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»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2024

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

Научный руководитель: кандидат физико-математических наук
Игнатьев Валерий Николаевич

Официальные оппоненты: **Шабанов Борис Михайлович,**
член-корреспондент РАН, доктор технических наук, доцент,
директор Межведомственного Суперкомпьютерного Центра РАН,
заместитель директора по научной работе Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Маркин Дмитрий Олегович,
кандидат технических наук,
сотрудник ФГКВОУ ВО «Академия Федеральной службы охраны Российской Федерации»

Ведущая организация: Федеральное государственное учреждение «Федеральный исследовательский центр Институт прикладной математики им. М.В. Келдыша Российской академии наук»

Защита состоится «___» июня 2024 г. в ___ часов на заседании диссертационного совета Д 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В. П. Иванникова Российской академии наук по адресу: 109004, г. Москва, ул. А. Солженицына, дом 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования им. В. П. Иванникова РАН.

Автореферат разослан «___» _____ 2024 года.

Ученый секретарь
диссертационного совета
Д 24.1.120.01,
кандидат
физико-математических наук Зеленов С. В.

Общая характеристика работы

Актуальность темы. 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 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.

[1] CWE-120: Buffer Copy without Checking Size of Input
[2] CVE-2014-0160: Heartbleed

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, 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 задача IFDS[3], 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

[3]Interprocedural Finite Distributive Subset problem

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 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.

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 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, 2 — в периодических научных журналах, индексируемых Web of Science и Scopus, 2 — 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 объем диссертации составляет **108** страниц текста с **5** рисунками, **9** таблицами и **7** листингами кода. Список литературы содержит **90** наименований.

Содержание работы

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

Первая глава посвящена обзору существующих методов и инструментов поиска уязвимостей, основанных на статическом анализе кода, а также в ней вводятся основные понятия анализа помеченных данных, такие как исток помеченных данных, сток, передаточная функция и санитайзер.

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.

Конкретный смысл, вкладываемый в понятие «помеченности» данных зависит от конкретной задачи, например:

1. Выход за границы буфера в результате использования слишком большого или отрицательного индекса массива. Помеченными в данном примере считаются данные, получаемые из недоверенного источника и которые могут быть использованы для индексации буфера.
2. Утечка 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 переменных, указывающих на уже освобождённые области памяти.
4. Использование константных паролей, при которых помеченными считаются данные, использующиеся в качестве паролей или ключей шифрования.

Помеченные данные могут быть заданы через *пути доступа*, состоящие из базового значения (указателя или другого значения в программе) и последовательности байтовых смещений и разыменований относительно него. С точки зрения пользователя они могут выглядеть как `var`, `*(p + 10)`, `this->my_array[1]`. *Помеченный факт* определяется как $\langle A, I \rangle$, где A — путь доступа к помеченным данным, а I — любая дополнительная информация, приписанная факту. В большинстве задач $I = \varnothing$.

Путём распространения помеченности или *трассой* предупреждения, соответствующей уязвимости, будет называть последовательность пар $\langle N, D \rangle$, где N — вершина графа потока управления (ГПУ), а D — помеченный факт в этой вершине. Пары перечисляются в порядке от истока к стоку помеченности, в которой каждая последующая пара зависит от предыдущей. Пути распространения помеченности должно быть достаточно для пользователя, чтобы понять, откуда берутся помеченные данные и как достигают точки, в которой уязвимость может быть реализована.

Для поиска путей распространения помеченности можно использовать решение задачи IFDS (Interprocedural Finite Distributive Subset) о достижимости по межпроцедурно реализуемым путям на расширенном суперграфе. Под *суперграфом* понимается объединение всех графов потоков управления (ГПУ) функций в программе, с добавленными межпроцедурными рёбрами, соединяющие точки вызова с точками входа и выхода вызываемых функций. *Расширенным суперграфом* называется межпроцедурный ориентированный граф, вершинами которого являются пары $\langle N, D \rangle$, где N — вершина суперграфа, а D — помеченный факт. Любая трасса предупреждения может быть получена из некоторого пути в расширенном суперграфе.

