Статический анализатор Svace как коллекция анализаторов разных уровней сложности

[1]А.Е. Бородин <alexey.borodin@ispras.ru >
[1,2]А.А. Белеванцев <abel@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1., стр. 52, факультет ВМК

Аннотация. В статье описывается статический анализатор Svace, разрабатываемый в ИСП РАН.

Текущая версия анализатора осуществляет поиск ошибок в программах, написанных на языках Си, Си++, Java и Си#. Svace осуществляет поиск дефектов различных типов, включая ошибки разыменования нулевого указателя, переполнение буфера, использование неинициализированных переменных, утечки памяти, двойные блокировки, наличие недостижимого кода, несогласованность конструкторов и деструкторов, ошибки деления на ноль, возвращение адреса локальных переменных, использование объектов после удаления. Целью анализа является поиск как можно большего количества дефектов при приемлемом количестве ложных срабатываний и времени анализа.

Дефекты в программе имеют разную природу и для их поиска необходимо правильно выбрать алгоритм анализа. Хороший инструмент будет включать в себя как простые детекторы, использующие синтаксический анализ, так и сложные детекторы, позволяющие найти нетривиальные межпроцедурные ошибки. Построение инструмента на основе нескольких анализаторов позволяет использовать преимущества этих видов анализаторов и находить больший диапазон ошибочных ситуаций.

Инструмент Svace состоит из набора анализаторов реализующих анализы разных типов: анализ на основе абстрактного синтаксического дерева, консервативный анализ потока данных для одной функции, потоко- чувствительный и межпроцедурный неконсервативный анализ с возможностью использовать чувствительность к путям.

Межпроцедурный анализ осуществляется на основе аннотаций. При таком подходе после анализа функции создаётся её аннотация, описывающая интересующие эффекты вызова функции. Аннотация используется при обработке вызова функции для эмуляции вызова, что позволяет избежать повторного анализа функции. На основе анализа отдельных функций реализован анализ пар конструкторов и деструкторов Си++, позволяющий находить несогласованность при их написании. Все описываемые

анализы иллюстрируются примерами ошибок, найденными анализатором на проектах с открытым исходным кодом.

Ключевые слова: статический анализ; язык Си; дефекты в исходном коде; абстрактное синтаксическое дерево; потоковая чувствительность; межпроцедурный анализ; чувствительность к путям; неконсервативный анализ; разыменование нулевого указателя; утечки памяти.

DOI: 10.15514/ISPRAS-2015-27(6)-8

Для цитирования: Бородин А.Е., Белеванцев А.А.. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

1. Введение

В современном обществе программное обеспечение используется повсеместно: в телефонах и компьютерах, в медицинском оборудовании, в банках, в самолётах и автомобилях, в детских игрушках и электростанциях. Зависимость от программного обеспечения повышается с каждым днём, а сами программы становятся сложнее и больше. Было показано, что плотность ошибок также растёт с размером программного обеспечения [1]. Многие ошибки могут приводить к катастрофическим последствиям. Поэтому при разработке программного обеспечения поиску и исправлению ошибок уделяют особое внимание.

Для поиска ошибок используют различные методики и инструменты: ручное и автоматическое тестирование, инструменты статического анализа, динамический анализ и др. Жизненный цикл разработки программ в крупных компаниях обязательно включает в себя применение инструментов статического анализа [2, 3]. Дополнительные требования к использованию статического анализа регламентировано регуляторами как в России (Роскомнадзор, ФСБ России [4]), так и в других странах.

Статические анализаторы осуществляют поиск ошибок в программе без её фактического запуска. При этом, как правило, сразу анализируется множество путей исполнения. Благодаря тому, что статические анализаторы просматривают сразу все пути исполнения и анализируют пути независимо от вероятности их выполнения, анализаторы находят многие ошибки на редко исполняемых путях, которые часто остаются незамеченными во время тестирования. Особенно это касается кода, обрабатывающего ошибочные ситуации. Другим преимуществом статических анализаторов является диагностика места ошибки. При выдаче предупреждения об ошибке статические анализаторы сразу показывают место ошибки и описывают, почему именно это является ошибкой.

