которые могут встретиться в промежуточном представлении анализируемых программ, конечно и известно заранее во время разработки анализатора.

Мы будем пользоваться стандартными для JSON терминами, такими как объект, массив, строка, число, литералы `true`, `false` и `null`. Уточним только значение первых двух понятий:

1. Объект в JSON — это неупорядоченный набор пар «имя»:«значение», заключённых в фигурные скобки и разделённых запятыми, где имя — корректная строка, а значение может иметь любой из перечисленных выше типов.
2. Массив в JSON — это упорядоченная последовательность значений любого из перечисленных выше типов, заключённых в квадратные скобки и разделённых запятыми.

Спецификации для анализа помеченных данных задаются в виде трёх текстовых файлов с расширением `.json`, описывающих истоки, передаточные функции и стоки соответственно.

Содержимое каждого файла представляет собой массив объектов «описание функции», описывающих поведение одной функции или одного набора схожих функций.

В зависимости от файла, поддерживаемые поля объекта «описание функции» могут различаться, разделим их на 4 категории. Общими для всех типов полями являются:

– *Functions* – строка или массив строк, значением которых является регулярное выражение, описывающее название функции или функций, описываемых этой спецификацией
– *MatchType* – строка, имеющее одно из трёх значений: `"SHORT_NAME"`, `"FULL_NAME"` или `"QUALIFIED_NAME"`. Влияет на то, насколько подробное название функции будет передаваться для сравнения с полем *Functions*. При анализе программ на C ни на что не влияет, поскольку в нём функция однозначно определяется своим названием. Для функций и методов из языка C++ эти значения могут задавать имена

`"method"`, `"namespace::class::method"` и `"int namespace::class<T>::method(int, char)"` соответственно.

- *ParamsNum* – строка, содержащая число, диапазон чисел, записанный через дефис, или любую их комбинацию, разделённую запятыми. Для C++ этот параметр позволяет задать, сколько аргументов должно быть у функции, чтобы к ней была применима эта спецификация — это также помогает различать функции с одинаковыми именами, но разными аргументами.
- *Checker* – строка или массив строк, содержащих название детектора, реализованного в анализаторе. Это поле определяет, в каких детекторах будет использоваться данная спецификация.

Для описания остальных полей сначала потребуется ввести понятие спецификации пути доступа, которые описываются БНФ-конструкцией, приведённой в Листинге 3.1:

Листинг 3.1: БНФ-конструкция, описывающая спецификацию пути доступа

```
1 <путь доступа> ::= <базовое значение> | *(<путь доступа>)
2                    | *(<путь доступа><смещение>)
3 <базовое значение> ::= ReturnValue | Argument
4                    | Argument<номер аргумента>
5 <номер аргумента> ::= <число> | [<число>-<число>]
6 <смещение> ::= + <число> | + @ | .<имя поля>
```

Примеры корректных путей доступа: `ReturnValue`, `*(Argument2+@)`, `*(*(Argument[2-4]))`, `*(Argument.DATA)`

В отличие от понятия пути доступа, использующегося при анализе, базовыми значениями здесь могут быть только возвращаемое значение функции или некоторые из её аргументов. К ним можно прибавлять константные смещения, произвольное смещение, обозначаемое символом `@`, а также именованные поля — они являются аналогом полей структур, позволяющим абстрагироваться от конкретных реализаций таких структур данных как строки и контейнеры, определив для них собственные поля, хранящие помеченные данные. Именованные поля позволяют задавать абстрактные смещения, которые равняются только самому себе и произвольному смещению `@`.

Для задания спецификаций истоков помеченности используются следующие дополнительные поля:

- *Parameters* – строка или массив строк, каждая из которых содержит спецификацию пути доступа, становящегося помеченным после вызова данной функции.
- *ElementsCount* – необязательное поле, содержащее номер аргумента, через который в функцию передаётся количество элементов при работе с массивами.
- *ElementsSize* – необязательное поле, содержащее номер аргумента, через который в функцию передаётся размер элементов при работе с массивами.
- *Type* – необязательное поле. Если его значением является строка `"Argument"`, то эта спецификация описывает не те значения, которые оказываются помеченными в результате вызова данной функции, а те аргументы функции, через которые сторонние программы могут передавать помеченные данные — это используется как при описании истоков помеченности в библиотеках, так и для описания передачи помеченности в программу через аргументы командной строки. В этом случае в *Parameters* не используется базовое значение `ReturnValue`.

Для задания спецификаций стоков помеченности используются следующие дополнительные поля:

- *Parameters* – строка или массив строк, каждая из которых содержит спецификацию пути доступа, в которые не должны передаваться помеченные данные. В этих спецификациях не используется базовое значение `ReturnValue`, поскольку оно используется только для возврата данных из функции, но не для передачи в неё.
- *HideImplementation* – если значение этого поля установлено в `true`, то анализатор должен игнорировать настоящую реализацию функции, если таковая имеется, и использовать только спецификацию.
- *Type* – строка, содержащее название конкретного поддетектора, для которого предназначена данная спецификация.
- *CustomTag* – произвольная строка, задающая пользовательский тег, которым будут сопровождаться предупреждения с этим стоком.

Для задания спецификаций передаточных функций используются следующие дополнительные поля:

- *Transitions* – объект или массив объектов, описывающих передачу помеченности от одних аргументов функции к другим. Каждый такой объект имеет следующие поля:
  - *From* – обязательное поле, содержащее спецификации путей доступа, через которые помеченность попадает в функцию. В них не используется базовое значение `ReturnValue`.
  - *To* – обязательное поле, содержащее спецификации путей доступа, через которые помеченность выходит из функции.
  - *Copy* – если значение этого поля равняется `true`, то пути доступа копируются из *From* в *To* с сохранением внутренней структуры. Иначе путь доступа *To* становится помеченным, если помеченным был путь доступа *From*.
  - *ElementsCount* – поле, аналогичное одноимённому полю из спецификации истоков помеченности.
  - *ElementsSize* – поле, аналогичное одноимённому полю из спецификации истоков помеченности.
- *HideImplementation* – действует аналогично одноимённому полю из спецификации стоков помеченности.
- *Pure* – при выставлении этого флага анализатор будет считать, что аргументы этой функции используются только для чтения и помеченность с них никогда не стирается.
- *ReturnedAddress* – необязательное поле, которое может содержать спецификацию пути доступа, адрес которого возвращает функция. Это поле используется для описания поведения операторов `operator[]` или метода `at()`, поскольку без него сложно описать передачу помеченных данных в коллекции языка C++, которая осуществляется не вызовом метода `push()` или аналогичным, а получением адреса некоторой ячейки коллекции и записью помеченного значения по нему.

Во время анализа в каждом случае применяется только одна спецификация — если несколько спецификаций подходят под описание, то выбирается та, которая находится ближе к началу файла. Это важный момент, поскольку такое поведение позволяет описывать исключения из правил, последовательно перечисляя спецификации для всё более широких классов функций, например:

1. Метод `insert` класса-контейнера с тремя аргументами (помимо `this`), аргументы которого содержат `const`

2. Метод `insert` класса-контейнера с двумя или тремя аргументами
3. Любой метод или функция с именем `insert`

Такая иерархия может быть схематично описана конструкцией, приведённой в Листинге 3.2, где в каждой последующей спецификации описывается более обобщённый класс функций `insert`, а в каждом конкретном случае будет применяться наиболее конкретная из них.

Листинг 3.2: Пример иерархии спецификаций различных классов функций

```
[
  {
    "Functions": ".*(vector|list|...)<.*>::insert\(.*const.*\)",
    "MatchType": "QUALIFIED_NAME",
    "ParamsNum": "3",
    ...
  },
  {
    "Functions": ".*(vector|list|...)::insert",
    "MatchType": "FULL_NAME",
    "ParamsNum": "2-3",
    ...
  },
  {
    "Functions": "insert",
    "MatchType": "SHORT_NAME",
    ...
  }
]
```

В Листинге 3.3 приведены реальные примеры спецификаций для библиотечных функций из языка C, используемых анализатором.

Описанный в данном разделе формат спецификаций обладает достаточной выразительностью для описания поведения функций с точки зрения анализа помеченных данных через решение задачи IFDS.

Также он позволяет достаточно гибко описывать функции и методы языка из C++, в условиях когда для однозначного указания на конкретную функцию недостаточно её имени, а важны также дополнительные характеристики вроде пространства имён, количества и типа принимаемых ей параметров.

Листинг 3.3: Примеры реальных спецификаций, используемых анализатором

```
// Истоки
{"Functions": ["recv", "(sock_|BIO_|SSL_|ssl2?_)?read"],
  "MatchType": "FULL_NAME",
  "Parameters": "*(Argument2+@)",
  "ElementsCount": "Argument3"
}
// Передаточные функции
{"Functions": "(__)?mem(cpy|move)(_chk)?",
  "MatchType": "FULL_NAME",
  "Transitions": {"From": "Argument2+@",
                  "To": "Argument+@",
                  "Copy": true,
                  "ElementsSize": "Argument3"
                 }
}
// Стоки
{"Functions": "mem(cpy|move|set)",
  "MatchType": "FULL_NAME",
  "Parameters": "Argument3"
}
```

Поскольку создание спецификаций для всех возможных библиотечных функций невозможно, полезными для анализатора являются эвристики, позволяющие либо автоматизированно описывать истоки, передаточные функции и стоки помеченности, либо выделить те функции, написание спецификаций для которых принесёт наибольшую пользу. Вторая задача решается сбором статистики о внешних функциях, в которые в ходе анализа чаще всего передавались помеченные данные — такой подход подходит для передаточных функций и стоков. Эвристики для обнаружения новых истоков помеченности приведены в разделе 3.2.2.

3.2.2 Эвристический поиск новых истоков

Поиск новых истоков помеченности является более сложной задачей, чем поиск передаточных функций или стоков, поскольку список возможных канди-

датов нельзя определить на основе статистики, собираемой в ходе анализа: без истоков помеченности нет и самих помеченных данных.

Теоретически возможно представить анализ потоков данных в обратном направлении от известных стоков помеченности, по которому бы собиралась статистика о том, какие функции определяли значения переменных, достигающих истоков. На практике такой анализ был бы избыточным в большинстве случае, поскольку количество возможных стоков в задаче поиска уязвимостей вида «выход за границы буфера» может более чем на несколько порядков превышать количество истоков.

В данном разделе предлагается эвристика, которая проверяет сигнатуры всех библиотечных функций, используемых в анализируемой программе, и опираясь на название функции и типы её аргументов, определяет её принадлежность к одному из предусмотренных классов истоков помеченности. Также эвристика автоматически определяет номера аргументов с помеченными данными, так что для применения спецификаций в анализе от пользователя не требуется дополнительных действий. Также здесь приведены оценки применения данной эвристики на реальных проектах, процент правильно определённых функций, действительно являющихся истоками помеченных данных, а также правильность определения аргументов, через которые данные помечаются.

Эвристика разбита на несколько последовательно выполняемых этапов.

На первом этапе происходит начальный отбор кандидатов на основе наличия ключевых слов в названии функции или метода, характерных для выбранного класса истоков, который выглядит следующим образом:

1. Найти все функции, в названии которых присутствует ключевое слово, без учёта регистра.
2. Исключить те из них, у которых ключевое слово присутствует в названии класса или области определения.
3. Исключить тех кандидатов, в которых ключевое слово не соответствует одному из трёх стилей записи “word”, “Word” или “WORD”, или является частью другого слова, как “read” в “thread”.
4. Дополнительно исключаются те кандидаты, которые содержат слова из чёрного списка.

