DOI: 10.15514/ISPRAS-2022-34(6)-4

Irbis: статический анализатор помеченных данных для поиска уязвимостей в программах на C/C++

[1] Н.В. Шимчик, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
[1,2] В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
[1,2] А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>

[1] Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Статический анализ помеченных данных может использоваться для обнаружения различных потенциальных уязвимостей в коде программ путём исследования потоков данных между истоками и стоками помеченных данных. Чаще всего помеченными называют данные, которые были получены из внешнего источника и не были должным образом проверены. Инструмент Irbis реализует статический межпроцедурный анализ помеченных данных на основе решения задачи IFDS (Interprocedural Finite Distributive Subset), а также различные расширения, улучшающие его масштабируемость, точность и полноту. В нём реализованы 4 детектора с разными определениями помеченных данных, используемыми для нахождения выхода за границы буфера, использования освобождённой памяти, использования константных паролей и утечек данных. Определения истоков, стоков и передаточных функций хранятся в формате JSON и могут изменяться пользователем. Мы сравнили результаты анализа на проекте Juliet Test Suite for C/C++ с несколькими другими анализаторами, такими как Infer, Clang Static Analyzer и Svace. Irbis смог продемонстрировать 100% покрытие на подмножестве тестов, имеющих отношение к помеченным данным для поддерживаемых нами CWE. При этом реализованные нами эвристики смогли подавить все ложные срабатывания на данном наборе тестов. Также мы демонстрируем масштабируемость и процент ложных срабатываний инструмента при запусках на реальных проектах и показываем примеры существующих уязвимостей, которые могут быть им обнаружены.

Ключевые слова: статический анализ; анализ помеченных данных; поиск уязвимостей

Для цитирования: Шимчик Н. В., Игнатьев В. Н., Белеванцев А. А. Irbis: статический анализатор помеченных данных для поиска уязвимостей в программах на C/C++. Труды ИСП РАН, том 34, вып. 6, 2022 г., стр. 51-66. DOI: 10.15514/ISPRAS-2022-34(6)-4

Благодарности. Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект №20-01-00581 А.

Irbis: static taint analyzer for vulnerabilities detection in C/C++

[1] N.V. Shimchik, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
[1,2] V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
[1,2] A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>

[1] Ivannikov Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2]Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. Static taint analysis can be used to find various security weaknesses and vulnerabilities in programs by discovering dataflow paths from taint sources to taint sinks. In most cases the data is called ”tainted” if it was obtained from an untrusted source without proper sanitization. In this paper we present a static taint analyzer Irbis. It implements analysis based on IFDS (Interprocedural Finite Distributive Subset) dataflow problem, as well as various extensions aimed at improving accuracy and completeness of the analysis. It supports different definitions of tainted data, which enables it to find such weaknesses as out of buffer access, use of freed memory, hardcoded passwords, data leaks and discover dataflow paths between user-defined sources and sinks. All sources, sinks and propagators definitions are stored in JSON format and can be adjusted to meet the users’ needs. We compare analysis results on Juliet Test Suite for C/C++ with several other analyzers, such as Infer, Clang Static Analyzer and Svace. Irbis manages to demonstrate 100% coverage on taint-related subset of tests for implemented CWEs, while suppressing all the false positives using heuristics. We also show performance and false positive rate on real projects, with examples of real vulnerabilities, which can be detected by Irbis.

Keywords: static analysis; taint analysis; vulnerabilities detection

For citation: Shimchik N. V., Ignatyev V. N., Belevantsev A. A. Irbis: static taint analyzer for vulnerabilities detection in C/C++. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 6, 2022. pp. 51-66 (in Russian). DOI: 10.15514/ISPRAS-2022-34(6)-4

Acknowledgments. This work was supported by the Russian Foundation for Basic Research, project №20-01-00581 A

1. Введение

В работе рассматривается задача поиска потенциальных уязвимостей и ошибок безопасности в коде программ на языках C и C++. Наличие уязвимостей является серьёзной проблемой для разработчиков программных продуктов, т.к. они могут быть использованы для несанкционированного доступа к данным или вмешательства в работу ПО. Ошибки могут годами не проявляться при обычном использовании программы, однако в случае обнаружения злоумышленниками создают угрозу сервисам, использующим этот код. Наиболее подвержены опасности программы, принимающие данные по сети и из других потенциально компрометируемых источников. Наличие ошибки в библиотеке делает уязвимыми все программы, ее использующие. Одним из известных примеров такого инцидента можно назвать CVE-2014-0160 [1,2], также известную как Heartbleed. Эта уязвимость в популярной криптографической библиотеке OpenSSL заключается в отсутствии проверки размера буфера, считанного из внешнего источника, и открывала злоумышленникам возможность специальным запросом получать содержимое некоторых областей памяти процесса.

Существует возможность находить такие уязвимости ещё на этапе разработки. Один из способов состоит в использовании статического анализа помеченных данных. Эта разновидность анализа сводится к обнаружению путей выполнения программы, ведущих от инструкций возникновения помеченных данных к инструкциям или функциям, в которых их использование нежелательно. В зависимости от того, что именно считается помеченными данными, такое определение подходит для описания различных классов ошибок и

потенциальных уязвимостей: от выхода за границы буфера, до использования константных паролей и утечки чувствительных данных. Подробнее об этом в разд. 2 и 3.