В данной статье будут рассмотрены различные подходы для поиска ошибок с помощью статического анализа. Описание статических анализаторов будет

приведено на примере инструмента Svace, разрабатываемого в ИСП РАН. В настоящее время инструмент содержит несколько видов анализаторов различной сложности и способен осуществлять поиск ошибок в программах, написанных на языках C, C++, Java и C#. Svace осуществляет поиск дефектов различных типов, включая ошибки разыменования нулевого указателя, переполнение буфера, использование неинициализированных переменных, утечки памяти и других ресурсов, двойные блокировки и тупики, наличие недостижимого кода, несогласованность конструкторов и деструкторов, ошибки деления на ноль, возвращение адреса локальных переменных, использование объектов после удаления и другие.

Примеры разных видов анализаторов будут иллюстрироваться ошибками в этих проектах, найденными с помощью Svace. С помощью Svace был проанализирован исходный код операционной систем tizen и android, а также 30 проектов, написанных на языках Си и Си++ с открытым исходным кодом. Большинство проанализированных проектов существуют уже много лет, их регулярно тестируют, проверяют с помощью статических и динамических анализаторов и исправляют найденные ошибки. Тем не менее с помощью Svace удалось обнаружить дефекты в исходном коде этих проектов.

2. Использование статического анализатора в жизненном цикле разработки программ

Статический анализатор осуществляет поиск ошибок в программе без её фактического запуска. После анализа выдаётся список предупреждений о дефектах в исходном коде программы. Предупреждение называют истинным, если оно соответствует дефекту в исходном коде, т.е. описывает ситуацию, которую желательно исправить. Если предупреждение не описывает некоторый дефект, то его называют ложным.

Идеальный анализатор после анализа программы за приемлемое время найдёт все дефекты в программе и не выдаст ни одного ложного срабатывания. К сожалению, создание такого анализатора невозможно, что следует из теоремы Райса, согласно которой для любого нетривиального свойства, определение того, вычисляет ли произвольный алгоритм функцию с таким свойством, является алгоритмически неразрешимой проблемой. Свойство считается нетривиальным, если существуют функции обладающие этим свойством и существуют функции не обладающие этим свойством. Поэтому проблема поиска дефектов в произвольной программе является алгоритмически неразрешимой задачей, т.к. содержит ли программа некоторую ошибку является нетривиальным свойством. Из-за проблемы неразрешимости для создания алгоритма, осуществляющего поиск ошибок, необходимо пожертвовать возможностью поиска всех ошибок, либо выдачей только истинных срабатываний, либо ограничить класс анализируемых программ. На практике также важной характеристикой любого алгоритма является время его работы. Реализация конкретного анализатора является компромиссом

между количеством выдаваемых истинных предупреждений, количеством ложных предупреждений и временем анализа.

Часть инструментов статического анализа используются для проверки программного обеспечения с высокими требованиями к надежности и безопасности. Для таких анализаторов важным требованием является поиск всех потенциальных ошибок. Такие анализаторы могут выдавать множество ложных срабатываний, а также принимать в качестве входа не все возможные программы.

Если анализатор выдаёт предупреждения со слишком большим количеством ложных предупреждений, то его никто не будет использовать. В результате даже истинные предупреждения не будут просмотрены, т.к. необходимо затратить много времени на фильтрацию ложных предупреждений. Более того, если первые 3 просмотренные предупреждения оказываются ложными, то пользователи часто вообще отказываются просматривать остальные предупреждения [5]. На практике более важным является не поиск всех дефектов, а исправление как можно большего количества дефектов, а также стоимость исправления каждого дефекта. Стоимость будет пропорциональна количеству ложных срабатываний, которые необходимо просмотреть на каждое истинное срабатывание. Поэтому важным требованием к анализаторам является небольшая доля выданных ложных срабатываний.

