DOI: 10.15514/ISPRAS-2025-37(3)-20

Поиск утечек памяти и ресурсов в статическом анализаторе Svace

[1,2] *Н.Е. Малышев, ORCID: 0000-0002-5245-4915 <neket@ispras.ru>*
[1] *А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,2] *А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*
[1,3] *В.А. Семенов, ORCID: 0000-0002-8766-8454 <sem@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, г. Москва, Ленинские горы, д. 1.*

[3] *Московский физико-технический институт, 141701, Россия, Московская область, г. Долгопрудный, Институтский пер., 9.*

Аннотация. В статье представлен метод поиска утечек памяти и других схожих ресурсов в статическом анализаторе Svace. Формулируются требования к статическому анализатору, представляющие стоящую за Svace философию, кратко описывается основная инфраструктура анализа на основе межпроцедурного символьного выполнения с объединением состояний и показывается, как ее можно применить для поиска утечек. Описываются атрибуты, которые необходимо вычислить в ходе анализа, демонстрируется, как они вычисляются, как используются спецификации функций выделения и освобождения памяти, а также как учитываются участки памяти, выходящие из-под контроля анализатора. Предлагается способ учета отсутствующих в исходном коде ограничений на выделение и освобождение памяти с помощью создания искусственных функций. Представляются экспериментальные результаты работы метода на наборе тестов Juliet и открытом пакете Binutils, показывающие жизнеспособность предлагаемых идей.

Ключевые слова: статический анализ; символьное выполнение; утечки памяти; чувствительность к путям; анализ потока данных; анализ указателей.

Для цитирования: Малышев Н.Е., Бородин А.Е., Белеванцев А.А., Семенов В.А. Поиск утечек памяти и ресурсов в статическом анализаторе Svace. Труды ИСП РАН, том 37, вып. 3, 2025 г., стр. 291–302. DOI: 10.15514/ISPRAS–2025–37(3)–20.

Detecting Memory and Resource Leaks in the Svace Static Analyzer

[1,2] *N.E. Malyshev, ORCID: 0000-0002-5245-4915 <neket@ispras.ru>*
[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,2] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*
[1,3] *V.A. Semenov, ORCID: 0000-0002-8766-8454 <sem@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Lomonosov Moscow State University 1 Leninskie Gory, Moscow, 119991 Russia.*

[3] *Moscow Institute of Physics and Technology (National Research University) 9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

Abstract. The paper presents an approach to detecting memory and other resource leaks in the Svace static analyzer. We lay down a set of static analysis requirements that show the philosophy behind Svace, briefly describe the main analysis infrastructure based on an interprocedural symbolic execution with state merging, and show how this infrastructure can be applied to leak detection. We list attributes that are computed during analysis and present how this computation is performed, how allocation and release functions are modeled via specifications, how escaped memory is accounted for. We propose a way to provide external information not present in the source code to the analyzer via creating artificial functions. Experimental results on the Juliet test suite and the Binutils package show the viability of our ideas.

Keywords: static analysis; symbolic execution; memory leak; path sensitivity; dataflow analysis; points-to analysis.

For citation: Malyshev N.E., Borodin A.E., Belevantsev A.A., Semenov V.A. Detecting Memory and Resource Leaks in the Svace static analyzer. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 3, 2025. pp. 291-302 (in Russian). DOI: 10.15514/ISPRAS-2025-37(3)-20.

1. Введение

Утечки памяти – это вид ошибки, часто встречающийся в программах на языках с ручным управлением памятью. Из-за того, что при компиляции утечки не приводят к ошибкам, а при выполнении – к немедленному аварийному завершению программы, они нередко остаются незамеченными и присутствуют в современных программах на C/C++. При этом в языках со сборкой мусора актуальность приобретает другой тип утечек – утечки ресурсов: файлов, сетевых соединений, графических дескрипторов и пр.

В связи с этим для статических анализаторов утечки памяти и ресурсов – неизменная тема исследований. Традиционным подходом к поиску утечек является анализ потока данных, обычно выраженный в виде анализа распространения информации между источниками и стоками (source-sink). Именно это направление, как правило, и развивается в настоящее время; из исключений можно назвать инструменты сепарационной и некорректной логики [1, 2]. Среди важных инструментов можно упомянуть работы [3-7]. Их особенность, однако, заключается в том, что они предлагают специфичные алгоритмы поиска утечек, не применимые к другим анализам.