После того как первоначальный список кандидатов определён, происходит проверка типов формальных параметров функции — она зависит от искомого

класса истоков. В данной работе предлагаются эвристики для двух классов истоков: «освобождение памяти» и «чтение данных».

Для истоков класса «освобождение памяти» в качестве ключевых были использованы слова “free”, “delete” и “remove”. Чтобы считаться истоком, функция должна иметь ровно один формальный параметр. Тип этого параметра должен быть указателем, который не указывает ни на константный тип, ни на другой указатель. Освобождаемой (помеченной, с точки зрения детектора) считается область памяти, на которую указывает единственный параметр функции.

Для истоков класса «чтение данных» использовалось ключевое слово “read”, а в чёрный список входит слово “random”, поскольку функции считывания последовательности случайных чисел вряд ли могут быть использованы в качестве источника данных, на которые влияет злоумышленник. Функция должна иметь только один формальный параметр, являющийся указателем на неконстантный целочисленный тип данных — он считается параметром, в который записываются считанные данные. Если дополнительно у функции ровно один параметр является целочисленным, то предполагается, что через него она может принимать объём считываемых данных.

Оценка результатов работы этих эвристик на реальном проекте приведена в Таблице 5 со следующими столбцами:

1. *Всего найдено* — общее количество функций, распознанных в качестве истоков помеченных данных выбранного класса.
2. *Функция* — процент найденных функций, которые действительно являются истоками.
3. *Параметр* — процент найденных функций, для которых был верно определён путь доступа, через который данные помечаются.
4. *Размер* — процент найденных функций, для которых был верно определён параметр, задающий размер считываемых данных — используется только для класса истоков «чтение данных».

В строке «Чтение данных» вне найденные функции были проверены вручную, в строке «Освобождение» были проверены случайным образом выбранные 20 функций.

Эти данные показывают, что на проекте OpenSSL эти эвристики обладают достаточной точностью, чтобы использоваться в автоматическом режиме — без необходимости создания спецификаций вручную, поскольку параметр определялся неверно только в 7% и 10% случаев соответственно. В случае класса

Таблица 5 — Результаты эвристики обнаружения истоков на OpenSSL

Тип истока	Всего найдено	Функция	Параметр	Размер
Чтение данных	15	100%	93%	47%
Освобождение	175	100%	90%	—

истоков «чтение данных» в 47% случаев также верно определялся параметр, задающий размер считываемой области данных, который не является обязательным для анализа помеченных данных, но в некоторых случаях может улучшить его точность.

Такое большое количество найденных истоков класса «освобождение памяти» объясняется наличием в OpenSSL макроса, позволяющего объявлять функции `typename_free` для любого типа `typename`. Часть обнаруженных функций принимали в качестве аргумента аналог «умных» указателей — в таблице они подсчитаны как имеющие неправильно распознанный параметр, поскольку функция освобождает не саму структуру, принимаемую в качестве аргумента, а одно из её полей. Наличие таких эвристик в анализаторе сильно упрощает задачу создания спецификаций для новых истоков помеченности, характерных для конкретного проекта, а потому важно для повышения полноты анализа.

Качество распознавания функций могло бы быть повышено за счёт учёта названий параметров, а не только их типов, но эта информация недоступна во внутреннем представлении LLVM.

Стоит также отметить, что в современных реалиях задачу классификации функции по её названию и типам параметров можно было бы решать при помощи инструментов машинного обучения: например, путём обучения нейросетей или применения больших языковых моделей (LLM). Хотя эти подходы являются более гибкими, чем алгоритмические, они не лишены недостатков:

1. Для обучения нейросети с нуля необходим большой набор данных с уже размеченными функциями, как являющимися истоками помеченных данных, так и не являющихся ими. При этом вероятно, что для разных классов помеченных данных требуется обучать отдельную модель.

2. Существует большой выбор из готовых больших языковых моделей, которые могут быть запущены локально на компьютере пользователя и применяются к разнообразным задачам без необходимости дополнительного обучения — только за счёт настройки запроса к модели. Многие из них поддерживают применение БНФ-грамматик, что позволяет формализовать ответ модели, чтобы его можно было разбирать автоматически. Тем не менее, даже относительно с относительно небольшими моделями процесс получения результатов для всех функций в программе может растянуться на часы, а применение наиболее популярных на данный момент моделей с более чем 30 миллиардами параметров выставляет значительно более высокие требования к компьютеру пользователя, чем сам анализ помеченных данных.
3. Доступ к большим языковым моделям как к услуге снимает требования к аппаратному обеспечению и позволяет использовать самые точные и передовые модели, однако их ключевым недостатком является необходимость оформления платной подписки, а также передача данных из запросов третьей стороне, что является неприемлемым в ряде случаев. Также не решается проблема времени, которое требуется для анализа сотен или тысяч сигнатур функций.

Таким образом, предложенные в данном разделе эвристики для автоматического определения новых истоков показали себя пригодными для использования без вмешательства человека и имеют свою нишу для применения несмотря на появление более современных, не алгоритмических подходов.

В данной главе был предложен алгоритм анализа косвенных вызовов на основе IFDS, позволивший повысить общее количество предупреждений за счёт повышения полноты анализа.

Также в данной главе были рассмотрены способы описания поведения внешних функций, как через составляемые вручную спецификации, так и позволяющие без участия человека находить новые, ранее неизвестные истоки помеченных данных.

Результаты третьей главы опубликованы в работах [3—6; 8].

Глава 4. Методы повышения масштабируемости статического анализа помеченных данных

Данная глава посвящена описанию предлагаемых методов повышения масштабируемости анализа, которая с точки зрения теории характеризуется временной и пространственной сложностью, а с точки зрения конкретной реализации — зависимостью времени анализа и необходимого объёма оперативной памяти от размера анализируемого проекта. Масштабируемость важна для возможности анализа реальных проектов на миллионы строк кода.

Первый раздел посвящён предлагаемому алгоритму направленного распространения глобальных переменных, который позволяет не передавать помеченные факты, основой которых являются глобальные переменные, в те функции, на поведение которых они не влияют. Необходимость такого алгоритма обнаружилась на тестовом наборе Juliet Test Suite, где анализ 6% тестов занимал больше 50% времени анализа — эта особенность проявилась именно на тестах, в которых помеченные данные передавались через глобальные переменные.

Второй раздел описывает изменения в использовании решателя задачи IFDS и используемые ограничения анализа.

Наибольшее влияние на масштабируемость анализа на основе IFDS оказывает необходимость построения и хранения расширенного суперграфа всей анализируемой программы, описывающего направления межпроцедурных потоков данных. Он состоит из $|N| \times (|D| + 1)$ вершин, где N — множество вершин суперграфа, которое линейно растёт с увеличением программы, а D — множество возможных помеченных фактов. В нашей постановке задачи множество D ограничено только множеством чисел, используемых для обозначения смещения, а также допустимом количестве смещений в одном пути доступа, потому необходимостью является использование модификации алгоритма [63], по которой используется ленивое построение расширенного суперграфа, который и содержит только те вершины и рёбра, которые действительно участвуют в распространении помеченных данных. В лучшем случае в анализируемой программе не будет истоков помеченности и анализ завершится мгновенно, в худшем случае граф построится целиком и применение модификации алгоритма на его размер не повлияет.

Альтернативным методом межпроцедурного анализа проектов произвольного размера, является анализ на основе резюме. Он подразумевает последовательность внутрипроцедурных анализов функций с созданием их резюме, обобщающих их поведение, и применением этих резюме в местах вызова функции — при этом каждая функция обходится только один раз. Такой подход хорошо масштабируется, поскольку с увеличением общего размера программы растёт только суммарный объём сохраняемых резюме, которые необязательно постоянно хранить в оперативной памяти. Тем не менее, у него есть несколько недостатков, влияющих на полноту анализа:

1. Вызываемые функции должны анализироваться перед вызывающими, из-за чего анализ испытывает сложности с рекурсивными вызовами и циклами в графе вызовов, поскольку резюме для вызываемых функций ещё не построены.
2. Заранее не известно, какие сведения о поведении функции понадобятся в ходе анализа, из-за чего приходится выбирать между излишне большими резюме и потерей части информации. Это вызвано тем, что структуры в программе могут содержать десятки и даже сотни полей, в том числе вложенных.

Таким образом, алгоритм табуляции для решения задачи IFDS может обеспечить бóльшую полноту анализа, а также предоставляет возможность «ленивого» анализа, при котором анализ осуществляется только для тех вершин и путей доступа, которые действительно будут содержать помеченные данные — однако хуже масштабируется на действительно большие программы из-за того, что размер расширенного суперграфа на порядки превышает сумму размеров всех графов потока управления в анализируемой программе.

С практической точки зрения проблема высокого потребления памяти при реализации анализа помеченных данных через решение задачи IFDS также хорошо известна. В первую очередь она следует из необходимости построения межпроцедурного графа распространения помеченности для всей программы и отмечалась как пользователями инструмента Flowdroid, так и его авторами: в частности, в [85] утверждается, что даже на системе с 730 Гб оперативной памяти и 64 процессорными ядрами, при анализе набора из 2950 Android приложений встретилось 16 приложений, для анализа которых не хватило памяти или сработало ограничение на время анализа в 24 часа. Поскольку на боль-

шинстве пользовательских устройств доступные ресурсы меньше на 1, а то и 2 порядка — проблема масштабируемости является актуальной.

4.1 Направленное распространение глобальных переменных

В данном разделе обсуждается вопрос влияния на масштабируемость анализа наличия помеченных фактов с путями доступа, базовым значением для которых являются глобальные переменные. Сначала перечислим, какие типы базовых значений могут встречаться в путях доступа:

1. *Локальные значения* соответствуют локальным переменным функции, а также различным временным значениям, возникающим в ходе вычислений. С точки зрения распространения помеченности их особенность состоит в том, что локальных значений может быть много, но использующие их факты существуют только в пределах своей функции: они передаются в вызываемые функции только при явном указании их в качестве аргумента вызова, а возвращаются из текущей функции только совпадают с одним из её формальных параметров или явно используются в операторе `return`.
2. *Параметры* соответствуют формальным параметрам функции, через которые она принимает данные. Как правило именно факты, базовым значением которых являются параметры, фигурируют на межпроцедурных рёбрах в графе распространения помеченности — при этом их количество в функции обычно сильно ограничено. В случае языка C++ анализатор также предполагает, что у каждого нестатического метода класса есть неявный параметр `this`.
3. *Глобальные значения* соответствуют глобальным переменным, объявленным в программе. Их особенностью является то, что факты с глобальным базовым значением всегда передаются в каждую вызываемую функцию и всегда возвращаются при выходе из текущей функции.

Из этих замечаний следует, что несмотря на то что глобальные значения составляют лишь незначительный процент от общего числа значений в программе, они могут оказывать сильное влияние на время анализа. В частности, на

тестовом наборе Juliet Test Suite, больше 50% времени анализа пришлось на 6% тестов — именно те, которые содержали помеченные переменные.

Чтобы это было проще продемонстрировать, сформулируем несколько предположений:

1. Любой возникающий факт (кроме пустого) имеет базовое значение, принадлежащее к одному из трёх перечисленных выше классов. Это следует из введённых нами определений.
2. Обозначим через h максимальное количество путей доступа от одного базового значения но с разными смещениями, которые могут возникать в ходе анализа. Предположим, что число h конечно.
3. Максимальное количество параметров любой функции ограничено константой. Например, в случае языков C и C++ стандарт гарантирует поддержку только 127 и 256 параметров функции соответственно — на практике же их количество редко превышает десяток.
4. Увеличение общего размера проекта происходит за счёт увеличения количества классов и функций, а не их размера.