Использовании статического анализатора в жизненном цикле разработки программ имеет свои особенности. Как правило, нет задачи найти все возможные дефекты в программе. Анализ проводится регулярно во время ночной сборки программы, поэтому время анализа ограничено примерно 12 часами. Количество ложных срабатываний должно быть достаточно низким, т.к. каждое ложное срабатывание отнимает время программиста и фактически выражается в убытках для компании. Для удовлетворения этих требований анализатор может в случае, когда нет достаточных оснований, что предупреждение будет истинным, просто не выдавать предупреждение. Это позволяет существенно уменьшить количество выдаваемых ложных срабатываний и в ряде случаев сократить время анализа. Такой анализатор может пропустить реальную ошибку, но, т.к. большинство выдаваемых предупреждений будут истинными, то увеличивается шанс, что каждый реально найденный дефект будет исправлен, и уменьшается стоимость каждого исправленного дефекта.

Долю истинных срабатываний в 60-70% можно считать приемлемой. При таком соотношении истинных и ложных срабатываний на одно истинное срабатывание приходится не более одного ложного и пользователь не тратит много времени на просмотр предупреждений. Естественно, более высокие показатели в 80-90% для истинных срабатываний предпочтительнее, но только в том случае, если при этом не происходит потери количества найденных дефектов.

Описанный выше подход используется в инструменте Svace. Целью анализа является поиск как можно большего количества дефектов при приемлемом количестве ложных срабатываний и времени анализа. Как упоминалось выше, инструмент Svace выдал истинные предупреждения для проектов, для которых уже проводилось тестирование и осуществлялся поиск ошибок с помощью других статических анализаторов. Тестирование не покрывает все возможные пути в программе, поэтому после проведения тестирования, запуск статических анализаторов может выявить необнаруженные ошибки. Среди причин, почему найденные ошибки не были исправлены после прогона других статических анализаторов, можно выделить 2:

- Другие анализаторы также не осуществляют поиск всех возможных дефектов и используют другие методы поиска дефектов и отсеивания ложных срабатываний.
- Анализаторы нашли все возможные дефекты, но соответствующие предупреждения либо не были просмотрены, либо были по ошибке помечены как ложные. Т.е. поиск всех дефектов не гарантирует, что они все будут исправлены.

Важно отличать дефекты в исходном коде и дефекты во время выполнения программы. Не все дефекты в исходном коде могут привести к проблемам во время выполнения программы, но они могут свидетельствовать о наличии ошибок в логике реализации алгоритмов, либо затруднять чтение кода и поддержку программы. Одним из таких примеров является наличие переменных, которые были инициализированы, но нигде не используются. Такие переменные не могут привести к падению программы, либо к другим дефектам, но являются признаком плохого кода, и возможно являются опечатками и свидетельствуют о наличии других проблем.

Даже если некоторая функция написана, но нигде не вызывается, желательным является поиск дефектов в её коде. Возможно, программист написал функцию, но не успел написать код её использующий. Если в функции есть дефект, то лучше, если на следующий день программист получит сообщение о дефекте в новой функции.

3. Классификация методов статического анализа

Многие подходы статического поиска ошибок исторически развились из области компиляции программ и оперируют абстракциями, взятыми оттуда: поток токенов, абстрактное синтаксического дерево, граф вызовов, граф потока управления, поток данных.

По типу используемых абстракций методы поиска ошибок можно разделить на следующие группы:

- Лексические анализаторы, рассматривающие программу как поток токенов. С их помощью можно найти только самые простейшие виды дефектов и в работе они рассматриваться не будут.

- Легковесные анализаторы (анализаторы 1го уровня), осуществляющие анализ преимущественно с помощью просмотра абстрактного синтаксического дерева (АСД), а также с использованием других абстракций уровня синтаксического анализа.
- Более сложные анализаторы (анализаторы 2го уровня), которые используют абстракции, связанные с фазами после синтаксического анализа.

В результате работы синтаксического анализатора компилятором строится абстрактное синтаксического дерево, которое позже передаётся на следующие фазы компиляции программы. Анализаторы, построенные на базе АСД, осуществляют проход по узлам АСД и делают относительно простые проверки анализируемых правил. Время работы таких анализаторов линейно зависит от размера программы.