Несмотря на научную ценность таких алгоритмов, их применение неизменно влечет за собой следующие сложности: во-первых, для реализации предложенных алгоритмов в рамках уже развивающейся системы многоязыкового, промышленного статического анализа требуется адаптация, часто непростая; во-вторых, сами по себе такие инструменты могут показывать результаты хуже, чем поиск утечек, пусть несложный, но сделанный на инфраструктуре развитого анализатора – на практике часто именно качество инфраструктуры вносит определяющее значение в результаты. Как следствие, бывает тяжело сравнить

опубликованные алгоритмы между собой, т.к. честное сравнение предполагает их реализацию в одной и той же базовой инфраструктуре.

В статье мы описываем подход к поиску утечек ресурсов, применяющийся в статическом анализаторе Svace [8, 9]. На протяжении последних десяти лет развивались и претерпевали изменения как инфраструктура анализатора по организации чувствительного к путям анализа методами символьного выполнения, так и методы моделирования памяти. В разделе 2 мы коротко опишем постановку задачи и ожидаемые свойства статического анализатора, которые, по нашему мнению, позволят успешно применить его для поиска утечек. Мы также прокомментируем наличие или отсутствие этих свойств на примере нескольких открытых анализаторов. В разделе 3 изложим алгоритм поиска утечек, начав с краткого изложения организации используемых им анализов в инструменте Svace. За подробным разбором необходимой для нас архитектуры Svace мы отсылаем читателя к предыдущей статье [10]. Здесь же сконцентрируемся на описании алгоритма поиска утечек и полученных экспериментальных результатов, которые представлены в разделе 4. Раздел 5 содержит заключение и планируемые в дальнейшем исследования.

2. Утечки ресурсов и статический анализ

Утечкой ресурсов называется ситуация, при которой некоторый ресурс ограниченного объема (распределяемый через некую систему управления доступом) был получен, но не был возвращен системе обратно. Самым частым примером такого ресурса является память (динамическая); другие примеры – файлы, сетевые соединения и соединения к базам данных, графические объекты. Система управления предполагает наличие явных точек (вызовов), в которых ресурс выделяется и освобождается. В противном случае утечки не возникает (например, в языках со сборкой мусора не бывает утечек памяти).

Обычно ресурс управляется через некоторый дескриптор (указатель на память, файл и пр.). Тогда утечка регистрируется в тот момент, когда все дескрипторы (ссылки на ресурс) вышли из области действия, или остались только рекурсивные ссылки без внешних ссылок на цикл, при этом ресурс не был освобожден.

Для успешной разработки и реализации алгоритмов статического анализа, ищущих утечки памяти и ресурсов, необходим анализатор с определенной инфраструктурой. Требования к этой инфраструктуре формируются, исходя из особенностей современных программ, в которых можно выделить три основных момента: как написаны программы (размеры, используемые свойства языков); как они разрабатываются (особенности современных систем непрерывной интеграции); какая информация доступна анализатору (весь ли исходный код или его часть, некоторая внешняя информация о программе). Все они так или иначе освещались в литературе, поэтому здесь перечислим их кратко с пояснениями в самых важных моментах.

1. **Чувствительность анализа к потоку, полям, путям и контексту.** Необходимость чувствительности к потоку не требует пояснений. Чувствительность к полям, как правило, не требует существенных затрат ресурсов, но в нее часто включается и чувствительность к элементам массива, которой можно добиться, например, путем отслеживания нескольких первых и последних элементов, а для циклов выражая свойства элементов массива через аффинные итерационные функции.

   Масштабируемая чувствительность к контексту достигается за счет рассмотрения ограниченного количества контекстов вызова некоторой функции за счет либо группировки контекстов (по стеку вызовов, типу объектов) либо создания резюме функции (summary), хранящего релевантные результаты анализа для группы контекстов вызова (в идеале – параметризованного таким образом, что единственное резюме хранит результаты для всех возможных контекстов). Искусство выстраивания межпроцедурного анализа заключается в определении компромисса

между размером резюме, сложностью его применения и количеством сохраняемых для функции данных.

Чувствительность к путям выполнения реализуется анализатором за счет либо интерпретации некоторого фиксированного количества путей в каждой функции, либо с помощью символьного выполнения с объединением состояний. В первом случае есть риск пропустить ошибку, если исключить из рассмотрения путь, на котором она возникает, зато сложность анализа растет линейно относительно длины пути; во втором случае с ростом длины путей растет и сложность формул, а их упрощение также неизбежно приводит к потере ошибок.