Рассмотрим утверждение о влиянии глобальных переменных на масштабируемость анализа на примере самого простого класса IFDS задач — локально-разделимых. В работах [15; 45] доказывалось, что для таких задач алгоритм табуляции для решения задачи IFDS имеет сложность $O(ED)$, где E — количество рёбер межпроцедурного графа потока управления, а D — максимальный размер домена фактов среди всех процедур. Исходя из этого можно конкретизировать это утверждение для используемых нами типов фактов и сформулировать Утверждение 1.

Утверждение 1. *Для локально разделимых задач в описанных выше предположениях сложность алгоритма решения задачи IFDS можно оценить как $O(E(D_l + D_g)h)$, где E — количество рёбер межпроцедурного графа потока управления, D_l — максимальное количество локальных значений в одной функции, D_g — количество глобальных переменных в программе, а h — максимальное количество различных путей доступа от одного базового значения, которые могут возникать в ходе анализа. При этом ED_l зависит от общего размера программы линейно, а ED_g — квадратично.*

Обоснование. То, что сложность алгоритма табуляции является полиномиальной и для локально разделимых задач составляет $O(ED)$, где E —

количество рёбер межпроцедурного графа управления программы, а D — количество помеченных фактов, возникающих в ходе анализа, уже было доказано в [15; 45]. И E и D зависят от размера анализируемой программы линейно, однако для любой конкретной точки программы N множество возможных помеченных фактов в ней D_N можно разбить на следующие составляющие: $D_N = D_p \cup D_l \cup D_g$, где D_p — множество фактов с параметром текущей функции в качестве базового значения, D_l — множество фактов с локальным базовым значением из текущей функции, D_g — множество фактов с глобальным базовым значением. При этом количество параметров функции можно ограничить константой, а максимальное количество локальных переменных любой отдельно взятой функции также зависит не от общего размера анализируемой программы, а от того, насколько сложные функции допускаются стилем кодирования, принятом в проекте. Из этого следует, что $E(D_p + D_l)$ на практике имеет линейную зависимость от размера проекта, а ED_g — квадратичную. Последнее объясняется тем, что если в глобальную переменную записываются помеченные данные, то соответствующий помеченный факт будет распространяться по всем межпроцедурным рёбрам, даже если во всей программе нет других обращений к этой переменной. □

Утверждение 1 показывает, что асимптотическое влияние количества помеченных глобальных переменных на время анализа выше, чем у локальных значений и параметров функций. Это согласуется с замечаниями авторов изначального алгоритма [15], которые упомянули, что обычно бóльшую часть множества фактов для каждой отдельной процедуры будут составлять факты, хранящие информацию о глобальных переменных, общих для всей программы. Это демонстрирует актуальность темы оптимизации распространения помеченности через глобальные переменные.

В данной работе предлагается алгоритм направленного распространения для помеченных путей доступа, базовым значением которых являются глобальные переменные. Для его работы необходимо наличие графа вызовов функций, а также, чтобы для каждой переменной был доступен список её использований в программе. Сам алгоритм состоит из двух этапов: на предварительном этапе для каждой глобальной переменной G вычисляются множества F^G_{direct} и $F^G_{indirect}$. Второй этап используется во время решения задачи IFDS и изменяет правила, по которым факты распространяются по внутрипроцедурным рёбрам графа потока управления.

На предварительном этапе для каждой глобальной переменной G все функции в программе разбиваются на три непересекающихся подмножества:

1. F^G_{direct} — функции, непосредственно использующие G, то есть те, в коде которых есть явное упоминание этой переменной (например, её значение считывается или переписывается).
2. $F^G_{indirect}$ — функции, косвенно использующие G, то есть те, в которых G явно не упоминается, но есть вызовы функций, непосредственно или косвенно использующих переменную.
3. Все остальные функции, которые не используют G и не влияют на него.

Множество F^G_{direct} вычисляется первым: для каждого использования переменной G, в множество F^G_{direct} добавляется функция, содержащая это использование.

В множество $F^G_{indirect}$ добавляются все функции, встреченные в процессе обхода графа вызовов в обратном направлении, начиная с функций из множества F^G_{direct}, за исключением самих функций из F^G_{direct}.

На втором этапе вносится изменение в алгоритм распространения фактов. Каждый раз, когда в оригинальном алгоритме в расширенный суперграф добавляется внутрипроцедурное ребро, в изменённом алгоритме вместо этого выполняется одно из двух действий:

1. *continue()* — распространение помеченного факта осуществляется по обычным правилам.
2. *move(vertices)* — вместо обычных, создаются рёбра к вершинам графа потока управления, перечисленным в *vertices*. Все вершины должны принадлежать той же функции, что и текущая.

Для фактов с путём доступа, базовое значение которого не является глобальной переменной, всегда выбирается действие *continue()*. Для глобальных фактов выбор действия осуществляется по Алгоритму 3.

Другими словами, если базовым значением текущего помеченного факта является глобальная переменная, то его распространение зависит от того, к какому множеству относится текущая функция:

1. Для непосредственно использующих функций, факт распространяется по обычным правилам.
2. Для косвенно использующих функций, факт передаётся в ближайшие вызовы косвенно или непосредственно использующих функций, или к выходу из текущей функции.

Алгоритм 3 Направленное распространение помеченной глобальной переменной

Входные данные: Текущая функция F, текущая вершина ГПУ N, множества F_{direct} и $F_{indirect}$ для текущего факта G

Выходные данные: *result*, содержащий либо результат выполнения *continue()* либо результат выполнения *move()* с некоторым множеством вершин

if $F \in F_{direct}$ **then**
  $result \leftarrow$ CONTINUE()
else if $F \in F_{indirect}$ **then**
  $vertices \leftarrow \varnothing$
  ⟨поиск в ширину, исследующий ГПУ текущей функции начиная с вершины N. Если очередная вершина является вызовом функции, входящей в $F_{direct} \cup F_{indirect}$, или выходом из текущей функции — такая вершина добавляется в $vertices$, иначе в очередь поиска добавляются вершины, следующие за текущей, если они встретились впервые⟩
  $result \leftarrow$ MOVE($vertices$).
else
  $vertices \leftarrow$ {F.GET_RETURN_VERTEX()}
  $result \leftarrow$ MOVE($vertices$).
end if

3. В остальных функциях факт сразу передаётся в точку выхода из функции в неизменном виде.

Для обоснования корректности этого алгоритма и того, что его применение не изменяет результаты анализа, можно сформулировать Теорему 1:

Теорема 1. *В предположении отсутствия в программе межпроцедурных псевдонимов для глобальных переменных, если для любой пары ⟨исток, сток⟩ обозначить через P_0 и P_1 множества путей распространения помеченности между ними, обнаруживаемых оригинальным и модифицированным алгоритмами соответственно, то:*

1. *$\forall p \in P_0, \exists q \in P_1$: q можно получить из p удалением некоторых точек, в которых базовым значением факта является глобальная переменная.*

2. $\forall p \in P_1, \exists q \in P_0 : q$ *можно получить из* p *добавлением точек, базовым значением факта в которых является глобальная переменная.*

Доказательство. Построение графа распространения помеченности начинается с истоков помеченности: множество истоков вычисляется одинаково в обоих вариантах алгоритма, потому начальное множество вершин совпадает.

Для фактов, базовым значением которых не является глобальное значение или его псевдоним, распространение помеченности происходит по тем же правилам, потому для них из одних и тех же вершин расширенного суперграфа на входе будут строиться одинаковые исходящие рёбра и добавляться одинаковые вершины в очередь обработки.

Рассмотрим отдельно три варианта того, к какому из классов может принадлежать текущая функция при распространении помеченного факта D с глобальной переменной G в качестве базового значения:

1. Если функция относится к классу F^G_{direct}, то алгоритм не изменяет правила распространения фактов в ней, а потому для тех же вершин будут создаваться те же исходящие из них рёбра.
2. Если функция не относится ни к F^G_{direct}, ни к $F^G_{indirect}$, то:
   - Все вызываемые в ней функции также не принадлежат к этим множествам, иначе она сама относилась бы к $F^G_{indirect}$.
   - В ней нет псевдонимов для переменной G.
   - Любое внутрипроцедурное исходящее ребро ведёт в соседнюю вершину ГПУ с тем же фактом D.
   - Ни в самой функции, ни в вызываемых из неё нет стоков, в которых использовался бы факт D или его псевдоним.

   Таким образом, если в изначальном алгоритме существовал путь распространения помеченности, проходящий через пару $\langle$текущая вершина, $D\rangle$ и достигающий некоторого стока, то за ней в пути обязательно присутствует пара $\langle$выход из текущей функции, $D\rangle$, а все пары между ними также содержат факт D.
3. Если функция относится к классу $F^G_{indirect}$, то в ней нет ни одного упоминания переменной G — но такие упоминания есть в одной или нескольких вызываемых из неё функциях. Из этого следует, что у помеченного факта D нет других псевдонимов в данной функции, он передаётся без изменений во всех вершинах кроме, возможно, вызовов

функций, принадлежащих множеству F_{direct} или $F_{indirect}$, пока не достигнет выхода из текущей функции. Упомянутый алгоритм заменяет «прямые» участки графа, на которых глобальная переменная распространяется без изменений на одно ребро из начала участка в конец, после чего помеченный факт распространяется по обычным правилам — передаётся в вызываемую функцию или возвращается из текущей.

Таким образом получается, что граф распространения помеченности формируется по тем же правилам, что и без применения алгоритма, за исключением некоторых участков, на которых помеченная глобальная переменная распространяется без изменений и не может встретить сток помеченности — такие участки могут быть упрощены путём замены на ребро, ведущее из начала участка в его конец, что не влияет на остальную часть графа. □

Оценка работы алгоритма направленного распространения помеченности приведены в Таблице 6.

Таблица 6 — Результаты тестирования алгоритма

Проект	Время (до)	Время (после)	Память (до)	Память (после)
Juliet	59м 48с	26м 46с (−55%)	2,02 Гб	1,8 Гб (−11%)
OpenSSL	14м 50с	14м 6с (−5%)	1,31 Гб	1,28 Гб (−2%)

Наибольший эффект этот алгоритм продемонстрировал на Juliet Test Suite, обеспечив ускорение анализа более чем в 2 раза — это связано с большим количеством помеченных глобальных переменных в тестовом наборе, а также с тем, что каждая из них используется только в «своём» тесте, вследствие чего множества F_{direct} и $F_{indirect}$ состояли всего из нескольких функций.

На проекте OpenSSL данный алгоритм уменьшил время анализа только на 5%, а объём используемой памяти на 2% — это объясняется тем, что на данном проекте помеченные данные через глобальные переменные передавались редко.

4.2 Практические изменения в использовании IFDS решателя

Основные проблемы масштабируемости, которые обнаружились при практическом использовании решателя задачи IFDS для поиска уязвимостей, заключаются в следующем:

1. Размер хранимого графа распространения помеченности увеличивается пропорционально размеру анализируемого проекта и на больших проектах перестаёт помещаться в оперативную память.
2. Обнаружились шаблоны анализируемого кода, при которых анализ IFDS не мог завершиться и мог продолжаться бесконечно, пока не сработает ограничение на глубину анализа или общее количество проанализированных вершин.