Анализ или построение расширенного суперграфа начинается от истока помеченных данных и продолжается вдоль путей их распространения, в том числе за пределы текущей функции. Алгоритм позволяет эффективно исследовать любые пути в программе, не требуя специальной обработки

циклов и рекурсивных вызовов и отслеживая только значения помеченных переменных. Недостатком этого подхода является то, что анализировать программу нужно целиком — то есть максимальный размер анализируемой программы ограничен объёмом доступной оперативной памяти.

Для сравнения, одним из основных подходов к межпроцедурному статическому анализу является анализ на основе резюме функций: он состоит из набора внутрипроцедурных анализов, результатом каждого из которых является резюме, описывающее поведение функции — эти резюме применяются в месте вызова, обеспечивая межпроцедурность. Такой подход хорошо масштабируется и позволяет контролировать максимальное время анализа, выставляя ограничения на время анализа одной функции и размер её резюме. Недостатком такого подхода является то, что во время анализа неизвестно, через какие аргументы текущей функции будут передаваться помеченные данные, потому либо в резюме должны сохраняться все возможные зависимости по данным, либо часть межпроцедурных путей распространения помеченности будет теряться.

Также в данной главе рассматриваются существующие подходы и инструменты анализа программ и делается вывод о том, что анализ помеченных данных на основе задачи IFDS является перспективным методом обнаружения потенциальных уязвимостей, который позволяет обобщить различные классы потенциальных уязвимостей.

Бóльшая часть инструментов, реализующих данный метод, предназначена для высокоуровневых языков программирования, наподобие Java. В других анализ помеченных данных имеет сильные ограничения с точки зрения полноты из-за выбранного представления для помеченных данных, отсутствия анализа псевдонимов или недостаточного количества спецификаций для библиотечных функций, из-за чего эти инструменты не подходят для анализа реальных проектов на языках C и C++. Инструменты, реализующие IFDS без дополнительных расширений, могут испытывать проблемы с масштабируемостью из-за размера расширенного суперграфа: например, инструмент Flowdroid не смог проанализировать часть проектов даже имея 730 Гб оперативной памяти и 24 часа на анализ — хотя таких проектов было меньше 1%.

В конце главы делается вывод о том, что тестирование и динамический анализ кода являются важными методами, использующимися для поиска уязвимостей, но демонстрируют меньшую полноту и масштабируемость анализа, чем статические методы. Статическое символьное выполнение демонстрирует хорошие результаты по всем трём критериям, но может испытывать проблемы с анализом циклов, рекурсивных вызовов и сложных межпроцедурных путей, что влияет на полноту анализа на реальных проектах. Решение при помощи IFDS подхода демонстрирует наибольшую полноту, но меньшую точность и нуждается в изменениях для практической применимости.

В настоящее время автору неизвестно 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.

Вторая глава посвящена описанию предлагаемых методов повышения точности статического анализа помеченных данных.

В начале главы дано описание общей схемы работы анализатора, приведённой на Рисунке 1. Эта схема даёт представление об очерёдности и взаимосвязях между различными компонентами анализа. За схемой следует краткое описание проблем и решений, которым посвящены главы 2–5.