2. **Точность и полнота анализа**. Если классические исследования, в частности, алгоритмов абстрактной интерпретации, предполагали консервативные анализы, в которых полнота (нахождение всех возможных ошибок) предпочиталась точности (отсутствию ложных срабатываний), то на практике, по нашему мнению, такой компромисс оправдан лишь для критических с точки зрения безопасности областей (авионики, атомной промышленности и пр.). Для больших программных систем общего назначения нормой стало допускать уменьшение полноты с целью увеличения точности, т.е. повышать процент истинных срабатываний в том числе за счет пропуска реальных ошибок, оговаривая контексты, в которых таковой пропуск возможен [11].
3. **Размеры обрабатываемых программ и циклы разработки в системах непрерывной интеграции.** Регулярное применение статического анализа в жизненном цикле разработки к программам в миллионы строк кода требует выстраивания упомянутых выше в разделе алгоритмов и свойств анализа таким образом, чтобы их масштабируемость позволяла анализировать программы за ночную сборку. Также важно иметь возможность организовывать инкрементальный анализ изменившихся частей программы для быстрого анализа каждого изменения (этот вопрос выходит за рамки статьи).
4. **Объяснение пользователю выдаваемого предупреждения об ошибке.** Программист или инженер безопасности, анализирующий результаты инструмента, должен справляться с определением истинности предупреждений с учетом, во-первых, упомянутых выше существенных размеров программ и соответствующего количества предупреждений, а во-вторых, того, что сам он может не являться экспертом в используемых языках программирования или нюансов кибербезопасности. При выдаче предупреждений разработчики статического анализатора не должны рассчитывать на схожий уровень знаний пользователя [12].
5. **Неявные контракты (предусловия и постусловия).** Практически постоянно анализируемая программа используется в неизвестных анализатору контекстах – по причине неполноты имеющегося кода, неточности анализа на большом объеме этого кода, либо тогда, когда информация о контекстах не содержится в самом коде. В этом случае анализатор должен предоставлять способы описать эту дополнительную информацию, либо в виде *спецификаций* функций или переменных, либо в виде конфигурируемых пользователем эвристических алгоритмов.

Последние три соображения, которые связаны больше со стратегией разработки анализатора, чем с самими алгоритмами анализа, приводят к следующим важным выводам:

- Анализатор может **не выдавать** даже истинные с точки зрения модели программы, памяти и алгоритмов анализа предупреждения, если 1) такие предупреждения не получается внятно объяснить пользователю на основе собранных данных или 2) вероятно существование неизвестных ограничений на контексты вызова, которые сделают эти предупреждения бесполезными для программиста.

- Анализатору стоит **выдавать** предупреждения не только об истинных с точки зрения его информации ошибках, но и о местах *потенциальных* ошибок, которые нереализуемы с учетом только известного кода программы, но в будущем в ходе разработки могут стать реальными ошибками (в частности, именно поэтому наш анализатор выдает ошибки в том числе в не используемом на данный момент коде либо в случаях, когда участок кода с местами возникновения и проявления ошибки полностью заключен в недостижимой области кода).

2.1 Схожие работы

Отсылая читателя к недавней работе по сравнению инструментов поиска утечек [13], кратко рассмотрим ряд известных открытых инструментов с учетом сказанного выше в разделе и нашей цели построить алгоритм, пригодный для использования в уже имеющемся промышленном статическом анализаторе.

Самым известным открытым анализатором языков C/C++ является Clang Static Analyzer [14], в котором реализовано символьное выполнение (однако без объединения состояний), межпроцедурный анализ (на основе встраивания); в отдельном компоненте есть вариант межмодульного анализа, также с ограниченной масштабируемостью. Тем самым анализатор удовлетворяет части выдвинутых требований по алгоритмам анализа, однако имеет сложности с масштабируемостью и, как следствие, применимостью для больших программ в регулярном цикле разработки.

Инструмент Smoke [6] находит исключительно утечки, причем масштабируемость чувствительного к путям анализа достигается за счет разреженных графовых представлений сущностей программы, которые используются для хранения лишь необходимой информации для поиска потенциальных утечек, и дальнейшей проверки совместности условий вдоль найденного пути с помощью SMT-решателя. Разреженные представления помогают добиться масштабируемости, однако плохо подходят для универсального статического анализатора, который поддерживает сотни детекторов и тем самым вынужден рассматривать с той или иной целью все инструкции программы.

Анализатор Sparrow [7] демонстрирует хорошую масштабируемость за счет использования резюме для сохранения результатов внутрипроцедурного анализа (параметризуются пути к точкам выделения, использования и освобождения памяти), однако его точность страдает из-за отсутствия чувствительности к путям выполнения.