Для решения первой проблемы был использован раздельный анализ истоков помеченности. При таком подходе для каждого истока создаётся отдельный решатель задачи IFDS и строится независимый от других граф распространения помеченности. Основным недостатком такого подхода является повторный анализ общих участков графов, что в худшем случае может привести к замедлению в N раз, где N — количество истоков в программе. На практике регулярно встречались небольшие группы схожих истоков, для которых совместный анализ является более выгодным, однако в среднем графы распространения помеченности от разных истоков либо не пересекаются, либо имеют лишь небольшие общие участки, а потому разница в скорости анализа не так ощутима.

Основными преимуществами раздельного анализа является то, что:

1. Общее потребление памяти не зависит от количества истоков.
2. Не возникает ситуация, при которой продолжительный анализ одних истоков истощает ограничения на анализ других истоков.
3. Анализ программы легко распараллелить, так как анализ каждого истока может быть вынесен в отдельный процесс.

Вторая проблема вызвана тем, что в нашем инструменте представление путей доступа к помеченным данным отличается от того, которое описывали авторы Flowdroid — это вызвано разницей между анализируемыми языками. В [16] путь доступа задаётся как базовая переменная или параметр и конечная последовательность взятия полей от неё — по умолчанию не больше пяти.

Элементы массивов в нём считаются неразличимыми между собой. Такой подход хорошо подходил для объектно-ориентированного языка Java и позволял разумно ограничить множество D помеченных фактов, которые могут возникать в ходе анализа.

В данной работе рассматривается вопрос анализа достаточно низкоуровневого языка C, а также C++. В них также существует понятие поля класса или структуры, однако информация о них теряется при приведении типов к `void *` или `char *` — например, при передаче указателя на структуру или объект в функцию копирования памяти. По этой причине, а также из-за использования биткода LLVM в качестве промежуточного представления анализируемой программы, в данной работе используется определение путей доступа, в которых вместо взятия полей используются целочисленные смещения и разыменования указателей независимо от заявленного типа базового значения.

Такой подход помогает не терять точность при преобразовании указателей и лучше соответствует модели памяти, используемой в языках C и C++, однако приводит к тому, что циклы и рекурсии, смещающие указатели на помеченные данные, могут создавать бесконечное количество новых фактов, как показано в Листинге 4.1.

Листинг 4.1: Пример цикла, генерирующего новые помеченные факты

```
1 void foo(char *p) {
2   *(p + 1) = TaintSource();
3   // Помеченный путь доступа: *(p + 1)
4   while (*p) {
5     char c = *(p++);
6     // Помеченные пути доступа: *p, *(p-1), c, *(p-2), *(p-3)...
7     TaintSink(c);
8   }
9 }
```

В этом примере помеченные данные, полученные через вызов истока `TaintSource()`, достигают стока `TaintSink()` через переменную `c` на второй итерации цикла. При этом из-за выполнения оператора `p++`, целочисленное смещение в пути доступа к помеченным данным через указатель `p` на каждой итерации уменьшается на 1.

В отсутствие ограничений это означало бы невозможность завершить анализ, а также трату времени на распространение новых помеченных фактов, которые скорее всего не приведут к новым стокам помеченности, что является

серьёзной проблемой. Выставление ограничений на максимальную глубину анализа и размер графа распространения помеченности от одного истока помогает избежать бесконечного анализа, однако масштабируемость остаётся проблемой: если анализ большинства истоков будет завершаться только по ограничению на время анализа, то с увеличением количества истоков либо общее время анализа станет слишком большим, либо ограничения придётся уменьшать до слишком низких значений.

Из опробованных нами эвристик и ограничений для решения этой проблемы наиболее универсальным нам показалось выделение обратных рёбер в графе потока управления и графе вызовов и ограничение на количество повторных проходов по ним. К каждому элементу в очереди выполнения решателя IFDS добавляется вспомогательная информация, в данном случае отмечающая, какие обратные рёбра были посещены в ходе распространения текущего факта и сколько раз. Если при переходе через обратное ребро счётчик его посещений превышает заранее заданную константу N, то факт дальше не передаётся. В зависимости от значения N:

1. При $N = 0$ фактам запрещено пересекать обратные рёбра — это означает невозможность анализа рекурсивных вызовов, а также то, что помеченные данные не могут покинуть цикл с предусловием, что является слишком сильным ограничением полноты.
2. При $N = 1$ каждое обратное ребро может быть пройдено помеченным фактом только однажды, что даёт обнаруживать проблемы, затрагивающие две итерации цикла, но ограничивает порождение новых фактов.
3. Дальнейшее увеличение константы N увеличивает теоретическую полноту анализа, однако почти не сокращает общее время анализа, по сравнению с вариантом без ограничений.

Значение $N = 1$ хорошо подходит для эффективного анализа, поскольку оно достаточно мало, чтобы не замедлять анализ большим количеством порождаемых фактов. Поскольку при решении задачи IFDS не отслеживаются конкретные значения переменных, а только факт помеченности областей памяти, то в отличие от более точных видов анализа, одной итерации цикла достаточно для нахождения большинства фактов, помечаемых внутри цикла.

Хотя теоретически возможно придумать цикл, для полного анализа которого необходимо любое наперёд заданное количества итераций цикла, как например на Листинге 4.2, на практике они встречаются реже.

Листинг 4.2: Пример цикла, в котором для достижения стока требуется анализ двух итераций

```
1 void foo(int i) {
2   char *a = TaintSource();
3   char *b, *c;
4   while (i--) {
5     c = b;
6     b = a;
7     TaintSink(c);
8   }
9 }
```

Помимо этого, при N равном 1, анализ по-прежнему анализирует две итерации, так что сток будет достигнут, но факты со второй итерации будут распространяться только в пределах самого цикла.

В данной главе был предложен алгоритм направленного распространения помеченности через глобальные переменные, позволяющий значительно сократить количество выполняемых итераций алгоритма IFDS для таких переменных при помощи информации об их использовании в программе.

Также в данной главе были описаны проблемы масштабируемости, которые возникают при анализе кода на языка C и C++ с использованием выбранного нами определения путей доступа. В частности указывается на то, что в некоторых условиях циклы и рекурсии способы порождать неограниченное количество помеченных фактов, из-за чего полный анализ программы становится невозможным — вместо этого предложены ограничения на количество итераций цикла и переходов по обратным рёбрам в графе вызовов.

Результаты четвёртой главы опубликованы в работах [3—8].

Глава 5. Особенности реализации и тестирование инструмента Irbis

Данная глава состоит из трёх разделов. Первый раздел описывает общую схему работы инструмента Irbis. Второй раздел описывает набор детекторов, реализованных в нём. Третий раздел описывает особенности реализации и эвристики, оказавшиеся полезными для практической применимости инструмента. Четвёртый раздел даёт общую оценку работы анализатора на наборе проектов с открытым исходным кодом.

5.1 Общая схема работы

В данной работе предполагается следующая общая схема работы инструмента, показанная на Рисунке 5.1:

1. Система перехвата сборки, предоставляемая инструментом Svace, переводит файлы исходного кода в файлы промежуточного представления LLVM.
2. Компоновщик собирает отдельные файлы промежуточного представления в единый анализируемый файл.
3. Инструмент загружает анализируемый файл и выполняет подготовительные этапы: построение суперграфа, поиск набора истоков для каждого типа детекторов, анализ косвенных вызовов и т.д.
4. В каждом детекторе выполняется решение задачи IFDS для каждого истока отдельно, результатом анализа является множество достигнутых стоков и расширенный суперграф, описывающий распространение помеченности.
5. При помощи чувствительного к путям обхода расширенного суперграфа строится новый граф, вершины которого дополнены предикатами и в котором любой путь является консистентным с точки зрения выбранных критериев. Для каждого достигнутого стока из графа выбирается наиболее короткая трасса.
6. Если планируется двухэтапный анализ, то модуль инструментации добавляет в файл биткода анализируемой программы информацию о

найденных трассах для последующей проверки сторонним инструментом.

7. Web-интерфейс, предоставляемый инструментом Svace, отображает все найденные трассы, используя информацию с этапа перехвата сборки для подсветки синтаксиса и возможности навигации по коду.

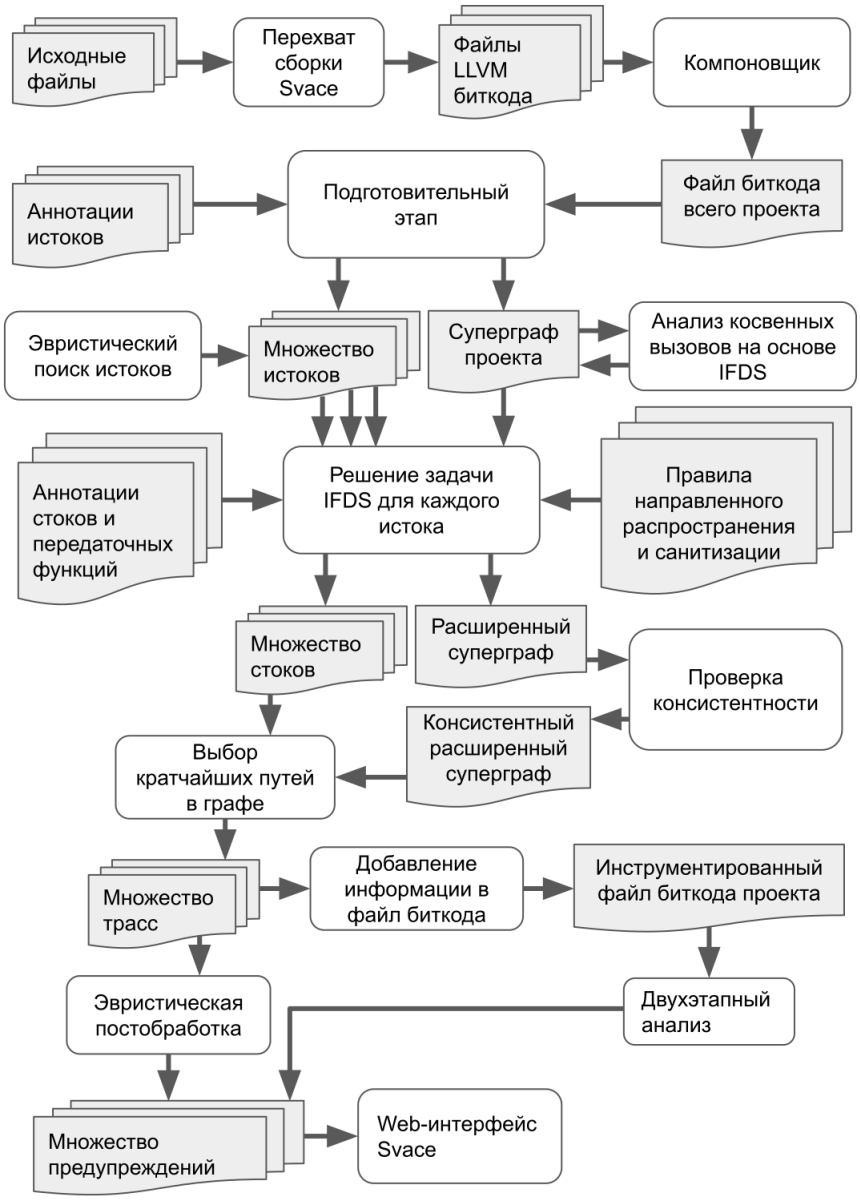

Рисунок 5.1 — Схема работы инструмента Irbis

Де-факто инструмент является частью инфраструктуры Svace, поскольку Irbis полагается на предоставляемые ей способы перехвата сборки анализируемого проекта и отображения результатов анализа, а его анализ может быть вызван из Svace. Несмотря на это, его реализация анализа является полностью независимой и не использует никакие компоненты анализатора Svace.