Для поиска уязвимостей в ПО существует множество других подходов и инструментов: динамический анализ [3-5], фаззинг [6-8], подходы, сочетающие динамический анализ и символьное выполнение [9-11]. Рассмотрение и анализ этих подходов находится за рамками данной работы, т.к. перечисленные подходы требуют непосредственного запуска программы.

Широко применяются для поиска уязвимостей и подходы на основе статического анализа, т.е. не требующие выполнения программы, включающие анализ абстрактного синтаксического дерева, анализ потоков данных [12], различные методы на основе символьного выполнения [13, 14], формальная верификация [15]. Среди распространенных статических анализаторов, реализующих такой тип анализа, можно упомянуть инструменты Svace [16], Clang Static Analyzer [17], Infer [18]. Все они ориентированы на поиск максимального числа истинных предупреждений за ограниченное время, поэтому оптимизированы таким образом, чтобы минимизировать вероятность выдачи ложного предупреждения.

Однако уязвимости часто скрываются в программах и проявляются только на редких путях выполнения в «необычных» ситуациях, которые, зачастую, эвристически отфильтровываются у статических анализаторов общего назначения. Irbis создается в результате необходимости альтернативного подхода, реализующего более консервативный анализ, при котором баланс между долей истинных и ложных предупреждений смещен с целью исключить возможность пропуска ошибки, но тем не менее допускающий это (в отличие от формальной верификации).

2. Статический анализ помеченных данных

Задача статического анализа помеченных данных состоит в построении пути выполнения между созданием и предварительно заданным способом использования помеченных данных без запуска программы. Возникновение помеченных данных происходит в результате выполнения функций или инструкций – *истоков (source)*. Сам анализ состоит в построении путей распространения помеченных данных в программе, приводящих к передаче помеченных данных в предварительно заданное множество функций или инструкций – *стоков (sink)*. В процессе распространения помеченные данные могут подвергаться манипуляциям, например, модифицироваться, передаваться в качестве аргументов в функции и т.д. Для того, чтобы определить, сохраняет ли указанная операция помеченность, вводятся *передаточные функции (propagators)* – правила распространения пометок для всех операций и *санитайзеры (sanitizers)* – функции или операции, снимающие помеченность с данных. Статический анализ помеченных данных может выполняться как на исходном коде, так и на бинарном представлении программы и анализирует все пути выполнения, включая недостижимые.

Таким образом, для задания отдельного детектора ошибки в инфраструктуре анализатора помеченных данных необходимо задать множества истоков, стоков, санитайзеров и направление анализа. Направление может быть как прямым, так и обратным и служит для оптимизации. Например, в детекторе константных паролей гораздо эффективнее проводить анализ в обратном направлении – от использования переменной в качестве пароля назад к записи в нее константной строки. В противном случае было бы необходимо помечать все строки в программе, лишь малая часть которых может впоследствии использоваться в качестве пароля. Передаточные функции, как правило, не зависят от конкретного детектора и реализованы единым образом для всех них.

Мы выделяем два подхода к реализации статического анализа помеченных данных: на основе статического символьного выполнения и на основе анализа потоков данных [19]. Первый подход предполагает распространение пометок по символьным значениям и способен

использовать информацию о различных зависимостях между переменными, а также учитывать условия на путях в программе, по которым проходят помеченные данные. Подход подразумевает однократный обход всех функций в обратном топологическом порядке по графу вызовов, чтобы собрать и сохранить в резюме информацию обо всех вызванных функциях. Таким образом подход обеспечивает масштабируемый межпроцедурный, чувствительный к путям, анализ, обеспечивающий высокую долю истинных срабатываний за счет использования SMT решателей для проверки достижимости путей и предусловий ошибки. Однако определенный порядок анализа функций приводит к невозможности обнаружения ошибок в случае, когда сток находится внутри нескольких обёрточных функций, т.к. неизвестно, могут ли на вход подаваться помеченные значения. Среди реализующих такой подход инструментов можно отметить Svace.

Второй подход основан на предложенном в работе [20] обобщении IFDS/IDE решателя, сводящего задачу к проблеме достижимости на графе. Множество задач анализа потоков данных, например, поиск достигающих определений или живых переменных, может быть переформулирован в виде межпроцедурной задачи анализа потоков данных с дистрибутивными передаточными функциями над конечными доменами фактов IFDS/IDE [21]. Эта задача относится к межпроцедурному анализу потоков данных, обладает чувствительностью к потоку и контексту вызова функции, но не обладает чувствительностью к путям: другими словами, предполагается, что распространение помеченности T в точке программы V не учитывает то, какой путь распространения предшествовал этой точке. Отсутствие чувствительности к путям предполагает большее количество ложных срабатываний по сравнению со статическим символьным выполнением, но зато такой подход позволяет покрыть почти все имеющиеся потоки данных в программе.