Инструмент Infer [15] является широко известным примером использования сепарационной логики [1] для моделирования памяти. Infer выдавал предупреждения об ошибках тогда, когда не мог доказать их отсутствие, тогда как применение в дальнейшем некорректной логики [2] позволило перейти к выдаче ошибок в том случае, когда имеются основания подозревать их присутствие. Такая стратегия используется в Pulse, следующей версии Infer, поиск утечек памяти в которой присутствует, однако официально еще не выпущен.

3. Поиск утечек ресурсов

Предлагаемый алгоритм поиска утечек опирается на существующие возможности анализатора Svace, за подробным описанием которых мы отсылаем читателя к статье [10]. Здесь же представим очень краткое резюме всей необходимой инфраструктуры.

Анализатор Svace осуществляет межпроцедурный контекстно-чувствительный анализ на основе резюме, а внутрипроцедурный анализ – на основе символьного выполнения с объединением состояний. Внутреннее представление Svace IR содержит обычные арифметические и логические операции, инструкции cast, load/store, инструкции split и join для отображения потока управления, инструкции call и annotate для вызова и создания резюме, инструкцию assume для записи предусловий данного пути. Символьное выполнение

осуществляется над атрибутами – базовой единицей информации анализа, представляющей из себя, по сути, пару «ключ-значение» (примеры атрибутов – интервал значений, двоичный или троичный флаг, условие выполнения некоторого факта). Таким образом, символьное хранилище содержит соответствие не переменных и символьных выражений, а ключей и значений атрибутов. Кроме того, значения одних атрибутов могут выступать в качестве ключей других атрибутов. Объединение состояний происходит при обработке инструкции join, для корректной обработки которой каждый тип значений атрибутов должен определять соответствующую операцию слияния. Таким образом, степень чувствительности к путям и полноты получаемых формул может контролироваться на этом этапе через специфику операций слияния. Циклы анализируются путем раскрутки нескольких первых, абстрактной «средней» и последней итераций (в будущем можно использовать резюме циклов [16] для уточнения анализа).

Пример символьного выполнения над атрибутами показан на рис. 1. Слева на рисунке классическое символьное выполнение, связывающее символьные выражения с переменными `a` и `x`; после разделения потока управления появляются инструкции `assume`, и в переменную `a` записываются результаты вызова `B()` или `C()`. После объединения состояний значение, содержащееся по указателю `a`, выражается через тернарный оператор (дизъюнкцию). Справа ссылки, соответствующие переменным, связываются с атрибутами, обновляющими свои значения после разделения потока; из резюме функций `B()` и `C()` получаем условия выделения памяти под `b` и `c` (`x>2` и `x<3`). При слиянии состояний эти условия пересекаются с предикатом пути и объединяются, давая финальную формулу выделения памяти, записанной в `a`.

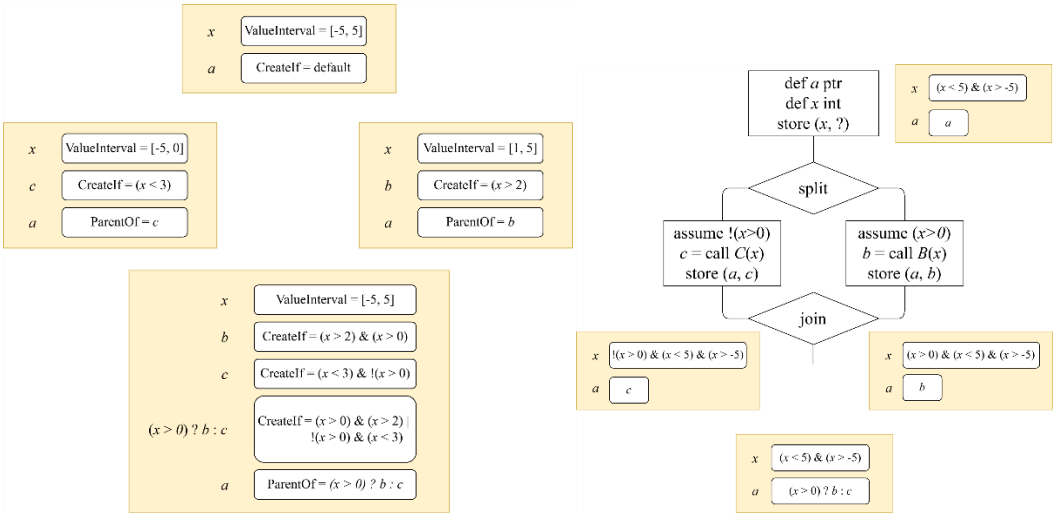

Рис. 1. Символьное выполнение над атрибутами.
Fig. 1. Symbolic execution with attributes.