В группе легковесных анализаторов можно выделить более сложные анализы, использующие некоторую модель памяти [6]. Такие анализаторы позволяют реализовать довольно сложные детекторы и найти ошибки, которые невозможно найти простым проходом по АСД. Но не имеет смысла делать их сложнее, чем анализаторы 2го уровня, т.к. более сложная реализация потребует абстракций, которые не доступны на фазе синтаксического анализа, но будут доступны позже. Такая реализация не будет иметь преимуществ перед детекторами, использующими следующие фазы компиляции.

На втором уровне может быть выполнено значительное количество анализов. Здесь доступна информация об алиасах в программе, о константах, возможных значениях переменных, графе вызовов и др. Методы анализа 2го уровня также можно разделить по тому какие свойства они учитывают при анализе: чувствительность к потоку, межпроцедурность, чувствительность к контексту и др.

Потоково-нечувствительные анализы рассматривают программу как неотсортированный набор инструкций. Часть дефектов в программах не зависят от порядка выполнения инструкций и их реализация с помощью потоково-нечувствительного анализа будет более эффективной. Потоково-чувствительный анализ учитывает порядок инструкций. Такие анализы по-разному отвечают на вопросы о свойствах программы в зависимости от точки программы. Такой анализ в среднем требует больше памяти на хранение информации для каждой точки программы. При эффективной реализации требования к памяти не растут линейно с ростом размера программы, т.к. большая часть информации о свойствах программы разделяется между различными точками программы.

Потоково-чувствительные анализы в свою очередь можно разделить на анализы с чувствительностью к путям и без чувствительности к путям. Анализы с чувствительностью к путям учитывают по какому пути прошло выполнение программы. Не все пути в программе могут быть выполнены, т.к. будут зависеть от противоречивых условий. Для решения задачи

выполнимости путей такие анализы часто используют SMT-решители, отсеивающие заведомо невыполнимые пути.

Многие дефекты являются результатом неправильного использования нескольких функций. Такие дефекты можно обнаружить только если проводить межпроцедурный анализ. Будем считать, что межпроцедурный анализ имеет контекстную чувствительность, если анализ отличает эффекты вызова функции в зависимости от контекста её вызова.

Учёт какого-либо свойства требует большего количества памяти и времени работы анализа, но зато даёт лучшую точность. Недостаточная точность анализа приводит либо к пропуску ошибок, либо к появлению ложных срабатываний. Будем считать более сложным анализ, который учитывает больше свойств. При этом анализы могут иметь разную степень чувствительности к какому-либо свойству. Например, при чувствительности к контексту вызова можно учитывать значения не всех переменных или анализировать разную высоту стека вызовов.

Критичность найденных ошибок, как правило, не зависит от сложности анализатора. Многие критичные ошибки являются результатом опечатки и часто могут быть найдены с помощью анализаторов 1го уровня. Более простые правила желательно реализовывать с помощью легковесных анализаторов на основе АСД. В этом случае и скорость написания правила и скорость анализа будет лучше, чем если бы оно реализовывалось в более тяжеловесном анализаторе. Кроме того, часть информации, которая доступна в АСД, может быть недоступной на более поздних фазах анализа. Например, наличие или отсутствие фигурных скобок можно проверить только на этапе синтаксического анализа, т.к. далее эта информация не сохраняется.

Для каждого конкретного правила желательно выбрать наиболее подходящую сложность используемых анализаторов. Если использовать более сложные анализаторы для поиска всех дефектов, то общее время работы инструмента увеличится. Разгрузив более сложные анализы от поиска дефектов, которые могут быть найдены более лёгкими анализами, можно увеличить скорость работы, а во многих случаях и скорость разработки детекторов.