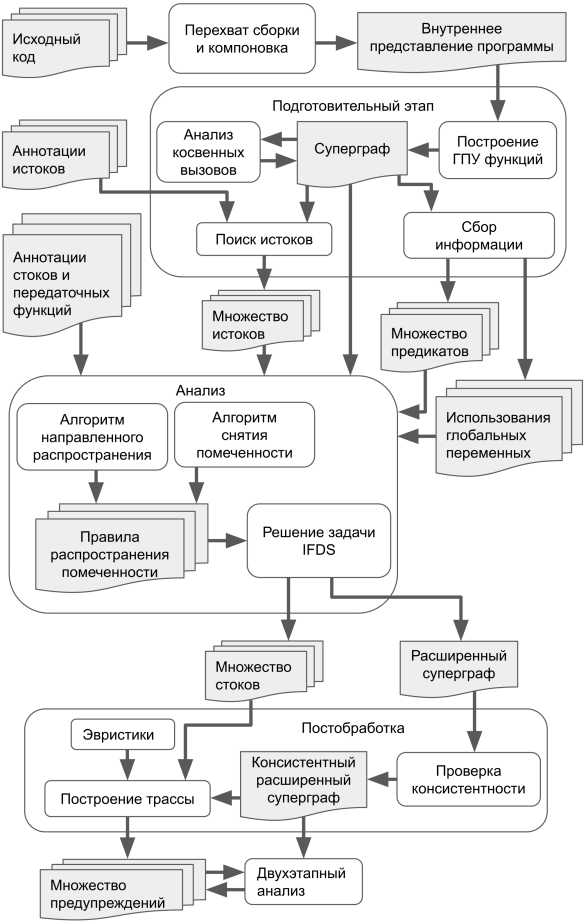

Рис. 1 — Общая схема работы анализатора

В разделе 2.1 рассматривается проблема отсутствия у алгоритма решения задачи IFDS чувствительности к путям: оно означает, что распространение помеченности зависит только от текущего помеченного факта D

и вершины суперграфа N и не различает одинаковые факты, пришедшие по разным путям выполнения программы. Это ограничение является существенным для обеспечения полиномиального времени работы алгоритма, однако в общем случае 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 1.

Листинг 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. IFDS анализ позволяет получить список трасс помеченности с высокой полнотой анализа и за приемлемое время.
2. Чувствительный к путям анализ проверяет найденные трассы предупреждения, отсекая часть ложных срабатываний за счёт проверки реализуемости предикатов путей — при этом анализ всей программы не требуется.

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.
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 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.

Было опробовано три варианта анализатора для второго этапа: KLEE, Svace и собственный проход по расширенному суперграфу.

В разделе 2.1.1 рассматривается использование статического символьного анализа из инструмента Svace в двухэтапном анализе. Информация о найденных парах ⟨исток, сток⟩ передаётся при помощи инструментации файла с биткодом анализируемой программы.

Двухэтапный анализ со Svace хорошо показал себя на выбранном подмножестве тестов Juliet Test Suite, устранив 100% ложных срабатываний и не уменьшив количество истинных. На OpenSSL с уязвимостью Heartbleed он уменьшил количество срабатываний на 37%, оставив предупреждение, соответствующее реальной уязвимости. Ключевыми недостатками было ощутимое увеличение общего времени анализа и невозможность проверить причины, по которым срабатывания были отсеяны.

В разделе 2.1.2 рассматривается двухэтапный анализ с дополнительным проходом по расширенному суперграфу в рамках самого инструмента, который позволяет выявлять неконсистентные пути распространения помеченности по трём критериям: несовместимость виртуальных вызовов, вызовов по указателю и целевых файлов в объединённом файле биткода проекта, что было невозможно в задаче IFDS из-за нечувствительности к путям. В Таблице 1 приведена оценка результатов — общее время анализа и объём оперативной памяти не зависит

от использования данного алгоритма, и исправляет трассу в части предупреждений на консистентную.

Таблица 1 — Алгоритм проверки консистентности путей

Проект	Всего предупреждений	Неконсистентных предупреждений	Общее время	Память
OpenSSL	1137	41	3 ч. 17 м.	3 Гб.
LibTIFF	283	1	7 м.	2 Гб.
binutils	1316	25	38 м.	10 Гб.

В разделе 2.2 рассматривается вопрос 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 ложных срабатываний, 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 предлагается 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 константами. Это наиболее распространённый способ проверки чисел, полученных из недоверенного источника, а предлагаемый алгоритм обладает чувствительностью к потоку выполнения и подходит для использования в решении задачи IFDS.

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 вычисление правого

операнда происходит в зависимости от результата проверки левого и на практике они находятся в разных базовых блоках, соединённых инструкцией условного перехода. Благодаря использованию дерева доминаторов, этот алгоритм подходит для использования в нечувствительном к путям анализе, поскольку пометки добавляются только если они верны независимо от конкретного пути выполнения. Этот подход позволил отсечь распространённый тип ложных срабатываний, связанных с использованием помеченного значения индекса массива.

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%).

В разделе 2.3 описываются другие 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.

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

Третья 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, этот параметр можно оценить только через сравнение с результатами работы других анализаторов. На тестовых наборах с заранее известными внедрёнными уязвимостями это значение можно посчитать достаточно точно.