Резюме функции, служащее передаточной функцией при обработке инструкции вызова, представляет из себя часть символьного хранилища, содержащего вычисленные атрибуты для аргументов функции, глобальных переменных и возвращаемого значения. На этом этапе формулы упрощаются и обрезаются для обеспечения масштабируемости. При применении резюме в формулы вместо формальных подставляются фактические параметры. Резюме неизвестных анализатору функций конструируются в ходе обработки спецификаций функций, в которых на анализируемом языке с помощью интерфейсов Svace описывается необходимая анализу семантика.

Выдача *предупреждений* происходит после того, как при обработке инструкций детектор проверяет условие ошибки и убеждается в его выполнимости. При необходимости условие транслируется в запрос стандарта SMT-LIB2 и проверяется на совместность SMT-решателем. Эффективное решение специфичных для статического анализа запросов исследовалось нами в работе [17].

3.1 Алгоритм анализа для обнаружения утечек

Вспомнив определение утечек из раздела 1, прежде всего заметим, что Svace не оперирует специальными сущностями для представления ячеек памяти, как на ранних этапах развития инструмента; вместо этого каждой отслеживаемой ячейке памяти соответствует два выражения над значениями, обозначающих ссылку и ее содержимое. Отношение между ссылкой и содержимым также может рассматриваться как атрибут и является объектом для символьного выполнения. Срез графа атрибутов, содержащий только подобные отношения между значениями, будем для краткости называть *графом значений*. Для сложных структур данных – записей и массивов – хранится базовая ссылка и смещение; в случае больших массивов, как и для циклов, моделируется несколько первых и последних элементов, а также абстрактный «средний».

Утечка определяется в тот момент, когда при выходе ссылки (дескриптора) из области действия не осталось других достижимых ссылок на указываемый ресурс. Так происходит либо при выходе из функции, либо при записи по ссылке нового значения инструкциями store или call, поэтому утечки проверяются при обработке детектором этих инструкций. Достижимые ссылки определяются обходом графа значений, содержащего в том числе указательные отношения (points-to).

Для тех ссылок, что оказались недостижимы, необходимо определить, был ли выделен и впоследствии освобожден ресурс. Мы вводим два атрибута, CreateIf и ReleaseIf, которые соответствуют условиям выделения и освобождения. Предупреждение выдается, если формула, состоящая из конъюнкции CreateIf, отрицания ReleaseIf, предиката пути и некоторых глобальных условий, оказывается совместной. SMT-решатель в таком случае предоставляет модель, используя которую, строится пример утечки. Из основной инфраструктуры Svace используется и добавляется в глобальные условия атрибут ValueInterval, позволяющий точнее учесть арифметические операции в условиях пути, а также результаты анализа недостижимого кода. В случае, когда анализ обладает неполной информацией о программе (из-за обрезания формул или резюме, или из-за специфичных свойств языка), вычисляется отдельный атрибут EscapeValue, и при его наличии предупреждение не выдается, чтобы избежать дополнительных ложных срабатываний.

Опишем более подробно устройство необходимых анализу атрибутов:

- **Анализ указательных отношений** выполняется через граф значений, где ключом служат выражения для ссылок, а значением – выражения для значений в памяти. Слияние выполняется через дизъюнкцию (классическое объединение состояний). В начале функции создаются ссылки для параметров, при обработке вызова – для возвращаемого значения и изменившихся параметров; в инструкции load связывается ссылка и указываемое значение, а в инструкции store – обновляется связь ссылки с новым значением.
- Атрибут **CreateIf** отвечает за анализ выделения памяти и связывает указательные значения с условиями, при выполнении которых для них выделена память. При обработке вызова, если произошло выделение новой памяти для возвращаемого значения или аргументов, значение атрибута переносится из резюме в контекст вызывающей функции. Спецификации функций выделения памяти (malloc, new и пр.) используют специальный вызов интерфейса анализатора sf_new (т.н. спецфункцию), которая устанавливает атрибут в истину.