Во время анализа строится расширенный суперграф, каждая вершина которого задается кортежем ⟨Вершина ГПУ, Помеченный факт, Контекст вызова функции⟩. Построение вершин графа начинается с истока и каждая вершина посещается не более одного раза. Поскольку состояние вершины не зависит от того, каким именно путём анализ пришел в конкретную точку программы, и от значений прочих переменных, алгоритм не подвержен взрывному росту количества исследуемых путей в программе. Такой подход реализован в Irbis, SharpChecker [22], FlowDroid [12].

Одним из развивающихся в настоящее время инструментов является PhASAR [14] – инфраструктура статического анализа на основе LLVM [23], которая позволяет пользователю задавать и решать некоторые задачи анализа потоков данных, включая IFDS/IDE, применимый для реализации анализа помеченных данных. Однако готовых детекторов ошибок не реализовано, а приводится только образец, демонстрирующий тривиальный сценарий и не имеющий практической значимости. Инструмент имеет также и другие существенные ограничения: задача объявления и построения множеств истоков, стоков, санитайзеров, а также моделирования библиотечных функций в данном инструменте не решается, интеграция со сборкой проекта отсутствует. Однако наиболее существенным ограничением является использование LLVM-Value для идентификации помеченных данных, что не позволяет моделировать, например, отдельные элементы массива. Таким образом, PhASAR может стать перспективным инструментом после устранения важных для поиска уязвимостей ограничений.

Irbis реализует статический анализ помеченных данных на основе решения задачи IFDS (Interprocedural Finite Distributive Subset problem), предложенной в статье [24] и применённой для поиска уязвимостей в программах на языке Java в инструменте Flowdroid [12]. Выбор подхода обусловлен желанием добиться максимально возможного покрытия путей выполнения. Это достигается за счет большего количества ложных срабатываний. Однако основная проблема статического анализа помеченных данных состоит не в реализации подхода, а в разработке набора алгоритмов и эвристик, обеспечивающих приемлемое время работы и долю ложных срабатываний. Кроме того, для анализа языков C и C++

дополнительно необходима реализация анализа псевдонимов. Детали реализации статического анализа помеченных данных в нашем инструменте подробно изложены в статьях [25, 26]. Для исключения срабатываний, пропущенных другими анализаторами, реализован алгоритм анализа виртуальных вызовов и вызовов по указателю, позволяющий не терять помеченность при таких вызовах, эвристический алгоритм обнаружения истоков, позволяющий исследовать, например, библиотеки, для которых истоки и стоки находится в разных компонентах связности графа вызовов, алгоритмы моделирования окружения — функций без исходного кода.

3. Типы детекторов и обнаруживаемых уязвимостей

В традиционных инструментах анализа помеченных данных, как было отмечено ранее, детекторы задаются кортежем из четырех компонентов:

⟨мн-во истоков, мн-во стоков, мн-во санитайзеров, направление анализа⟩.

В Irbis отсутствуют функции-санитайзеры, однако применяется ряд эвристик, снимающих помеченность, описанных в разделе 4.3. Поскольку большинство функций не снимают помеченность с некоторых поданных на вход данных, а возвращают очищенные от помеченности данные в виде результата, то такие санитайзеры моделируются исключением функции-санитайзера из множества передаточных функций, останавливая таким образом распространение помеченности. Основные детекторы, поддерживаемые Irbis, представлены в табл. 1. Они сгруппированы по нескольким принципам, образуя иерархическую классификацию. Первичное деление выполнено по типу истоков, в результате чего выделено 4 группы истоков.

1) *Ненадежные входные данные* формируют истоки для многих детекторов критических уязвимостей. Они включают данные, которые могут быть модифицированы пользователем. Попадание таких данных без проверки в критическую инструкцию, например, индекс массива, или функцию, например, exec является серьезной уязвимостью. Ненадежные данные могут быть как результатом выполнения функции, так и формальными параметрами функции, например, argc, argv в функции main. Irbis предоставляет возможность выбирать интересующие аналитика группы истоков. Данные истоки могут приводить к уязвимостям, связанным с недостаточной проверкой входных данных (CWE-20).
2) *Чувствительные данные*, такие как, например, криптографические ключи, пароли, рассматриваемые в качестве истока помеченных данных, формируют класс уязвимостей типа «утечка данных». Запись чувствительных данных в файлы, вывод на экран или отправка по сети без шифрования могут скомпрометировать всю программу.
3) *Криптографические функции* являются истоком для анализа в обратном направлении в детекторах использования константных ключей шифрования, паролей.
4) *Указатель на участок освобожденной памяти* `p`, полученный, например, после выполнения `free(p);` или `delete p;` выступает в качестве истока для детекторов повторного освобождения или использования освобожденной памяти.

Для категории 1 предлагается более детальная классификация обнаруживаемых классов ошибок в зависимости от стоков. Одним из наиболее важных типов проблем безопасности являются ошибки в работе с памятью:

- 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).

Табл. 1. Список основных детекторов инструмента Irbis
Tab. 1. List of main detectors of the Irbis tool

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 (`free()`, `delete`)	разыменование указателя на освобожденную память
PASSED_TO_PROC_AFTER_FREE	освобожденная память (`free()`, `delete`)	использование в качестве фактического аргумента в вызове `free()`, `delete`
DOUBLE_FREE	освобожденная память (`free()`, `delete`)	

Другой группой проблем является 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.