Также в Irbis активно используется открытая компиляторная инфраструктура LLVM, которая используется для открытия промежуточного представления анализируемой программы, компоновки, инструментации файлов, а также предоставляет такую информацию как деревья доминаторов, список использований переменных и др.

5.2 Реализованные детекторы

Как упоминалось в Главе 1, само понятие «помеченных данных» является абстрактным, а набор истоков, передаточных функций, санитайзеров и стоков помеченности зависит от того, какой тип уязвимости планируется искать. Детекторы уязвимостей в инструменте Irbis разбиты на 4 большие группы, характеризующихся следующими типами помеченности:

1. *Ненадежные входные данные* соответствуют данным, получаемым из внешнего источника, которые могут быть непредусмотренным образом изменены потенциальным злоумышленником: это может быть текст слишком большой длины или содержащий специальные символы, слишком большое или наоборот – отрицательное число и не только. Ненадежные данные могут быть возникать в результате вызова некоторой функции-истока, так и являться формальными параметрами функции, например, `argv` в функции `main`. Попадание таких данных без проверки в критическую инструкцию, такую как использование числа в качестве индекса массива, или функцию, такую как `exec` или `memcpy` может приводить к непредсказуемым ошибкам памяти или даже позволять злоумышленнику выполнять произвольный код. Этот тип помеченных данных используется в большинстве детекторов, реализованных в нашем инструменте.
2. *Чувствительные данные* соответствуют данным, содержащим пользовательскую информацию, доступ к которой может быть получен в самой программе, но которая не должна передаваться третьим лицам. Примерами такой информации могут являться криптографические ключи, пароли, информация о компьютере пользователя и не только. В зависимости от анализируемой программы, запись их в файлы, вывод

на экран или отправка по сети без шифрования могут рассматриваться как уязвимости «утечка данных».

3. *Аргументы функций безопасности*, такие как ключи шифрования, криптографическая соль и пароли являются истоком для детекторов вида «использование константных паролей». Особенностью этого типа помеченности является то, что она распространяется в обратном направлении: от использования в функции безопасности до объявления переменной или записи в неё константного значения.
4. *Освобождённая память*, полученная в результате вызова функции `free()` или `delete` является истоком в детекторах использования освобождённой памяти и повторного освобождения. В зависимости от контекста, это может приводить как ошибкам памяти, так и к считыванию данных, которые программист считал уже удалёнными.

Для первой группы предлагается более детальная классификация обнаруживаемых потенциальных уязвимостей:

- чтение или запись за пределами ожидаемой области памяти (CWE-121, 122, 124, 126, 127),
- ошибки, связанные с форматной строкой (CWE-134),
- выделение слишком большого участка памяти (CWE-789),
- целочисленное переполнение при выделении памяти (CWE-680),
- отказ в обслуживании, например, при использовании ненадежных данных в условии цикла для контроля количества итераций (CWE-400), а также попадание помеченных данных в другие чувствительные конструкции управления,
- использование ненадежных данных в критических функциях, что может приводить, например, к запуску команд с повышенными привилегиями (CWE-78), внедрению данных в БД (CWE-89, CWE-90).

Более полный список детекторов приведён в Таблице 7. Детекторы с одинаковыми типами истоков, как правило, реализуются в виде единого детектора путём объединения их множеств стоков — такой подход позволяет не строить несколько графов распространения помеченности на один и тот же исток.

Детектор	**Исток**	**Сток**
TAINTED_ARG	ненадежные данные	список критических функций
TAINTED_PTR.LOAD	ненадежные данные	инструкция чтения памяти по помеченному указателю
TAINTED_PTR.STORE	ненадежные данные	инструкция записи в память по помеченному указателю
TAINTED_LOOP_CONDITION	ненадежные данные	условие цикла
SETTING_MANIPULATION	ненадежные данные	сохранение в качестве настройки, например, `sethostname`
PROCESS_CONTROL	ненадежные данные	запуск процесса, например, `exec`
SQL_INJECTION	ненадежные данные	использование в SQL запросе
RESOURCE_INJECTION	ненадежные данные	использование в качестве идентификатора ресурсов, например, пути файловой системы
GETLOGIN	`getlogin` (ненадежные данные)	сравнение с константой
GETHOSTBY	`gethostbyaddr` (ненадежные данные)	сравнение с константой
DATA_LEAK	чувствительные данные	печать, отправка по сети, сохранение в файл и т.д.
HARDCODED_PASSWORD	криптографические функции	константная строка
HARDCODED_PASSWORD_EXTRA	регулярное выражение имени переменной	константная строка
HARDCODED_SALT	криптографические функции	константная строка
HARDCODED_KEY	криптографические функции	константная строка
USE_AFTER_FREE	освобожденная память	разыменование указателя на освобожденную память
PASSED_TO_PROC_AFTER_FREE	освобожденная память	использование в качестве фактического аргумента в вызове
DOUBLE_FREE	освобожденная память	`free(), delete`

Таблица 7 — Список основных детекторов инструмента Irbis

5.2.1 Теги предупреждений

Для того чтобы разделить предупреждения в рамках одного детектора по подтипам или по уровню достоверности используются *теги*. Они добавляются к типу предупреждения, группируя их и позволяя пользователю просматривать в первую очередь те предупреждения, которые с большей вероятностью могут соответствовать настоящей уязвимости. Предупреждения могут иметь более одного тега одновременно.

Список используемых в инструменте тегов:

- `BUFFER_LENGTH` используется для выделения предупреждений, связанных с переполнением буфера, в которых у Irbis оказалось достаточно информации о размерах задействованных буферов, а потому само предупреждение с большей вероятностью является истинным.
- `HEURISTIC_SOURCE` используется для предупреждений, исток для которых был получен при помощи эвристик. В таких предупреждениях нужно в первую очередь смотреть, правильно ли был определён исток и если да — имеет смысл добавить найденную функцию в спецификации.
- `INCONSISTENT` используется для предупреждений, в которых выбор кандидатов для вызова функции в разных участках пути не является консистентным: например, в последовательных виртуальных вызовах были выбраны реализации методов из разных классов-наследников.
- `MACRO` используется для предупреждений, в которых инструкция из стока была получена путём раскрытия макроса (директивы `define`). Сам по себе этот факт ничего не говорит об истинности или ложности данного срабатывания, но поскольку во всех местах использования макросы порождают схожие инструкции, то и рассматривать предупреждения с ними имеет смысл вместе.
- `MALLOC` используется для предупреждений, в которых сток соответствует вызову `malloc` или `new`, поскольку не для всех программ попытку выделения произвольно большой области памяти можно считать уязвимостью.
- `OVERFLOW` используется для предупреждений, где помимо использования помеченной переменной в качестве индекса при обращении к буферу или при выделении памяти, возможно целочисленное перепол-

нение этой переменной, что повышает вероятность того, что данное предупреждение может оказаться настоящей уязвимостью.

– `OVERTAINT` используется для предупреждений, которые скорее всего являются ложными из-за того, что анализатор пометил слишком большую область памяти. Формальным критерием для этого тега является то, что в пути распространения помеченности есть факты, помеченность в которых распространяется сразу на несколько полей структуры или класса: хотя такое и возможно в реальных программах, в большинстве случаев наличие таких фактов будет свидетельствовать о недостаточной точности анализатора при работе с моделью памяти.
– `VARARG` используется для того чтобы отметить, что помеченные данные были переданы через функцию с переменным числом аргументов и возможны неточности, как правило избыточная помеченность.

5.3 Особенности реализации

Чтобы инструмент был полезен на практике, он должен обладать удовлетворительной скоростью работы и низким процентом ложных срабатываний. Чаще всего приходится искать баланс между масштабируемостью, точностью и полнотой, поскольку улучшение одного показателя может приводить к ухудшению двух других.

Опишем ключевые отличия от схемы работы анализатора, предложенной в работе [16].

Первым отличием от изначальной схемы являются сами помеченные факты. Поскольку C и C++ являются более низкоуровневыми языками, чем Java, то в них не всегда можно однозначно установить тип данных, соответствующий некоторой области памяти или переменной. По этой причине вместо описания понятия «путь доступа» через базовое значение или объект и последовательность выбора полей в нём, в работе [86] был предложен альтернативный вариант с базовым значением и последовательностью смещений в байтах (константных, либо переменных, обозначаемых специальным символом «@») и разыменований относительно него. Такое представление намного ближе к особенностям данных языков и внутреннему представлению LLVM и по-прежнему позволяет указы-

вать на конкретные поля структур и классов через их смещения, причём при необходимости их названия могут быть восстановлены за счёт использования метаданных программы.

Примеры различных способов отображения одного и того же пути доступа:

- *[%12, 0, 10, 0]* — значение под номером 12 из внутреннего представления LLVM, относительно которого было взято два разыменования: с нулевым смещением и со смещением в 10 байт. Последний 0 соответствует «текущему смещению»;
- **(buf + 10)* — отображаемый путь доступа, если установить факт, что значение под номером 12 соответствовало адресу переменной buf;
- **(buf.data)* — отображаемый путь доступа, если установить факт, что buf имеет тип структуры, в которой смещение в 10 байт соответствует полю под названием data.

Следует отметить, что количество возможных значений целочисленных смещений очевидно больше количества различных полей в объектах, фигурировавших в изначальной формулировке. Это приводит к значительному увеличению количества помеченных фактов, которые возможно задать (D) и на котором базируется оценка сложности алгоритма решения задачи IFDS.

По этой причине практическое применение данного алгоритма на больших проектах невозможно без применения некоторых отсечений, которые ограничивают множество рассматриваемых путей в расширенном суперграфе данной задачи. Типичными ограничениями являются максимальное количество разыменований в пути доступа, максимальная расстояние от истока помеченности, количество итераций в цикле или рекурсивных вызовов или же просто ограничение по максимальному количеству рассматриваемых рёбер суперграфа.

Помимо времени решения, существует и проблема памяти. Объём используемой при решении задачи IFDS памяти напрямую зависит от времени анализа, поскольку алгоритм требует хранение всего расширенного суперграфа потоков управления программы, который достраивается на каждой итерации алгоритма. Как следствие, решения на основе IFDS исчерпывают всю доступную память при попытке анализа достаточно больших проектов, содержащих помеченные данные.

Для борьбы с этой проблемой в данной работе применяется разбиение анализа на подзадачи: на вход IFDS решателю подаётся только 1 исток по-

меченности за раз, каждый запрос к анализу псевдонимов порождает новую задачу IFDS (либо использует ранее вычисленный и сохранённый результат) и т.д. Такой подход увеличивает время анализа, поскольку одни и те же участки графа могут пересчитываться несколько раз в различных подзадачах, однако позволяет ограничить потребление памяти.

По схожей причине в данной работе не используется понятие нулевого факта, выполняемого во всех точках программы — вместо этого выполнение IFDS решателя начинается непосредственно с истока помеченности.

5.3.1 Анализ псевдонимов

Наличие анализа псевдонимов — важное условие для реализации статического анализа, особенно в случае относительно низкоуровневого языка C, где манипулирование указателями осуществляется вручную.

Реализация анализа псевдонимов основана на концепции, предложенной и реализованной в инструменте Flowdroid [16]. В рамках этой концепции анализ псевдонимов осуществляется лениво (по запросу) и только для переменных, содержащих помеченные значения.