- Атрибут **ReleaseIf** отвечает за анализ освобождения памяти и связывает указательные значения с условиями, при выполнении которых память для этих значений освобождается. Аналогично, при обработке вызова при освобождении памяти аргументов или возвращаемого значения значение атрибута переносится из резюме в контекст вызывающей функции. Спецификации функций освобождения памяти используют спецфункцию sf_delete, чтобы установить атрибут в истину.
- Атрибут **ValueInterval** из основной инфраструктуры Svace связывает символьные выражения с целочисленными интервалами. Значения атрибута сливаются и обрабатываются передаточными функциями арифметических и битовых операций обычным для интервального домена способом [18]; соответственно, возможна потеря точности при слиянии из-за того, что результат операции должен быть представим единственным интервалом. Тем не менее, для значения, являющегося результатом слияния, в графе значений можно найти составные части и при необходимости восстановить точность. При обработке инструкции assume интервал модифицируется с учетом условия из инструкции, а при обработке вызова переносится в контекст вызывающей функции.
- Атрибут **EscapeValue** устанавливается для тех выражений, которые «выходят» из-под контроля анализа, и становится невозможно или не нужно определять, были ли они освобождены. В частности, при обработке вызова он устанавливается для возвращаемых значений и выражений, на которые указывают аргументы, в случае, если вызываемая функция неизвестна анализу. Также может устанавливаться спецфункцией sf_escape, что используется в ряде библиотечных функций (например, realloc).

Отдельно нужно упомянуть метод, используемый для анализа утечек в случае ограничений, не следующих напрямую из исходного кода. Например, в языке C++ деструктор должен освобождать выделенную в конструкторе память, и если этого не происходит, мы сочтем такую ситуацию ошибочной, даже если в видимом анализаторе участке программы он не реализуется (вспомним требования к анализу и выводы, обсуждавшиеся в разделе 2). Общий метод представления таких ограничений и соответствующего анализа является предметом отдельной статьи; здесь же заметим, что в контексте утечек мы создаем искусственные функции, задающиеся некоторой парой функций: сначала в искусственную функцию вставляется тело первой функции пары, затем – вызов второй функции пары. Мы создаем функции для пар «конструктор-деструктор», «конструктор-оператор присваивания», «оператор присваивания-оператор присваивания» и «оператор присваивания-деструктор». Эти искусственные функции анализируются обычным образом в конце анализа, т.к. они не вызываются и не требуют создания резюме. Если в ходе анализа обнаруживаются ошибки, мы сообщаем об утечке памяти в программе.

4. Экспериментальные результаты

Предложенный алгоритм поиска утечек реализован в анализаторе Svace для утечек памяти и ресурсов. Мы протестировали его на тестах из набора Juliet [19] для класса CWE-401 (Missing Release of Memory after Effective Lifetime), определив число истинных и ложных срабатываний. Для открытых проектов мы оценили результаты на операционной системе Tizen и пакете Binutils версии 2.24.

На тестах Juliet для языка C мы определили 868 функций как имеющие ошибку, из них успешно выдали 560 ошибок, а из безошибочных функций (которых всего 2266) выдали ложное срабатывание в 32-х (см. табл. 1). Уровень ложных срабатываний (FPR) вычислили как FP / (FP + TN), а уровень пропусков (FNR) – как FN / (FN + TP). Аналогично вычислили данные для языка C++ за тем исключением, что после ручного просмотра часть выданных

срабатываний были исключены из рассмотрения, так как несмотря на то, что они были выданы в тестах, помеченных как «хорошие» с точки зрения набора, на самом деле эти тесты содержали релевантную ошибку в другом месте (а именно, в парах конструктор «конструктор-оператор присваивания», описанных нами выше).

Для пакета Binutils мы оценивали результаты отдельно по каждому типу детектора утечки памяти, который реализован в рамках анализатора. Все варианты детектора используют одну и ту же инфраструктуру, описанную нами выше, но разнятся в стратегии выдачи и показываемых ситуациях – такое разделение упрощает работу с результатами детектора. Общее название всех вариантов – MEMORY_LEAK, среди них базовый детектор (без суффикса в имени) содержит наиболее надежные срабатывания, которые были дополнительно отфильтрованы набором эвристик. Детектор MEMORY_LEAK.EX содержит те срабатывания, для которых составленные формулы были признаны совместными, однако дополнительной фильтрации эвристиками предпринято не было. Варианты .PAIR и .CTOR служат для предупреждений, выданных при генерации искусственных функций, как описано в конце раздела 3. Варианты .STRUCT и .STRDUP содержат подмножества срабатываний основного детектора, выделенные для удобства работы и содержащие ситуации соответственно со структурами и функцией strdup.

Результаты разметки представлены в табл. 2. Выделены истинные и ложные срабатывания, а также срабатывания “won’t fix”, которые являются истинными, но, как правило, разработчики их не исправляют. Как видно, процент истинных срабатываний превысил 80%, что по сравнению с более ранними результатами статьи [10] объясняется улучшениями чувствительного к путям анализа в части отслеживания условий отдельных веток при слиянии имеющих отношение к утечкам атрибутов.