В третью группу выделяется использование ненадежных данных в критических функциях, что может приводить, например, к запуску команд с повышенными привилегиями (Command

injection: CWE-78), внедрению данных в БД (SQL Injection: CWE-89, LDAP Injection: CWE-90).

Отдельно рассматриваются детекторы `GETLOGIN` и `GETHOSTBY`, поскольку они обнаруживают ситуацию, когда поведение программы определяется результатом, возвращаемым функциями `getlogin` и `gethostaddr`.

Также следует отметить, что для детектора `DATA_LEAK` набор истоков и стоков задаётся самим пользователем, поскольку информация о том, какие данные являются секретными, а также выбор конкретных способов утечки, зависит от конкретного проекта.

3.1 Теги типов детекторов

Для разбиения множества предупреждений внутри одного детектора используются теги. Они присоединяются к типу предупреждения в заданном порядке, образуя подгруппы, обладающие определенными общими свойствами. Это позволяет пользователю просматривать более «интересные» предупреждения в первую очередь, за счёт выделения с помощью тегов группы с наибольшим числом истинных срабатываний, характерных для анализируемого проекта или наоборот – использования тегов, обозначающих «плохие» предупреждения. Одно предупреждение может иметь одновременно несколько тегов.

- `MACRO` означает, что сток обнаруженной ошибки происходит из макроса. Если одно предупреждение с таким тегом является ложным, то и остальные такие срабатывания с большой вероятностью окажутся ложными, так макрос предполагает одинаковую реализацию стоков.
- `OVERTAINT` помечает предупреждения, для которых в процессе распространения пометок консервативно были помечены соседние поля структуры или класса. Например, если в структуре был помечен *i*-ый элемент массива, топри отрицательных или достаточно больших значениях *i* могли быть помечены соседние с массивом поля структуры.
- `OVERFLOW`, как правило, свидетельствует о высокой вероятности истинности предупреждения, т.к. помечает предупреждения, где возможно целочисленное переполнение индекса при обращении к буферу или размера выделенной памяти.
- `HEURISTIC_SOURCE` используется для срабатываний, исток которых был угадан Irbis на основе эвристик, что бывает полезно для библиотек, когда нельзя доказать, что функция, например, читает пользовательские данные.
- `UNLIKELY` используется для известных шаблонов кода, характерных для ложных срабатываний, которые в очень редких случаях могут быть реальной уязвимостью.
- `INCONSISTENT` помечает ложные срабатывания, которые связаны с девиртуализацией и разрешением косвенных вызовов, но из-за ограничений статического анализа могут оказаться истинными. Такой тег получают пути распространения пометок, на которых для одного объекта были последовательные вызовы виртуальных функций из различных классов-наследников или вызовы по указателю функций из разных групп инициализации.
- `VARARG` сигнализирует о том, что помеченные данные были переданы через функцию с переменным числом аргументов и возможны неточности, как правило избыточная помеченность.
- `MALLOC` помечает предупреждения, в которых сток соответствует `malloc` или `new`, поскольку для многих программ это может не считаться уязвимостью.
- `BUFFER_LENGTH` помечает группу потенциально истинных предупреждений, в которых у Irbis оказалось достаточно информации о размерах задействованных буферов, чтобы судить о возможности переполнения.

4. Схема работы инструмента

В этом разделе рассматриваются три вопроса: место инструмента в инфраструктуре инструмента Svace, общая схема самого анализатора и улучшения, которые были внесены для того, чтобы инструмент имел практическую применимость.

4.1 Инфраструктура

В качестве основы для представления программы наш инструмент использует не её исходный текст, а внутреннее представление LLVM, создаваемое компилятором Clang. Получить его для целого проекта можно различными способами, одним из самых доступных из которых является использование проекта WLLVM [27], который указывается вместо обычного компилятора при сборке проекта и может создавать файлы бит-кода, соответствующие его исполняемым файлам или библиотекам.

Наш анализатор во многом полагается на инфраструктуру анализатора Svace, который также использует внутреннее представление LLVM. Хотя мы не задействуем его подсистемы анализа, мы используем реализованную в нём систему перехвата сборки для получения файлов бит-кода отдельных модулей компиляции и информации о вызовах компоновщика. Затем мы запускаем собственный скрипт, который использует *llvm-link* для объединения отдельных файлов в соответствии с полученной информацией о ходе сборки – именно результат работы этого скрипта и подаётся на вход анализатору.

После окончания анализа, его результаты переводятся в формат *svres*, что позволяет экспортировать их обратно в инфраструктуру Svace и отобразить в удобном для пользователя виде: в браузере, с переходами по точкам трассы, поиском, возможностью разметки предупреждений и другими возможностями.

В результате вся схема анализа требует минимального участия пользователя: от него требуется только последовательно вызвать набор команд, одной из которых указать команду для сборки проекта (в большинстве случаев это `make`), дождаться окончания анализа, после чего открыть браузер для просмотра сгенерированных срабатываний. Существует также возможность ручного задания набора истоков, стоков и передаточных функций, а также выбора нестандартных опций запуска, но этот этап является необязательным.