В разделе 3.1 рассматривается проблема вызовов функций по указателю (косвенных вызовов), в которых вызываемая функция не может быть определена однозначно.

Для решения этой проблемы необходим анализ виртуальных и косвенных вызовов, который вычисляет набор функций-кандидатов для каждого такого вызова. В межпроцедурный граф 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 вызовов по указателю тривиальным подходом можно назвать подстановку всех функций из программы с подходящими типами параметров, адрес которых был взят хотя бы 1 раз. Этот подход плохо подходит для практического использования из-за большого количества ложных срабатываний, так типы параметров совпадают у большого количества несвязанных между собой функций. Помимо этого, такой подход не может гарантировать и подстановку всех возможных кандидатов, поскольку на практике в языках C и C++ может использоваться преобразование указателей на функцию к другим типам.

В данной работе предлагается метод получения множества кандидатов для вызова функции по указателю, путём сведения этой задачи к решению задачи IFDS. Фактом такого анализа является функция, адрес которой был взят, и путь доступа, по которому адрес хранится. Истоками являются инструкции взятия адреса функции, а также инструкции, использующие значения глобальных переменных, в инициализаторе которых фигурирует адрес какой-либо функции. Передаточные функции совпадают с теми, которые используются в основном анализе помеченных данных, а стоками являются вызовы функций по указателю. При достижении стока, функция из факта анализа сохраняется как потенциальный кандидат для

данного вызова и после окончания анализа добавляется в граф вызовов, как показано в Алгоритме 2