Табл. 1. Результаты на пакете Juliet CWE-401.
Table 1. Juliet CWE-401 results.

Язык	TP	TN	FP	FN	FPR	FNR
C	560	2234	32	308	1,4%	35,5%
C++	511	1929	35	279	1,8%	35,3%

Табл. 2. Результаты на пакете Binutils.
Table 2. Binutils results.

Детектор	**Всего**	**TP**	**FP**	**WF**
MEMORY_LEAK	20	16	4	0
MEMORY_LEAK.EX	50	45	5	0
MEMORY_LEAK.STRUCT	18	10	8	0
MEMORY_LEAK.STRDUP	0	0	0	0
MEMORY_LEAK.CTOR	0	0	0	0
MEMORY_LEAK.PAIR	0	0	0	0
Всего	**88**	**71**	**17**	**0**

На рис. 2 представлены два примера срабатываний на пакете Binutils. Они связаны с обработкой нескольких выделений памяти (проверяется лишь одно, код на рисунке справа) и с ошибочной организацией потока управления (код на рисунке слева). Остальные ошибки в этом пакете чаще всего связаны с отсутствием освобождения ранее выделенной памяти при

возникновении неожиданных ситуаций в функциях, при которых они не может продолжать свою работу.

Тестирование на исходном коде операционной системе Tizen [18], содержащей более сотни написанных на C и C++ пакетов, показало для предыдущей версии алгоритма утечек около 67% истинных срабатываний (включая “won’t fix”) [10]; разметка результатов последней версии реализации пока не проведена.

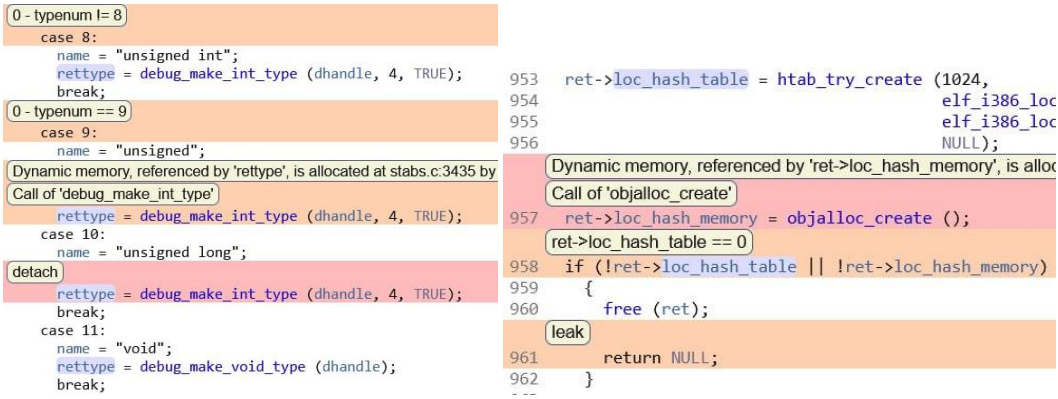

Рис. 2. Примеры ошибок из пакета Binutils.
Fig. 2. Binutils package error examples.

5. Заключение

В статье представлен подход к поиску утечек памяти и ресурсов на основе инфраструктуры промышленного статического анализатора Svace. Мы описали требования, которым, по нашему мнению, должен удовлетворять инструмент для успешного поиска ошибок в больших программах, и показали, как на его основе построить анализ для обнаружения утечек. Результаты тестирования нашей реализации на открытых пакетах показывают, что сформулированные в разделе 2 требования проходят проверку практикой. В дальнейшем мы планируем развивать подход к представлению в анализе внешней для программы информации, как показано на примере искусственных функций в разделе 3, и проводить эксперименты с резюме вложенных циклов и SMT-решателями, в том числе специализированными.

Список литературы / References

[1]. J. C. Reynolds, "Separation logic: a logic for shared mutable data structures," Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark, 2002, pp. 55-74, doi: 10.1109/LICS.2002.1029817.

[2]. Peter W. O'Hearn. Incorrectness logic. Proc. ACM Program. Lang. 4, POPL, Article 10 (January 2020), 32 pages. https://doi.org/10.1145/3371078

[3]. Wen Li, Haipeng Cai, Yulei Sui, and David Manz. PCA: memory leak detection using partial call-path analysis. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, 1621–1625. https://doi.org/10.1145/3368089.3417923

[4]. Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In Proceedings of the 25th International Conference on Compiler Construction (CC '16). Association for Computing Machinery, New York, NY, USA, 265–266.