4.2 Схема анализа

Анализатор получает на вход программу во внутреннем представлении LLVM и выполняет несколько подготовительных этапов, которые могут различаться в зависимости от выбранных опций;

1) построение графа потока управления (ГПУ) для всех функций в программе;
2) поиск кандидатов для всех вызовов виртуальных методов, использующий метаданные LLVM;
3) поиск кандидатов для вызовов функций по указателю, использующий решение задачи IFDS, в которой истоками являются операции взятия адреса функции, а стоками — вызовы по указателю;
4) построение графа вызовов;
5) определение ограничений на допустимые значения переменных, следующихиз условий в коде программе.

Затем для каждого включённого детектора решается соответствующая ему задача IFDS, а найденные пути от истоков до стоков экспортируются в виде трассы распространения помеченности.

1) Осуществляется поиск всех истоков в программе, соответствующих данному детектору. Запоминаются соответствующие истокам вершины ГПУ и пути доступа, по которым записываются помеченные данные.

2) Для каждой пары ⟨Вершина ГПУ, Помеченный факт⟩ запускается отдельный IFDS-решатель, который распространяет помеченный факт в соответствии с заданными передаточными функциями и стоками (в зависимости от детектора, факты могут распространяться не только в прямом, но и в обратном направлении). Для всех достигнутых стоков запоминается вершина расширенного суперграфа, представляющая собой тройку ⟨Вершина ГПУ, Текущий помеченный факт, Текущий контекст функции⟩ .
3) После окончания анализа очередного истока берётся соответствующий ему расширенный суперграф распространения помеченности и из него выделяется самый короткий путь от истока до каждого стока – эти пути экспортируются либо в текстовый формат, либо в формат, подходящий для экспорта в Svace.

В текущей версии инструмента появилась возможность выполнять некоторые этапы анализа параллельно. При выборе соответствующей опции, каждый исток может анализироваться в отдельном процессе, полностью независимо от других – при этом количество одновременно запущенных процессов определяется автоматически или указывается пользователем. Это позволяет задействовать больше ресурсов компьютера, но для проектов с небольшими графами распространения помеченности и большим количеством истоков стоимость накладных расходов на создание процессов и передачу данных между ними может превысить выгоду от параллельного анализа и приводить к замедлению вместо ускорения. В частности, такая ситуация может быть характерна для анализа тестовых наборов.

4.3 Особенности анализа

Наивная реализация алгоритма IFDS испытывала проблемы как с масштабируемостью, так и с полнотой и точностью анализа. Ключевыми для решения этих проблем стали следующие пункты:

1) Irbis использует отдельный IFDS-решатель для каждого истока; это приводит к замедлению, за счёт повторного анализа одних участков программы, но ограничивает максимальный размер расширенного суперграфа в каждый конкретный момент времени, решая проблему нехватки памяти при анализе больших проектов;
2) для понимания того, как ведёт себя помеченность при вызове библиотечных функций, необходимы аннотации, создаваемые вручную. В ходе анализа мы отслеживаем, какие функции чаще стирают помеченность, а значит нуждаются в аннотациях в первую очередь;
3) мы также создали различные эвристики, которые убирают часть «плохих» срабатываний, либо помечают их тегами, которые позволяют пользователю смотреть их в последнюю очередь; имеются и эвристики, которые помогают находить неизвестные истоки в программе, используя названия функций и типы их аргументов.

4.3.1 Аннотации

Аннотации – это краткие описания того, что делает данная функция с точки зрения распространения помеченности. Количество и качество аннотаций напрямую влияет на качество анализа, поскольку при проведении статического анализа кода программы доступны только реализации функций, реализованных в самой программе.

Аннотации могут использоваться не только для описания поведения библиотечных функций, но и для обобщения поведения известных функций, что позволяет ускорить анализ, избавляя от необходимости «честного» распространения помеченности внутри такой функции.

В нашем инструменте аннотации задаются в формате JSON и могут свободно дополняться самим пользователем, если имеющихся по умолчанию оказывается недостаточно. Аннотации делятся на три вида: описания истоков, передаточных функций и стоков. Они позволяют задавать:

- название или сигнатуру функции или целого набора схожих функций при помощи регулярных выражений;
- порядковые номера аргументов, через которые проходит помеченность: входные и выходные аргументы, в зависимости от того, идёт речь о стоках, истоках или передаточных функциях;
- количество разыменований указателя относительно базового аргумента, байтовые смещения в структурах и именованные поля;
- прочие специальные поля, позволяющие настраивать поведение указателя; например, поле `HIDE_IMPLEMENTATION` позволяет полностью игнорировать реализацию функции и использовать только её аннотацию.

4.3.2 Константная эвристика

По умолчанию анализатор предполагает, что если помеченные данные передаются по ссылке или указателю в библиотечную функцию, о поведении которой ничего не известно, то помеченность с этих данных может быть стёрта. Это поведение можно вручную переопределить при помощи создания аннотации, в которой явно указано, что значение данного аргумента функции не изменяется.

Поскольку создавать такие аннотации для всех возможных библиотечных функций не представляется возможным, была добавлена константная эвристика, которая использует ключевое слово `const`, имеющееся в исходном коде программы (но отсутствующее в биткоде LLVM), означающее, что данное значение предназначено только для чтения. Эта эвристика состоит из двух частей:

1) патч в компилятор Clang, который сохраняет в метаданные информацию об использованиях ключевого слова const в параметрах библиотечных функций;
2) отдельный метод в анализаторе, считывающий эти метаданные и сообщающий, если помеченный факт в данном вызове не будет переписан.