Из-за проблемы неразрешимости задачи поиска дефектов в исходном коде программ, любой статический анализатор находит приближённое решение. Анализ будем считать консервативным, если приближённое решение гарантированно включает все возможные варианты при выполнении программы. Консервативность гарантирует, что все полученные выводы являются корректными. Это очень важно для оптимизирующих компиляторов, которые не должны изменять семантику программы. Если нет достаточно уверенности, что оптимизация безопасна, то лучше вообще её не применять, чем изменить поведение программы. При поиске же ошибок, иногда желательно выдать предупреждение об ошибке, которое может оказаться ложным, либо основано на некоторых эвристиках. Использование неконсервативного анализа позволяет создать более простой и быстрый

анализатор, и в тоже время находить довольно сложные типы ошибок. Расплатой за это является недостаточно точная модель программы, что в некоторых случаях может приводить к непонятным ложным срабатываниям.

На рис. 1 показан фрагмент кода, где осуществляется запись в переменную x через указатель p. В конце функции происходит считывание значения переменной в переменную i, которая используется для доступа к массиву размера 10. Таким образом может произойти переполнение массива, если значение i равно 20. Оптимизирующий компилятор в данном случае не может сделать такой вывод, т.к. возможно на эту область памяти указывает другой указатель, который обновляет память при вызове функции func. В данном случае ничего не известно про функцию func, она может менять значение переменной x, причём переменная может меняться при любых условиях, либо только при некоторых условиях. Будет ли в примере ошибка, зависит от реализации функции func. Человек, использующий статический анализатор, скорее всего, хотел бы, чтобы данная ошибка была выдана. Исключением можно назвать ситуацию, когда функция func тривиальна и перезаписывает значение x на безопасное.

```
char buf[10];
int* p = &x;
*p = 20;
func();
int i = *p;
buf[i] = 0;
```

Рис. 1. Потенциальное переполнение буфера.

Подчеркнём, что проблема не в том, что неизвестна реализация функции func, а в том, что в общем случае невозможно определить, что именно она будет делать. Даже если доступен исходный код функции, не всегда можно определить при каких условиях функция выполняет некоторые действия. К примеру, в функции есть условие, описываемое некоторой недоказанной математической теоремой. Если теорема верна, то условие будет тривиально истинным.

4. Краткое описание Svace

Инструмент Svace, разработанный в Институте системного программирования РАН, осуществляет статический поиск дефектов, используя несколько видов анализаторов. Текущая версия Svace[1] для анализа Си и Си++ программ

[1] В статье описывается только часть инструмента Svace для поиска ошибок в программах на языках Си и Си++.

содержит статические анализаторы двух уровней: легковесный анализатор и основной анализатор. Легковесный анализатор, интегрированный в компилятор Clang, осуществляющий поиск дефектов просматривая абстрактное синтаксическое дерево. Более сложный анализатор, осуществляет межпроцедурный потоково- и контекстно-чувствительный анализ.

Основной анализатор осуществляет межпроцедурный анализ путём обхода графа вызовов «снизу-вверх», начиная с листьев графа таким образом, чтобы вызываемые функции анализировались до вызывающих. Каждая функция обходится только один раз, вся необходимая в дальнейшем информация помещается в специальную структуру данных, называемую аннотацией. Такой подход позволяет анализировать каждую функцию только один раз, что существенно улучшает скорость анализа и масштабируемость. При таком подходе естественным образом получается чувствительность к контексту, т.к. каждая аннотация применяется независимо для каждого контекста вызова. Недостатком такого подхода является необходимость хранить все данные, которые могут потребоваться: если какая-то информация не будет сохранена, то в дальнейшем получить её будет уже невозможно.

При анализе каждой функции создаётся граф потока управления и производится его топологическая сортировка. Затем осуществляется анализ 2х видов: консервативный анализ потока данных и неконсервативный анализ, использующий сложную модель памяти. Оба вида анализов осуществляют как прямой, так и обратный анализы. При этом прямые анализы осуществляют прямой топологический обход графа потока управления и обычно отвечают на вопрос, что произошло или могло произойти с переменной. Обратные анализаторы выполняют обратный обход и отвечают на вопрос, что произойдёт или может произойти с переменной. Например, классический анализ живых переменных является обратным анализом и отвечает на вопрос может ли значение переменной быть использовано.