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 суперграфа, 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 инструкциями вызова по указателю в качестве стоков
    for all N ∈ Sinks do
        Candidates ← Candidates ∪ {⟨N, F⟩}
    end for
end for
```

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 функций. 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 может 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 анализатора, 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 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 2.

Таблица 2 — 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 наличия анализа косвенных вызовов на общую полноту и масштабируемость анализа можно увидеть в Таблице 3.

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

Таблица 3 — 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.

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 продолжительность анализа за счёт того, что было исследовано больше путей, но позволило найти 3 дополнительных предупреждения.

В разделе 3.2 рассматривается общая для большинства видов статического анализа проблема, заключающаяся в необходимости описывать поведение внешних для проекта (библиотечных или системных) функций. Для её решения обычно используются спецификации — упрощённые описания распространённых библиотечных функций, заменяющие их настоящее поведение. Спецификации могут создаваться либо разработчиками инструмента, либо его пользователями. Для анализа помеченных данных необходимы спецификации для трёх видов функций: истоков, передаточных функций и стоков помеченности.

В разделе 3.2.1 предлагается формат для пользовательского описания истоков, стоков и передаточных функций в текстовом формате, основанном на JSON, использующий регулярные выражения и учитывающий особенности перегрузки имён функций в C++. Для того чтобы определить, добавление аннотаций для каких функций принесёт наибольшую пользу, инструмент подсчитывает статистику, через аргументы каких ещё не аннотированных внешних функций чаще всего передавались помеченные данные — это позволяет проще находить возможные передаточные функции и стоки. Для истоков были созданы отдельные эвристики, анализирующие имена и типы аргументов всех внешних функций, вызываемых в программе — они описаны в разделе 3.2.2.

В разделе 3.2.2 предложена эвристика, опирающаяся на название функции и типы её аргументов, чтобы предположить, какие библиотечные функции могут иметь заданное поведение (например, осуществлять чтение данных из стороннего источника). Эта эвристика была реализована в инструменте и при включении автоматически добавляет найденные истоки к анализу, при этом итоговые предупреждения помечаются тегом `HEURISTIC_SOURCE`. В разделе приведена оценка результатов работы эвристик для функций «Чтение данных» и «Освобождение памяти» на OpenSSL. В сумме было обнаружено 200 новых функций, а все из них действительно оказались истоками помеченных данных. Параметр функции, через который помечаются данные, был верно определен эвристиками в

93% и 90% случаев соответственно, что позволяет использовать их в автоматическом режиме — без необходимости создания аннотаций вручную. В случае истока «Чтение данных» дополнительно удалось определить параметр, отвечающий за размер считываемой области памяти, в 47% случаев. Большое количество найденных функций типа «Освобождение памяти» (175) 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. Наличие таких эвристик в анализаторе сильно упрощает задачу аннотации новых истоков помеченности, специфичных для конкретного проекта, а потому важны для полноты анализа.

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

Четвёртая глава посвящена описанию предлагаемых методов повышения масштабируемости статического анализа помеченных данных, которая определяется зависимостью времени анализа и требуемой оперативной памяти от размера анализируемого проекта. Масштабируемость необходима для того, чтобы инструмент мог анализировать реальные проекты большого размера.

В разделе 4.1 описывается влияние распространения помеченности через глобальные переменные на общее время выполнения анализа и предлагается алгоритм для решения этой проблемы.

В Утверждении 1 показано, что попадание помеченности в глобальные переменные оказывает на масштабируемость анализа большее влияние, чем помеченные локальные переменные и параметры функции. Интуитивное обоснование заключается в том, что с ростом проекта увеличивается количество функций, а не их средний размер, и бóльшая часть помеченных данных может выйти за пределы текущей функции только через один из её параметров. Глобальные же переменные, будучи один раз помеченными, могут распространять её по всей программе без каких-либо ограничений.

Сформулируем несколько предположений:

1. Любой возникающий факт (кроме пустого) имеет базовое значение, являющееся либо локальным значением, либо параметром функции, либо глобальным значением.
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.
4. Увеличение общего размера программы происходит за счёт увеличения количества функций, а не их размера.

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.

В данной работе предлагается алгоритм направленного распространения глобальной помеченности, предназначенный для ускорения анализа проектов, использующих помеченные глобальные переменные. Для его работы необходимо предварительно вычислить множества функций F_{direct} и $F_{indirect}$ для каждой глобальной переменной G с помощью следующего алгоритма: перед началом анализа для каждой глобальной переменной G все функции в программе разбиваются на три непересекающихся подмножества:

1. F_{direct} — функции, непосредственно использующие G, то есть те, в коде которых есть явное упоминание этой переменной (например, её значение считывается или переписывается).
2. $F_{indirect}$ — функции, косвенно использующие G, то есть те, в которых G явно не упоминается, но есть вызовы функций, непосредственно или косвенно использующих переменную.
3. Не использующие G — все остальные функции, не вошедшие в первые два подмножества. Вычислять это подмножество в явном виде не требуется.

Множество F_{direct} получается явно из информации об использовании функций, предоставляемых инфраструктурой LLVM. Множество $F_{indirect}$ получается путём обхода графа вызовов в обратном направлении, начиная с функций из множества F_{direct}.

В ходе построения расширенного суперграфа, после вычисления помеченных фактов на выходе передаточной функции становятся возможны два действия:

1. *continue()* — распространение помеченного факта осуществляется как обычно, вдоль рёбер графа потока управления или вдоль межпроцедурных рёбер.