Этот анализ выглядит как два связанных IFDS решателя, распространяющих факты в противоположных направлениях. Помеченный факт в них задаётся как путь доступа и вершина активации — точка в программе, для которой нужно осуществить поиск псевдонимов. На первом шаге обратному решателю передаётся факт, состоящий из пути доступа, для которого требуется найти псевдонимы, и точки в программе, из которой поступил запрос. Этот факт распространяется в обратном направлении, при этом вершина активации остаётся неизменной, а путь доступа может изменяться, отображая изменения в значениях указателей, через которые можно получить доступ к данной области памяти. При обнаружении факта копирования указателей, связанных с текущим путём доступа, новый факт передаётся прямому IFDS решателю, который совершает его распространение аналогично обычному IFDS анализу (в том числе он может добавлять новые факты к обратному IFDS решателю) — если такой факт достигает вершины активации, то содержащийся в нём путь доступа считается псевдонимом для фигурировавшего в запросе в данной точке программы.

В рамках данной работы в эту схему было внесено несколько изменений:

1. Как и в случае с разбиением на подзадачи основного анализа, каждое обращение к анализу псевдонимов оформляется в виде отдельной задачи с двумя IFDS решателями (которые в свою очередь могут порождать новые запросы). Результаты запросов сохраняются и могут быть переиспользованы без повторного запуска решателя.
2. Чтобы дополнительно ограничить потребление памяти, в процессе решения задачи IFDS не сохраняются рёбра расширенного суперграфа — только его посещённые вершины. Это возможно потому, что для анализа псевдонимов не требуется восстанавливать трассу, по которой проходил помеченный факт.
3. IFDS решатели, используемые в анализе псевдонимов, осуществляют только внутрипроцедурное распространение.

Одним из ключевых изменений стала разработка алгоритма отсечения тупиковых ветвей в прямом анализе псевдонимов. Рассмотрим рисунок 5.2.

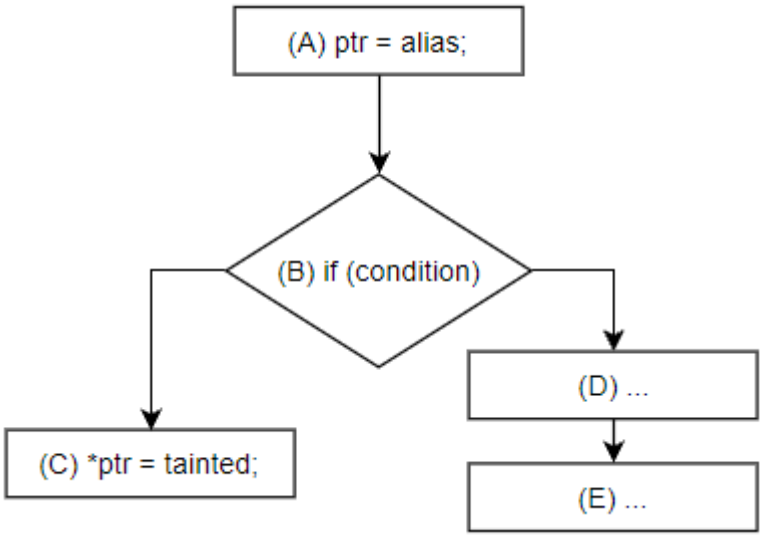

Рисунок 5.2 — Иллюстрация проблемы тупиковых ветвей в анализе псевдонимов

В вершине *(C)* вызывается анализ псевдонимов для факта $<*ptr, C>$, чтобы узнать, какие ещё пути доступа оказались помечены присваиванием переменной *tainted*. Обратным распространением этот факт достигает вершины *(A)*, из которой запускается прямой анализ с новым фактом $<*alias, C>$. После достижения этим фактом вершины *(C)*, путь доступа **alias* объявляется псевдонимом **ptr* в данной точке.

Проблема тупиковых ветвей в прямом анализе заключается в том, что факты прямого анализа распространяются в том числе и в те ветви программы, из которых невозможно достичь вершины активации. Анализ таких ветвей не даёт никаких результатов ввиду уточнения о внутрипроцедурности проводимого анализа, при этом за (D) может располагаться значительное количество других вершин, в том числе и порождающих новые запросы к анализу псевдонимов.

Алгоритм отсечения тупиковых ветвей выглядит следующим образом:

1. При каждом запросе к анализу псевдонимов, для вершины активации вводится два множества: допустимые вершины и исключённые. Допустимыми называются те вершины в функции, из которых существует путь в вершину активации (на рисунке 5.2 это вершины A, B и C). Исключёнными называются те вершины, в которые можно напрямую попасть из допустимой вершины, но из которых не существует пути в вершину активации (на рисунке 5.2 это вершина D).
2. Множество допустимых вершин строится путём обхода в ширину в обратном направлении, начиная от вершины активации.
3. После завершения предыдущего шага, для каждого элемента этого множества, кроме вершины активации, проверяются все исходящие рёбра. Если ребро ведёт в вершину, не входящую в множество допустимых, то она добавляется в множество исключённых.
4. Вычисленное таким способом множество исключённых вершин сохраняется для повторного использования и передаётся прямому IFDS решателю — при посещении вершин из этого множества распространение помеченного факта должно досрочно прерываться.

Следует отметить, что построенное таким образом множество включает включает в себя только те вершины, в которые действительно можно попасть в ходе анализа псевдонимов. В частности, оно включает в себя вершину *(D)*, но не включает *(E)* и следующие за ней — это помогает экономить память, расходуемую на хранение исключённых вершин для каждой вершины активации.

5.4 Оценка результатов работы

Для тестирования результатов работы реализованных в инструменте Irbis алгоритмов использовались несколько проектов.

Первым проектом является подмножество тестового набора “Juliet 1.3 test suite for C/C++”, включающее в себя тесты на потенциальные уязвимости из следующих классов:

- CWE-124 – Buffer Underwrite
- CWE-126 – Buffer Overread
- CWE-127 – Buffer Underread
- CWE-134 – Uncontrolled Format String
- CWE-194 – Unexpected Sign Extension
- CWE-195 – Signed to Unsigned Conversion Error
- CWE-400 – Resource Exhaustion
- CWE-680 – Integer Overflow to Buffer Overflow
- CWE-789 – Uncontrolled Mem Alloc

Из подмножества были исключены тесты, содержащие в названии слова “w32” или “wchar”, поскольку они предназначены для сборки в ОС Windows.

Среди оставшихся были отобраны 4656 тестов, в заголовках которых поле “BadSource” содержится слово “read”. Остальные 3830 тестов не представляют интереса для анализа помеченных данных, поскольку вместо пользовательских данных для выхода за пределы буфера используются константные некорректные индексы. Для поиска подобных ошибок следует использовать средства динамического анализа, фаззинга, символьного выполнения или формальной верификации.

Каждый тест включает в себя один содержащий уязвимость тестовый пример и один или более тестовых примеров, на которых уязвимость отсутствует или не осуществима. Чтобы облегчить классификацию срабатываний на ложные и истинные, тестовый набор поддерживает два макроса препроцессора: OMIT_GOOD и OMIT_BAD, которые, будучи объявлены во время сборки, убирают тестовые примеры соответствующей группы.

В этом случае все срабатывания на тестах, собранных с макросом OMIT_GOOD, можно считать истинными, а все срабатывания на тестах, собранных с макросом OMIT_BAD, можно считать ложными. В этом случае

время анализа считается как сумма времени двух запусков, а потребление памяти берётся по максимальному показателю среди запусков.

Вторым тестовым проектом является OpenSSL версии 1.0.1f, содержащий уязвимость Heartbleed [87; 88], которую должен обнаруживать инструмент. Помимо этого запускался анализ на проектах LibTIFF и BinUtils.

Тестирование производилось на виртуальной машине под управлением Ubuntu 22.04, которой предоставлено 16 Гб оперативной памяти и 4 процессора. Оценка общей точности и масштабируемости приведена в Таблице 8. В случае Juliet Test Suite и LibTiff были размечены все выданные предупреждения, в случае OpenSSL и BinUtils — не менее 50 случайно выбранных предупреждений. В столбце «всего предупреждений» указаны два числа — первое это количество предупреждений всех типов, а в скобках — за вычетом `TAINTED_PTR.LOAD` и `TAINTED_PTR.STORE`, в которых стоками являются не вызовы функций, а инструкции чтения и записи данных по помеченному адресу.

Истинность предупреждения не является гарантией наличия в данном месте уязвимости, которой может воспользоваться злоумышленник — проверялось только наличие зависимости по данным между указанным истоком и стоком помеченности, что может свидетельствовать о наличии потенциальной уязвимости, а также отсутствие в трассе очевидных для человека проверок, которые бы указывали на ложность найденного предупреждения.

Таблица 8 — Оценка точности и масштабируемости анализа

Проект	Всего пред-ний	Истинных	Строк кода	Память	Время анализа
Juliet	4656	100%	855 тыс.	1.8 Гб	27м
LibTIFF	36 (28)	56%	69 тыс.	1.9 Гб	4м
OpenSSL	822 (127)	64%	278 тыс.	2.9 Гб	1ч 13м
BinUtils	487 (288)	66%	3.27 млн.	9.6 Гб	5ч 9м

На примере LibTiff, все срабатывания анализатора соответствовали реально существующим зависимостям по данным между истоком и стоком помеченных данных. Основные причины, по которым часть из этих срабатываний были размечены как ложные:

1. Значение помеченной целочисленной переменной было согласовано с размером буфера, в который осуществлялась запись в стоке помеченности, из-за чего фактически уязвимость отсутствует, т.к. выхода за

границы буфера нет. Несмотря на то, что в анализаторе существует эвристика для фильтрации таких срабатываний, в текущей реализации она работает только внутрипроцедурно, в области стока. Чтобы отсеять такие ложные срабатывания, можно расширить данную эвристику, используя метод, описанный в разделе 2.1.2.

2. Значение помеченной целочисленной переменной было ограничено сверху, но не константой, а значением другой переменной или выражения. Хотя с точки зрения человека этого бывает достаточно, чтобы говорить о снятии помеченности — у анализатора на основе IFDS недостаточно информации о возможных значениях сравниваемых переменных. Такие срабатывания можно было бы уточнить при помощи двухэтапного анализа с использованием статического символьного выполнения, по аналогии с 2.1.1.
3. Недостаточно точное представление помеченных строк в анализаторе, из-за чего отсутствует информация о их возможной длине, которая определяется положением нулевого байта относительно начала строки.

Анализатор намеренно настроен так, чтобы в подобных спорных ситуациях выдавать предупреждение, т.к. в данной работе был отдан приоритет полноте анализа, а значит предпочтительнее выдать ложное срабатывание, чем потерять истинное, которое может соответствовать уязвимости.

На OpenSSL абсолютное большинство (695 из 822) срабатываний пришлось на два детектора, в которых стоками являются не вызовы функций, а инструкции чтения или записи данных по помеченному адресу — в первую очередь это вызвано большим количеством криптографических функций в проекте. Из-за сложности в понимании и проверке такого кода, срабатывания этих детекторов на данном проекте не размечались и не учитывались в проценте истинных и ложных срабатываний. Основной целью анализа данного проекта было обнаружение уязвимости Hearbleed, что было успешно достигнуто: инструмент продемонстрировал трассу распространения помеченных данных от вызова `BIO_read()` внутри функции `ssl3_read_n` в файле `s3_pkt.c` до вызова `memcpy()` внутри функции `dtls1_process_heartbeat` в файле `d1_both.c`.