4.3.3 Виртуальные и косвенные вызовы

В языках C и C++ существуют вызовы, в которых на этапе компиляции неизвестно, какая именно функция будет вызвана. Более того, одна и та же инструкция в программе может в разные моменты выполнения приводить к вызову различных функций.

Такие вызовы бывают двух типов:

1) виртуальные – вызов метода, помеченного ключевым словом virtual; в этом случае может быть вызван как сам метод, так и метод одного из наследников класса, в зависимости от объекта, метод которого вызывается;
2) косвенные – вызов по хранимому в памяти указателю на функцию; в этом случае вызываемая функция полностью определяется тем, какой указатель был записан в данную ячейку памяти.

Консервативный подход к статическому анализу предполагает, что виртуальные и косвенные вызовы трактуются так же, как и вызовы неизвестной функции, однако на практике это сильно ограничивает полноту анализа.

Альтернативой является поиск всех возможных функций-кандидатов для таких вызовов, с предположением, что вызываться может любой из найденных кандидатов. Это может приводить к появлению новых ложных срабатываний, поскольку IFDS анализ нечувствителен к путям и не может проверить, что такая комбинация вызываемых кандидатов действительно возможна, но повышает полноту анализа в тех проектах, где часто используются виртуальные или косвенные вызовы.

Для виртуальных вызовов мы восстанавливаем иерархию классов при помощи метаданных, имеющихся в биткоде LLVM, после чего добавляем в список кандидатов все реализации вызываемого метода из классов-наследников.

Для косвенных вызовов мы осуществляем отдельный IFDS анализ, в котором истоками являются взятия адресов функций в программе, а стоками – их использование в вызовах по указателю. Все адреса, достигшие места косвенного вызова, добавляются в качестве его кандидатов.

4.3.4 Снятие помеченности с целочисленных переменных

Хотя Irbis не позволяет задавать санитайзеры в явном виде, в нём была реализована эвристика, позволяющая снимать помеченность с целочисленных переменных. Если включена опция `--integer-sanitization`, то такие переменные перестают считаться помеченными, если в данной точке программы их значение ограничено и сверху, и снизу путём выполнения соответствующих проверок.

Чтобы определить наличие таких ограничений, на предварительном этапе выделяются все сравнения целочисленных переменных с константами. В зависимости от типа сравнения (`<`, `>`, `==`) и того, является ли тип переменной знаковым или беззнаковым (беззнаковые ограничены снизу нулём), мы указываем, что в базовых блоках, соответствующих началу истинной и ложной веток условия, её значения ограничены сверху и/или снизу. Затем эта информация распространяется на те следующие за ними блоки, которые доминируются одновременно самим условием и началом выбранной ветки.

Во время анализа, при попадании помеченного факта в новый базовый блок, сначала проверяется, соответствует ли он одной из переменных, на которые в данном блоке существуют ограничения. Если это так, и ограничения есть одновременно и сверху и снизу, то помеченный факт стирается.

Данная эвристика работает только с условиями, расположенными недалеко друг от друга и только внутрипроцедурно, однако её оказывается достаточно для многих практических случаев, когда проверяется значение индекса массива или размера выделяемого буфера. В частности, именно она позволила избавиться от всех ложных срабатываний на тестовом наборе Juliet Test Suite [28].

5. Результаты

Для демонстрации результатов анализа использовались несколько проектов.

Для тестирования было выбрано четыре анализатора, реализующих статический анализ помеченных данных: Irbis, Svace, Clang Static Analyzer (CSA) и Infer Static Analyzer. Поскольку большинство из перечисленных инструментов являются анализаторами общего назначения и ищут различные ошибки, включая синтаксические, для большинства анализаторов учитывалась только часть срабатываний:

- для Irbis использовались все имеющиеся детекторы.
- Для Svace сравнение проводилось по всем срабатываниям, содержащим в своём названии слово *TAINT*: *TAINTED_INT*, *TAINTED_INT.LOOP*, *TAINTED_ARRAY_INDEX*, *TAINTED_PTR.FORMAT_STRING* и другим.
- Для Clang Static Analyzer сравнение проводилось по результатам типа *Use of Untrusted Data* и *Out-of-bound access* со включёнными детекторами *alpha.security.taint.TaintPropagation* и *alpha.security.ArrayBoundV2*.
- Для инструмента Infer Static Analyzer сравнение проводилось по результатам работы детекторов *InferBO* и *Quandary*.

Первым анализируемым проектом является подмножество тестового набора Juliet 1.3 test suite for C/C++ [28], 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 только в ОС Windows.

Среди оставшихся были отобраны 4656 тестов, в заголовках которых поле `BadSource` содержит слово `read`. В остальных тестах ошибка проявляется на каждом запуске, а не на специальным образом сформированных пользовательских данных, поскольку определяется некорректными константами в коде – обнаружение таких ошибок не имеет отношения к анализу помеченных данных, потому такие тесты в данной работе не рассматриваются.