[5]. Yungbum Jung and Kwangkeun Yi. 2008. Practical memory leak detector based on parameterized procedural summaries. In Proceedings of the 7th international symposium on Memory management (ISMM '08). Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1375634.1375653 https://doi.org/10.1145/2892208.2892235

[6]. G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou and C. Zhang, "SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code," 2019 IEEE/A CM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 72-82, doi: 10.1109/ICSE.2019.00025.
[7]. X. Sun et al., "A Projection-Based Approach for Memory Leak Detection," 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 2018, pp. 430-435, doi: 10.1109/COMPSAC.2018.10271.
[8]. A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, D. Zhurikhin. Design and development of Svace static analyzers. In 2018 Ivannikov Memorial Workshop (IVMEM), Yerevan, Armenia, 2018, pp. 3-9, doi: 10.1109/IVMEM.2018.00008.
[9]. Бородин А.Е., Белеванцев А.А. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды Института системного программирования РАН. 2015;27(6):111-134.
[10]. N. Malyshev, A. Borodin and A. Belevantsev, "A Comprehensive Approach to Finding Resource Leaks via Static Analysis," 2024 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russian Federation, 2024, pp. 1-9, doi: 10.1109/ISPRAS64596.2024.10899157.
[11]. Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. Commun. ACM 58, 2 (February 2015), 44–46. https://doi.org/10.1145/2644805
[12]. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53, 2 (February 2010), 66–75. https://doi.org/10.1145/1646353.1646374
[13]. H. Aslanyan, Z. Gevorgyan, R. Mkoyan, H. Movsisyan, V. Sahakyan and S. Sargsyan, "Static Analysis Methods For Memory Leak Detection: A Survey," 2022 Ivannikov Memorial Workshop (IVMEM), Moscow, Russian Federation, 2022, pp. 1-6, doi: 10.1109/IVMEM57067.2022.9983955.
[14]. “Clang Static Analyzer.” [Online]. Available: https://clanganalyzer.llvm.org/
[15]. Calcagno, C. et al. (2015). Moving Fast with Software Verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds) NASA Formal Methods. NFM 2015. Lecture Notes in Computer Science(), vol 9058. Springer, Cham. https://doi.org/10.1007/978-3-319-17524-9_1
[16]. C. Scherb, L. B. Heitz, H. Grieder, and O. Mattmann, “Divide, conquer and verify: Improving symbolic execution performance,” arXiv preprint arXiv:2310.03598, 2023.
[17]. N. Malyshev, I. Dudina, D. Kutz, A. Novikov and S. Vartanov, "SMT Solvers in Application to Static and Dynamic Symbolic Execution: A Case Study," 2019 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russia, 2019, pp. 9-15, doi: 10.1109/ISPRAS47671.2019.00008.
[18]. Операционная система Tizen. [Online]. Available: https://docs.tizen.org/platform/get-started/open-source-project/

Информация об авторах / Information about authors

Никита Евгеньевич МАЛЫШЕВ – младший научный сотрудник ИСП РАН, ассистент кафедры системного программирования ВМК МГУ. Сфера научных интересов: статический анализ, SMT-решатели, оптимизации программ.

Nikita Evgenyevich MALYSHEV – researcher at ISP RAS, assistant at Chair for System Programming of Department of Computational Mathematics and Cybernetics at MSU. Research interests: static analysis, SMT solvers, program optimization.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher. Research interests: static analysis for finding errors in source code.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор кафедры системного программирования ВМК МГУ. Сфера

научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., leading researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.

Виталий Адольфович СЕМЕНОВ – доктор физико-математических наук, профессор, заведующий отделом системной интеграции и прикладных программных комплексов ИСП РАН с 2015 года. Сфера научных интересов: модельно-ориентированные методологии и инструменты программной инженерии для создания цифровых платформ и мультидисциплинарных программных комплексов, визуализация и компьютерная графика, технологии информационного моделирования в архитектуре и строительстве, проектное управление и календарно-сетевое планирование.

Vitaly Adolfovich SEMENOV – Dr. Sci. (Phys.-Math.), Prof., Head of the Department of System Integration and Multi-disciplinary Applied Systems of the ISP RAS since 2015. Research interests: model-driven methodologies and CASE toolkits for creating digital platforms and advanced applied systems, visualization and computer graphics, building information modeling, project management and scheduling.