Также было проведено сравнение на тестах, входящих в Juliet test suite, с реализациями статического анализа помеченных данных в трёх инструментах: Svace [62], Clang Static Analyzer (CSA) [89] и Infer Static Analyzer [90]. Сравнение проводилось по всем тестам с чтением данных из внешнего источника

и относящимся к одной из перечисленных категорий потенциальных уязвимостей: CWE-124, CWE-126, CWE-127, CWE-134, CWE-194, CWE-195, CWE-400, CWE-680 и CWE-789. Учитывались только предупреждения детекторов, реализующих анализ помеченных данных. Результаты приведены в Таблице 9.

Таблица 9 — Сравнение с анализом помеченных данных в других инструментах

Анализатор	TP	FN	FP	RAM
Irbis	4656	0	0	3 Гб
Svace 3.2	3666	990	248	4 Гб
CSA	1197	3459	17	<1 Гб
Infer	753	3903	437	<1 Гб

Лучшие результаты по полноте анализа продемонстрировали инструменты Irbis и Svace, причём только Irbis покрыл все выбранные тесты и не выдал ни одного ложного предупреждения. Инструменты CSA и Infer оказались наиболее легковесными, поскольку потребовали меньше 1 гигабайта памяти, а сам анализ происходил в процессе сборки анализируемого проекта, из-за чего измерить время, затраченное на собственно анализ, оказалось невозможным. Irbis и Svace продемонстрировали схожие требования по памяти и времени анализа.

В данной главе было приведена общая схема работы инструмента Irbis, реализующего предложенные в данной работе методы и алгоритмы. Указано его взаимодействие с инфраструктурой Svace, перечислены реализованные типы детекторов, используемые теги для предупреждений и особенности реализации.

Также в данной главе приведены результаты оценки результатов общей работы анализатора на различных проектах с открытым исходным кодом и сравнение с несколькими другими анализаторами помеченных данных на тестовом наборе Juliet Test Suite, которые показали высокую полноту и масштабируемость подхода, а также возможность обнаруживать такие уязвимости как Hearbleed. Разработанные в данной работе методы позволили отсеять все ложные срабатывания инструмента на тестовом наборе, и показать точность более 50% на реальных проектах. Это ниже точности, характерной для индустриальных статических анализаторов, предназначенных для поиска ошибок в программах, но в сочетании с другими факторами это делает данную совокупность методов подходящей как для поиска максимального числа потенциальных

уязвимостей с ручной проверкой результатов, так и для использования на первом этапе двухэтапного анализа, в котором точность повышается за счёт использования инструментов на основе символьного выполнения.

Заключение

Основные результаты работы заключаются в следующем.

1. Разработаны алгоритмы и эвристики, повышающие полноту анализа помеченных данных: в частности, анализ косвенных вызовов на основе решения задачи IFDS и эвристики для автоматического обнаружения новых истоков помеченности определённых видов. Анализ косвенных вызовов позволил найти правильных кандидатов для всех косвенных вызовов на Juliet Test Suite и до 30% от теоретически возможного на OpenSSL, без ложных срабатываний. Эвристики для обнаружения новых истоков на OpenSSL обнаружили 15 функций типа «чтение данных» и 175 функций типа «освобождение памяти» соответственно, при этом до 90% из найденных истоков могут быть использованы в автоматическом режиме, без уточнения человеком.
2. Разработаны алгоритмы повышения точности анализа помеченных данных: двухэтапный анализ, позволяющий добавить проверку консистентности путей, снятие помеченности с целочисленных переменных, которые позволили устранить все ложные срабатывания на тестовом наборе Juliet Test Suite, не потеряв истинные. На реальных проектах они позволили на четверть сократить количество ложных срабатываний.
3. Разработан алгоритм, повышающий масштабируемость анализа помеченных данных: направленное распространение глобальных переменных. На тестовом наборе Juliet Test Suite он продемонстрировал двухкратное ускорение анализа, на OpenSSL он позволил найти больше срабатываний за счёт того, что меньше времени уходило на анализ глобальных переменных.
4. На основе предложенных методов и алгоритмов автором был разработан инструмент Irbis, реализованный на базе открытой компиляторной инфраструктуры LLVM. Инструмент был испытан на реальных программных проектах и тестовом наборе Juliet Test Suite с искусственно созданными уязвимостями с целью оценить полноту, точность и масштабируемость анализа. На Juliet Test Suite инструмент прошёл 4258 тестов на поддерживаемые анализатором типы уязвимостей, в которых

есть исток помеченных данных. При использовании всех описываемых в работе алгоритмов, как ложные, так и пропущенные срабатывания отсутствуют. На OpenSSL инструмент способен обнаружить уязвимость Heartbleed, показывая межпроцедурный путь распространения помеченных данных от считывания данных из сети до использования переменной `payload` в качестве размера копируемого буфера.

Публикации автора по теме диссертации

1. Сравнительный анализ двух подходов к статическому анализу помеченных данных. / М. Беляев, Н. Шимчик, В. Игнатьев, А. Белеванцев // Труды Института системного программирования РАН. — 2017. — Т. 29, № 3. — С. 99—116.
2. Comparative analysis of two approaches to static taint analysis / M. Belyaev, N. Shimchik, V. Ignatyev, A. Belevantsev // Programming and Computer Software. — 2018. — Т. 44, № 6. — С. 459—466.
3. *Шимчик, Н.* Поиск уязвимостей при помощи статического анализа помеченных данных. / Н. Шимчик, В. Игнатьев // Труды Института системного программирования РАН. — 2019. — Т. 31, № 3. — С. 177—190.
4. *Шимчик, Н.* Irbis: статический анализатор помеченных данных для поиска уязвимостей в программах на C/C++ / Н. Шимчик, В. Игнатьев, А. Белеванцев // Труды Института системного программирования РАН. — 2023. — Т. 34, № 6. — С. 51—66.
5. *Shimchik, N.* Improving Accuracy and Completeness of Source Code Static Taint Analysis / N. Shimchik, V. Ignatyev, A. Belevantsev // 2021 Ivannikov Ispras Open Conference (ISPRAS). — IEEE. 2021. — С. 61—68.
6. *Шимчик, Н.* Поиск критических ошибок с помощью межпроцедурного анализа помеченных данных в программах на Си/Си++ / Н. Шимчик, С. Гайсарян // Ломоносовские чтения 2018 ф-т ВМК МГУ. — 2018.
7. *Chibisov, D.* Improving Scalability of Inter-module Source Code Static Taint Analysis / D. Chibisov, N. Shimchik, V. Ignatyev // 2023 Ivannikov Ispras Open Conference (ISPRAS). — IEEE. 2023.
8. *Корябкин, Д.* Повышение точности анализа помеченных данных с помощью символьных вычислений / Д. Корябкин, В. Игнатьев, Н. Шимчик // Ломоносовские чтения-2020. Секция «Вычислительной математики и кибернетики». — 2020.
9. *Свидетельство о гос. регистрации программы для ЭВМ № 2019661044 от 16.08.2019.* Подсистема валидации предупреждений об ошибках, сгенерированных анализатором помеченных данных, с помощью методов

символьного выполнения / Н. Шимчик, Д. Корябкин, В. Игнатьев, М. Беляев. — (Рос. Федерация).

10. *Свидетельство о гос. регистрации программы для ЭВМ № 2019660638 от 09.08.2019.* Инфраструктура статического анализа помеченных данных для программ на языках Си и C++ / Н. Шимчик, Д. Корябкин, В. Игнатьев, М. Беляев. — (Рос. Федерация).
11. *Свидетельство о гос. регистрации программы для ЭВМ № 2017660157 от 18.09.2017.* Инфраструктура анализа помеченных данных инструмента статического анализа «SharpChecker» / В. Игнатьев, В. Кошелев, А. Борзилов, А. Белеванцев, Н. Шимчик, М. Беляев. — (Рос. Федерация).
12. *Свидетельство о гос. регистрации программы для ЭВМ № 2017660156 от 18.09.2017.* Детектор недостижимого кода в программах на языке C# инструмента статического анализа «SharpChecker» / В. Игнатьев, В. Кошелев, А. Борзилов, А. Белеванцев, Н. Шимчик, М. Беляев. — (Рос. Федерация).
13. *Свидетельство о гос. регистрации программы для ЭВМ № 2017660039 от 13.09.2017.* Расширение Microsoft Visual Studio 2015 для интеграции с инструментом статического анализа «SharpChecker» / В. Игнатьев, В. Кошелев, А. Борзилов, А. Белеванцев, Н. Шимчик, М. Беляев. — (Рос. Федерация).

Список литературы

14. *Marinescu, P. D.* KATCH: High-Coverage Testing of Software Patches / P. D. Marinescu, C. Cadar // Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. — Saint Petersburg, Russia : Association for Computing Machinery, 2013. — С. 235—245. — (ESEC/FSE 2013). — URL: https://doi.org/10.1145/2491411.2491438.

15. *Reps, T.* Interprocedural dataflow analysis via graph reachability / T. Reps, M. Sagiv, S. Horwitz. — Citeseer, 1994.

16. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps / S. Arzt [и др.] // Acm Sigplan Notices. — 2014. — Т. 49, № 6. — С. 259—269.

17. *Nethercote, N.* Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation / N. Nethercote, J. Seward // SIGPLAN Not. — New York, NY, USA, 2007. — Июнь. — Т. 42, № 6. — С. 89—100. — URL: https://doi.org/10.1145/1273442.1250746.

18. *Cadar, C.* KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs / C. Cadar, D. Dunbar, D. Engler // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — San Diego, California : USENIX Association, 2008. — С. 209—224. — (OSDI'08).

19. *D'Silva, V.* A Survey of Automated Techniques for Formal Software Verification / V. D'Silva, D. Kroening, G. Weissenbacher // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. — 2008. — Т. 27, № 7. — С. 1165—1178.

20. *Parvez, R.* Combining Static Analysis and Targeted Symbolic Execution for Scalable Bug-Finding in Application Binaries / R. Parvez, P. A. S. Ward, V. Ganesh // Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering. — Toronto, Ontario, Canada : IBM Corp., 2016. — С. 116—127. — (CASCON '16).

21. *Sutton, M.* Fuzzing: Brute Force Vulnerability Discovery / M. Sutton, A. Greene, P. Amini. — Addison-Wesley Professional, 2007.

22. *Oehlert, P.* Violating assumptions with fuzzing / P. Oehlert // IEEE Security Privacy. — 2005. — Т. 3, № 2. — С. 58—62.

23. *Godefroid, P.* SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. / P. Godefroid, M. Y. Levin, D. Molnar // Queue. — New York, NY, USA, 2012. — Янв. — Т. 10, № 1. — С. 20—27. — URL: https://doi.org/10.1145/2090147.2094081.

24. american fuzzy lop [Электронный ресурс]. — URL: https://lcamtuf.coredump.cx/afl/.

25. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking / H. Post [и др.] // 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. — 2008. — С. 188—197.

26. SMT-Based False Positive Elimination in Static Program Analysis / M. Junker [и др.] // Formal Methods and Software Engineering / под ред. T. Aoki, K. Taguchi. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. — С. 316—331.

27. *Brat, G.* Combining static analysis and model checking for software analysis / G. Brat, W. Visser // Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). — 2001. — С. 262—269.

28. *Fehnker, A.* Model checking driven static analysis for the real world: Designing and tuning large scale bug detection / A. Fehnker, R. Huuck // Innovations in Systems and Software Engineering. — 2013. — Март. — Т. 9.