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

В этом случае, после базового просмотра полученных предупреждений, все срабатывания на проекте, собранном с макросом `OMIT_GOOD`, можно считать истинными, а все срабатывания на проекте, собранном с макросом `OMIT_BAD`, можно считать ложными.

Результаты запусков на Juliet Test Suite приведены в табл. 2.

Табл. 2. Результаты анализа выбранного подмножества Juliet Test Suite
Table. 2. Results of the analysis of the selected subset of Juliet Test Suite

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

Все инструменты продемонстрировали сравнимое время анализа (меньше 1 часа в сумме по двум запускам), однако поскольку на этом проекте время сборки сопоставимо или превышает время анализа, а для инструментов CSA и Infer их нельзя разделить – сравнивать этот показатель напрямую нецелесообразно. Для анализа проекта инструментом Infer пришлось отредактировать имеющиеся сборочные файлы, так как перехват сборки этого анализатора не поддерживает абсолютные пути к компилятору.

Вторым тестовым проектом является библиотека OpenSSL версии 1.0.1f, содержащая уязвимость CVE-2014-0346 (Heartbleed) [2], которую должен обнаруживать инструмент. Учитываются те же типы срабатываний, что и для предыдущего проекта.

Результаты анализа приведены в табл. 3, с более подробным рассмотрением типов срабатываний инструмента Irbis в табл. 4. Срабатывания Irbis с «ослабляющими» тегами игнорировались.

Табл. 3. Результаты анализа OpenSSL 1.0.1f
Table 3. Analysis' results of OpenSSL 1.0.1f

Анализатор	Срабатываний	TP-rate	Время (мин.)	RAM (Гб)
Irbis	3996 (279)	12% (41%)	87	3
Svace	14	71%	13	6
CSA	22	14%	8	<1
Infer	24 (469)	15%	17	<1

Будем трактовать срабатывание как истинное в том случае, если по нему пользователь может предположить, что помеченные данные действительно достигают указанного стока – эксплуатируемость найденной ошибки безопасности не проверялась. Для каждого типа предупреждений мы вручную разметили не менее 20 случайным образом выбранных срабатываний.

Табл. 4. Оценка процента истинных срабатываний Irbis на OpenSSL (по 20 размеченных срабатываний на каждый тип предупреждения)
Table 4: Estimates of the percentage of true Irbis hits on OpenSSL (20 labeled hits per alert type)

Тип предупреждения	Всего	TP-rate
TAINTED_ARG	172	45%
TAINTED_ARG.BUFFER_LENGTHS	1	100%
TAINTED_ARG.MALLOC	5	40%
TAINTED_LOOP_CONDITION	64	40%
TAINTED_PTR.LOAD	895	25
TAINTED_PTR.STORE	262	50%
RESOURCE_INJECTION	6	100%
PROCESS_CONTROL	2	100%
HARDCODED_PASSWORD_EXTRA	1	0%
USE_AFTER_FREE	1182	0%
PASSED_TO_PROC_AFTER_FREE	1378	0%
DOUBLE_FREE	28	0%

Как можно заметить по этим таблицам, хотя инструмент и производит значительно большее количество срабатываний, однако 93% из них приходится на всего 4 типа предупреждений: в основном на детекторы использования освобождённой памяти и на детектор чтения и записи по помеченному указателю, которые пользователь может проигнорировать. Более того, все просмотренные ложные срабатывания, связанные с освобождённой памятью, являлись следствием того, что отсутствие чувствительности к путям мешает анализатору проверить корректность использования функции *CRYPTO_realloc_clean* – в теории, все эти срабатывания можно пометить новым тегом при помощи добавления ещё одной эвристики, либо убрать добавлением аннотации для этой функции. Большинство предупреждений о чтении и записи по помеченному указателю произошли в криптографических функциях, потому их проверка и классификация на истинные или ложные затруднена.

Для удобства в скобках мы указали статистику без этих 4 типов предупреждений, внёсших наибольший вклад в количество срабатываний. Для инструмента Infer в скобках приведена статистика по срабатываниям уровня L1, обладающим наибольшей достоверностью.

Из четырёх представленных инструментов только Irbis продемонстрировал срабатывание, соответствующее искомой уязвимости. Это предупреждение имеет тип *TAINTED_ARG*, его истоком является чтение данных функцией *BIO_read в s3_pkt.c:239*, а стоком — вызов memcpy с размером, задаваемым помеченной переменной `payload` в *d1_both.c:1487*.

Также, ранее мы демонстрировали возможность обнаружения уязвимости CVE-2018-15209 в проекте LibTIFF версии 4.0.9 в работе [25].

6. Заключение

Мы представили свой статический анализатор помеченных данных для программ на языках C/C++ Irbis, работающий в инфраструктуре Svace. Он реализует 4 основных детектора, решающих задачу IFDS для разных типов помеченных данных, что позволяет ему искать такие потенциальные уязвимости и ошибки безопасности, как выход за границы буфера, обращение к освобождённой памяти, использование константных паролей, утечки чувствительных данных и другие.

Из-за отсутствия чувствительности к путям и других ограничений выбранного метода анализа, этот анализатор обладает более высоким процентом ложных срабатываний на реальных проектах, чем инструменты на основе символьного выполнения, однако он способен распространять помеченность даже по длинным межпроцедурным путям в программах, что делает возможным обнаружение реальных уязвимостей – таких как Heartbleed.