2. *move(nodes)* — вместо обычных, создаются рёбра к вершинам графа потока управления, перечисленным в *nodes*. Все вершины должны принадлежать той же функции, что и текущая.

Для помеченных фактов, не основанных на глобальной переменной, всегда выбирается действие *continue()*. Для глобальных фактов выбор действия осуществляется по Алгоритму 3.

Во время анализа, если текущий помеченный факт основан на глобальной переменной, то поведение зависит от того, к какому множеству относится текущая функция:

1. Для непосредственно использующих функций, факт распространяется по обычным правилам.

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**
  $nodes \leftarrow \varnothing$
  ⟨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 *nodes*, иначе в очередь поиска добавляются вершины, следующие за текущей, если они встретились впервые⟩
  $result \leftarrow$ MOVE($nodes$).
else
  $nodes \leftarrow$ {F.GET_RETURN_VERTEX()}
  $result \leftarrow$ MOVE($nodes$).
end if

2. Для косвенно использующих функций, факт передаётся в ближайшие вызовы косвенно или непосредственно использующих функций, или к выходу из текущей функции.
3. В остальных функциях этот факт не распространяется за ненадобностью, а в случае передачи факта в такую функцию, глобальный факт сразу передаётся в точку возврата из функции в неизменном виде.

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.*

Оценки работы алгоритма направленного распространения помеченности приведены в Таблице 4. Наибольший эффект этот алгоритм продемонстрировал на Juliet Test Suite, обеспечив ускорение анализа более

Таблица 4 — 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%)

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 одном 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.

В разделе 4.2 рассматриваются изменения в схеме использования решателя задачи IFDS, которые оказались необходимы для того чтобы инструмент стал применим на практике. В отличие от алгоритмов анализа, в которых анализ разбивается на последовательность внутрипроцедурных анализов отдельных функций с созданием аннотаций и применением их в последующих местах вызова, алгоритм табуляции предполагает построение межпроцедурного графа распространения помеченности для всей программы сразу. Основным преимуществом такого подхода является возможность «ленивого» анализа, при котором анализ осуществляется только для тех вершин и путей доступа, которые будут содержать помеченные данные. Также упрощается анализ рекурсивных вызовов и взаимно рекурсивных функций, когда неясен порядок, в котором следует анализировать функции.

Среди недостатков такого подхода в первую очередь следует отметить пространственную сложность алгоритма: объём необходимой для анализа оперативной памяти растёт пропорционально увеличению размера анализируемой программы — помимо прочего, об этой проблеме сообщали и пользователи инструмента Flowdroid.

Для более контролируемого потребления памяти можно осуществлять раздельный анализ истоков: после окончания анализа каждого отдельного истока и генерации предупреждений, построенный граф распространения удаляется и анализ следующего истока начинается с нуля. Это приводит к ухудшению временно́й сложности алгоритма за счёт того, что некоторые части графа будут проанализированы повторно, однако позволяет контролировать использование памяти: оно будет зависеть от объёма памяти, необходимого для хранения самой анализируемой программы и от максимального возможного размера графа от отдельного истока помеченности. Если при этом добавить ограничения на глубину и максимальное количество итераций анализа от одного истока, то полученный инструмент сможет закончить анализ почти любой программы, представление которой помещается в оперативную память — а под отсечение попадут только самые длинные пути распространения помеченности, которые с

большей вероятностью являются нереализуемыми на практике из-за несовместных условий пути.

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

Пятая глава посвящена описанию общей схемы работы инструмента, особенностей реализации алгоритма, а также алгоритмов и эвристик, оказавшихся полезными для практического применения анализатора: чтобы инструмент был полезен, нужно соблюдать требования к скорости его работы и количеству ложных срабатываний, которые он выдаёт. Чаще всего речь идёт о компромиссах и поиске баланса между масштабируемостью, точность и полнотой, поскольку многие изменения, улучшающие один из этих показателей, могут отрицательно влиять на два других. В конце главы приведена оценка общей работы инструмента на различных проектах с открытым исходным кодом.

В данной работе предполагается следующая общая схема работы инструмента:

1. Система перехвата сборки, предоставляемая инструментом Svace, переводит файлы исходного кода в файлы промежуточного представления программы.
2. Компоновщик собирает отдельные файлы промежуточного представления в единый анализируемый файл.
3. Инструмент загружает анализируемый файл и выполняет подготовительные этапы: построение суперграфа, определение набора истоков для каждого типа детекторов, анализ косвенных вызовов и т.д.
4. Для каждого истока выполняется решение задачи IFDS, результатом которого 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 предупреждений 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.

Оценка точности и масштабируемости работы инструмента на различных проектах приведена в Таблице 5. 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.

Таблица 5 — Оценка точности и масштабируемости анализа

Проект	Всего пред-ний	TP	Строк кода	Память	Время
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

Также было проведено сравнение на тестах, входящих в Juliet test suite, с тремя инструментами, реализующими статический анализ помеченных данных: Svace, Clang Static Analyzer (CSA) и Infer Static Analyzer. 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 6.

Наибольшую полноту продемонстрировали Irbis и Svace, причём только Irbis покрыл все выбранные тесты и не выдал ни одного ложного предупреждения. Инструменты CSA и Infer оказались наиболее легковесными, но покрыли в 3 раза меньше тестов.

Таблица 6 — 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

В **заключении** приведены основные результаты работы, которые 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. Разработанные модели и алгоритмы были положены в основу инструмента Irbis, реализованного на базе открытой компиляторной инфраструктуры LLVM и испытаны 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).

Шимчик Никита Владимирович

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

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

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