29. *Chipounov, V.* S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems / V. Chipounov, V. Kuznetsov, G. Candea // SIGPLAN Not. — New York, NY, USA, 2011. — Март. — Т. 46, № 3. — С. 265—278. — URL: https://doi.org/10.1145/1961296.1950396.

30. Unleashing Mayhem on Binary Code / S. K. Cha [и др.] // 2012 IEEE Symposium on Security and Privacy. — 2012. — С. 380—394.

31. *Trinh, M.-T.* S3: A Symbolic String Solver for Vulnerability Detection in Web Applications / M.-T. Trinh, D.-H. Chu, J. Jaffar // Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. — Scottsdale, Arizona, USA : Association for Computing Machinery, 2014. — С. 1232—1243. — (CCS '14). — URL: https://doi.org/10.1145/2660267.2660372.

32. *Bounimova, E.* Billions and billions of constraints: Whitebox fuzz testing in production / E. Bounimova, P. Godefroid, D. Molnar // 2013 35th International Conference on Software Engineering (ICSE). — 2013. — С. 122—131.

33. *Babic, D.* Calysto: Scalable and Precise Extended Static Checking / D. Babic, A. J. Hu // Proceedings of the 30th International Conference on Software Engineering. — Leipzig, Germany : Association for Computing Machinery, 2008. — С. 211—220. — (ICSE '08). — URL: https://doi.org/10.1145/1368088.1368118.

34. *Dillig, I.* Sound, Complete and Scalable Path-Sensitive Analysis / I. Dillig, T. Dillig, A. Aiken // SIGPLAN Not. — New York, NY, USA, 2008. — Июнь. — Т. 43, № 6. — С. 270—280. — URL: https://doi.org/10.1145/1379022.1375615.

35. Статический анализатор Svace для поиска дефектов в исходном коде программ / В. Иванников [и др.] // Труды Института системного программирования РАН. — 2014. — Т. 26, № 1.

36. *Бородин, А.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности / А. Бородин, А. Белеванцев // Труды Института системного программирования РАН. — 2015. — Т. 27, № 6.

37. Design and Development of Svace Static Analyzers / A. Belevantsev [и др.] // 2018 Ivannikov Memorial Workshop (IVMEM). — IEEE. 2018. — С. 3—9.

38. *Sen, K.* CUTE: A Concolic Unit Testing Engine for C / K. Sen, D. Marinov, G. Agha // SIGSOFT Softw. Eng. Notes. — New York, NY, USA, 2005. — Сент. — Т. 30, № 5. — С. 263—272. — URL: https://doi.org/10.1145/1095430.1081750.

39. *Sen, K.* DART: Directed Automated Random Testing / K. Sen // Hardware and Software: Verification and Testing / под ред. K. Namjoshi, A. Zeller, A. Ziv. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — С. 4—4.

40. Dowser: a guided fuzzer to find buffer overflow vulnerabilities / I. Haller [и др.] // Proceedings of the 22nd USENIX Security Symposium. — 2013. — С. 49—64.

41. Hercules: Reproducing crashes in real-world application binaries / V.-T. Pham [и др.] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Т. 1. — IEEE. 2015. — С. 891—901.

42. *Zamfir, C.* Execution synthesis: a technique for automated software debugging / C. Zamfir, G. Candea // Proceedings of the 5th European conference on Computer systems. — 2010. — С. 321—334.

43. Practical fault localization for dynamic web applications / S. Artzi [и др.] // 2010 ACM/IEEE 32nd International Conference on Software Engineering. Т. 1. — IEEE. 2010. — С. 265—274.

44. *Kildall, G. A.* A unified approach to global program optimization / G. A. Kildall // Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — 1973. — С. 194—206.

45. *Reps, T.* Precise Interprocedural Dataflow Analysis via Graph Reachability / T. Reps, S. Horwitz, M. Sagiv // Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — San Francisco, California, USA : Association for Computing Machinery, 1995. — С. 49—61. — (POPL '95). — URL: https://doi.org/10.1145/199448.199462.

46. *Newsome, J.* Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. / J. Newsome, D. X. Song // NDSS. Т. 5. — Citeseer. 2005. — С. 3—4.

47. *Clause, J.* Dytan: a generic dynamic taint analysis framework / J. Clause, W. Li, A. Orso // Proceedings of the 2007 international symposium on Software testing and analysis. — 2007. — С. 196—206.

48. libdft: Practical dynamic data flow tracking for commodity systems / V. P. Kemerlis [и др.] // Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments. — 2012. — С. 121—132.

49. *Kemerlis, V. P.* libdft: Dynamic data flow tracking for the masses / V. P. Kemerlis // Rn. — 2022. — Т. 1. — R2.

50. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform / A. Henderson [и др.] // Proceedings of the 2014 International Symposium on Software Testing and Analysis. — 2014. — С. 248—258.

51. ShadowReplica: efficient parallelization of dynamic data flow tracking / K. Jee [и др.] // Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. — 2013. — С. 235—246.

52. Parallelizing security checks on commodity hardware / E. B. Nightingale [и др.] // ACM SIGARCH Computer Architecture News. — 2008. — Т. 36, № 1. — С. 308—318.

53. Parallelizing dynamic information flow tracking / O. Ruwase [и др.] // Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures. — 2008. — С. 35—45.

54. A practical off-line taint analysis framework and its application in reverse engineering of file format / B. Cui [и др.] // Computers & Security. — 2015. — Т. 51. — С. 1—15.

55. SWIFT: Decoupled System-Wide Information Flow Tracking and its Optimizations. / C.-W. Wang, S. W. Shieh [и др.] // J. Inf. Sci. Eng. — 2015. — Т. 31, № 4. — С. 1413—1429.

56. *Whelan, R.* Architecture-independent dynamic information flow tracking / R. Whelan, T. Leek, D. Kaeli // International Conference on Compiler Construction. — Springer. 2013. — С. 144—163.

57. *Yadegari, B.* Bit-level taint analysis / B. Yadegari, S. Debray // 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. — IEEE. 2014. — С. 255—264.

58. *Rawat, S.* Static taint-analysis on binary executables / S. Rawat, L. Mounier, M.-L. Potet. — 2011.

59. Still: Exploit code detection via static taint and initialization analyses / X. Wang [и др.] // 2008 Annual Computer Security Applications Conference (ACSAC). — IEEE. 2008. — С. 289—298.

60. Android taint flow analysis for app sets / W. Klieber [и др.] // Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. — 2014. — С. 1—6.

61. *Bodden, E.* Inter-procedural data-flow analysis with ifds/ide and soot / E. Bodden // Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. — 2012. — С. 3—8.

62. Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace / А. Бородин [и др.] // Труды Института системного программирования РАН. — 2021. — Т. 33, № 1. — С. 7—32.

63. *Naeem, N.* Practical Extensions to the IFDS Algorithm / N. Naeem, O. Lhoták, J. Rodriguez //. — 03.2010. — С. 124—144.

64. *Schubert, P. D.* PhASAR: An Inter-procedural Static Analysis Framework for C/C++ / P. D. Schubert, B. Hermann, E. Bodden // Tools and Algorithms for the Construction and Analysis of Systems / под ред. T. Vojnar, L. Zhang. — Cham : Springer International Publishing, 2019. — С. 393—410.

65. *Lattner, C.* LLVM: A compilation framework for lifelong program analysis & transformation / C. Lattner, V. Adve // International Symposium on Code Generation and Optimization, 2004. CGO 2004. — IEEE. 2004. — С. 75—86.

66. travich/whole-program-llvm: A wrapper script to build whole-program LLVM bitcode files [Электронный ресурс]. — URL: https://github.com/travitch/whole-program-llvm.

67. Chex: statically vetting android apps for component hijacking vulnerabilities / L. Lu [и др.] // Proceedings of the 2012 ACM conference on Computer and communications security. — 2012. — С. 229—240.

68. *Yang, Z.* Leakminer: Detect information leakage on android with static taint analysis / Z. Yang, M. Yang // 2012 Third World Congress on Software Engineering. — IEEE. 2012. — С. 101—104.

69. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale / C. Gibler [и др.] // International Conference on Trust and Trustworthy Computing. — Springer. 2012. — С. 291—307.

70. *Fuchs, A. P.* Scandroid: Automated security certification of android applications / A. P. Fuchs, A. Chaudhuri, J. S. Foster // Manuscript, Univ. of Maryland, http://www.cs.umd.edu/avik/projects/scandroidascaa. — 2009. — Т. 2, № 3.

71. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis / D. he [et al.] //. — 11/2019. — P. 267—279.

72. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis / D. He [и др.] // 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). — IEEE. 2019. — С. 267—279.

73. *Zaher, A. K.* Parameterized Algorithms for Scalable Interprocedural Data-flow Analysis / A. K. Zaher // arXiv preprint arXiv:2309.11298. — 2023.

74. *Karakaya, K.* Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems / K. Karakaya, E. Bodden // arXiv preprint arXiv:2401.14813. — 2024.

75. *Hsu, M.-Y.* Efficient Program Analyses That Scale to Large Codebases : дис. ... канд. / Hsu Min-Yih. — University of California, Irvine, 2023.

76. Combining Static and Dynamic Analysis to Discover Software Vulnerabilities / R. Zhang [и др.] // 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. — 2011. — С. 175—181.

77. Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed / B. Kiss [и др.] // Hardware and Software: Verification and Testing / под ред. N. Piterman. — Cham : Springer International Publishing, 2015. — С. 39—50.

78. Combining dynamic symbolic execution, code static analysis and fuzzing / A. Y. Gerasimov [и др.] // Труды Института системного программирования РАН. — 2018. — Т. 30, № 6.

79. Software Assurance Reference Datasheet [Электронный ресурс]. — URL: https://samate.nist.gov/SRD/testsuite.php.

80. *Lengauer, T.* A fast algorithm for finding dominators in a flowgraph / T. Lengauer, R. E. Tarjan // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1979. — Т. 1, № 1. — С. 121—141.

81. *Bacon, D. F.* Fast static analysis of C++ virtual function calls / D. F. Bacon, P. F. Sweeney // Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. — 1996. — С. 324—341.

82. *Carini, P. R.* Flow-sensitive type analysis for C++ / P. R. Carini, M. Hind, H. Srinivasan. — Citeseer, 1995.

83. *Lu, K.* Where does it go? Refining indirect-call targets with multi-layer type analysis / K. Lu, H. Hu // Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. — 2019. — С. 1867—1881.

84. JSON [Электронный ресурс]. — URL: https://www.json.org/json-ru.html.

85. Mining Apps for Abnormal Usage of Sensitive Data / V. Avdiienko [et al.] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. — 2015. — P. 426—436.

86. *Кошелев, В.* Межпроцедурный анализ помеченных данных на базе инфраструктуры LLVM / В. Кошелев, А. Избышев, И. Дудина // Труды Института системного программирования РАН. — 2014. — Т. 26, № 2.

87. Heartbleed Bug [Электронный ресурс]. — URL: https://heartbleed.com/.

88. CVE - CVE-2014-0160 [Электронный ресурс]. — URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

89. *Arroyo, M.* An user configurable clang static analyzer taint checker / M. Arroyo, F. Chiotta, F. Bavera // 2016 35th International Conference of the Chilean Computer Science Society (SCCC). — IEEE. 2016. — С. 1—12.

90. *Calcagno, C.* Infer: An automatic program verifier for memory safety of C programs / C. Calcagno, D. Distefano // NASA Formal Methods Symposium. — Springer. 2011. — С. 459—465.