Возможности анализатора были продемонстрированы как на тестовом наборе Juliet Test Suite, так и на реальных проектах, содержащих уязвимости – он показал высокую степень покрытия, при этом время выполнения оказалось сопоставимым с индустриальными инструментами.

Список литературы / References

[1] CVE - CVE-2014-0160. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160, accessed 01.11.2022.

[2] Heartbleed bug. Available at: https://nvd.nist.gov/vuln/detail/CVE-2014-0160, accessed 01.11.2022.

[3] Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Notices, vol. 42, issue 6. 2007, pp. 89-100.

[4] Haldar V., Chandra D., Franz M. Dynamic taint propagation for java. In Proc. of the 21st Annual Computer Security Applications Conference (ACSAC'05), 2005, pp. 303-311.

[5] Newsome J., Song D.X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proc. of the Network and Distributed System Security Symposium, 2005, 17 p.

[6] Sutton M., Greene A., Amini P. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional, 2007, 576 p.

[7] American fuzzy lop. Available at: https://lcamtuf.coredump.cx/afl/, accessed 01.11.2022.

[8] Godefroid P., Levin M.Y., Molnar D. SAGE: whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft. Queue, vol. 10, issue 1, 2012, pp. 20-27.

[9] Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008, pp. 209-224.

[10] Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. SIGPLAN Notices, vol. 46, issue 3, 2011, pp.:265-278.

[11] Cha S.K., Avgerinos T. et al. Unleashing mayhem on binary code. In Proc. of the IEEE Symposium on Security and Privacy, 2012, pp. 380-394.

[12] Arzt S., Rasthofer S. et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. ACM SIGPLAN Notices, vol. 49, issue 6, 2014, pp 259-269.

[13] Бородин А.Е., Горемыкин А.В. и др. Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace. Труды ИСП РАН, том 33, вып. 1, 2021 г., стр. 7-32. DOI: 10.15514/ISPRAS–2021–33(1)–1 / Borodin A., Goremykin A. et al. Searching for Taint Vulnerabilities with Svace Static Analysis Tool. Programming and Computer Software, vol. 47, issue 6, 2021, pp. 466-481.

[14] Schubert P.D., Hermann B., Bodden E. PhASAR: an inter-procedural static analysis framework for C/C++. Lecture Notes in Computer Science, vol. 11428, 2019, pp. 393-410.

[15] D' Silva V., Kroening D., Weissenbacher G. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, issue 7, 2008, pp. 1165-1178.

[16] Иванников В.П., Белеванцев А.А. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7 / Ivannikov V.P., Belevantsev A.A. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, issue 5, 2014, pp. 265-275.

[17] Arroyo M., Chiotta F., Bavera F. An user configurable clang static analyzer taint checker. In Proc. of the 35th International Conference of the Chilean Computer Science Society (SCCC), 2016, pages 1-12.

[18] Calcagno C, Distefano D. Infer: an automatic program verifier for memory safety of C programs. Lecture Notes in Computer Science, vol. 6617, 2011, pp. 459-465.

[19] Беляев М.В., Шимчик Н.В. и др. Сравнительный анализ двух подходов к статическому анализу помеченных данных. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 99-116. DOI: 10.15514/ISPRAS-2017-29(3)-7 / Belyaev M.V., Shimchik N.V. et al. Comparative analysis of two approaches to static taint analysis. Programming and Computer Software, vol. 44, issue 6, 2018, pp. 459-466.

[20] Bodden E. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In Procю of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, 2012, pp. 3-8.

[21] Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, pp. 49-61.

[22] Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2 / Koshelev V.K., Ignatiev V.N. et al. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, issue 4, 2017, pp. 268−276.

[23] Lattner C., Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In Proc. of the International Symposium on Code Generation and Optimization, 2004, pp. 75‑86.

[24] Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, pp. 49-61.

[25] Shimchik N.V., Ignatyev V.N. Vulnerabilities Detection via Static Taint Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 177-190. DOI: 10.15514/ISPRAS-2019-31(3)-14.

[26] Shimchik N., Ignatyev V., Belevantsev A. Improving accuracy and completeness of source code static taint analysis. In Proc. of the 2021 Ivannikov Ispras Open Conference (ISPRAS), 2021, pp. 61-‑68.

[27] A wrapper script to build whole-program LLVM bitcode files. Available at: https://github.com/travitch/whole-program-llvm, accessed 01.11.2022.

[28] Juliet C/C++ 1.3. Available at: https://samate.nist.gov/SARD/test-suites/112, accessed 01.11.2022.

Информация об авторах / Information about authors

Никита Владимирович ШИМЧИК – младший научный сотрудник ИСП РАН. Его научные интересы включают статический анализ программного обеспечения.

Nikita Vladimirovich SHIMCHIK is a researcher at ISP RAS. His research interests include static analysis of programs.

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник ИСП РАН, доцент кафедры системного программирования факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV, PhD in computer sciences, senior researcher at Ivannikov Institute for System Programming RAS and associate professor at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr.Sc., Leading Researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.