Неконсервативный анализ имеет поддержку чувствительности к путям, контексту и поддерживает межпроцедурный анализ. Ядро анализатора поддерживает возможность проводить анализ разной степени сложности, а использование этих возможностей зависит от конкретного детектора. Благодаря этому для каждого детектора можно выбрать необходимую степень учитываемых свойств. Основное неконсервативное предположение, которое используется в этом анализе - отсутствие алиасов среди входных параметров функции и глобальных переменных. Как правило, это условие выполняется, т.к. при наличии большого количества алиасов программисту самому сложно понимать семантику функции.

В конце анализа запускается парный анализ конструкторов и деструкторов, осуществляющий поиск несогласованностей при их реализации. Описание этой фазы приведено в главе 10.

Описание Svace также можно найти в [7-9].

Инструмент Svace имеет приемлемые характеристики истинных срабатываний и время анализа. По данным наших заказчиков стабильные детекторы Svace, реализованные в основном анализаторе для Си и Си++, имеют долю истинных срабатываний от 69 до 95%. Детекторы из легковесного анализатора имеют лучший процент истинных срабатываний.

Для оценки времени анализа необходимо в общее время также включать время сборки проекта. Общее время для сборки и анализа должно позволять проводить анализ во время ночной сборки. Время анализа для небольших проектов (busybox, cairo, dnprogs) составляет около 5 минут на обычном компьютере. Время сборки сравнимо. Анализ операционной системы android-5.0.2 на сервере с 256Гб оперативной памяти занимает около 5 часов. Сборка андроида занимает 2.5 часа. Таким образом общее время, необходимое на сборку и анализ андроида составляет 7.5 часов, что является приемлемым для ежедневного анализа во время ночной сборки проекта.

5. Анализатор на основе АСД

Многие виды дефектов могут быть обнаружены только на стадии синтаксического анализа, т.к. необходимая информация не передаётся на следующие стадии. Такие детекторы осуществляют просмотр АСД и проверяют его соответствие некоторым правилам. На этой фазе не доступна информация об алиасах программы, графе потока управления, графе вызовов, и не производится межпроцедурный анализ. Поэтому сложные дефекты не могут быть обнаружены на этой фазе. Но, т.к. критичность ошибок не зависит от сложности используемого анализа, а также то, что выданные предупреждения имеют высокий процент истинных срабатываний из-за относительно простой реализации, анализ на основе АСД позволяет находить множество дефектов в программах.

Детектор NO_EFFECT.SELF_ASSIGN проверяет аргументы для узла присваивания в АСД. Если аргументы соответствуют одному и тому же символу, то выдаётся предупреждение. Детектор способен найти ошибки вида: «x = x» или «p->q = p->q». Но так как сравнение происходит только для символов и отсутствует информация об алиасах, то ошибка не будет выдана для следующего кода:

```
x = y;
y = x;//здесь переменная x уже имеет значение y.
```

Пример предупреждения NO_EFFECT.SELF_ASSIGN, выданное для проекта nss-3.12.9+ckbi-1.82 приведён на рис. 2. Строка 921 содержит присваивание для одной и той же переменной. Скорее всего ошибка является результатом опечатки.

```
913| if (rawptr >= end) {
 pubk->u.fortezza.DSSKey.len = pubk->u.fortezza.KEAKey.len;
```

```
 pubk->u.fortezza.DSSKey.data=
          pubk->u.fortezza.KEAKey.data;
921|   pubk->u.fortezza.DSSprivilege.data =
          pubk->u.fortezza.DSSprivilege.data;
goto done; }
```

Рис. 2. Пример срабатывания NO_EFFECT.SELF_ASSIGN.

6. Консервативный анализ потока данных

Консервативный анализ используется главным образом для сбора вспомогательных данных о функции и переменных: анализ живых переменных, недостижимого кода, функций, завершающих выполнение программы. Результаты этого анализа используется неконсервативным анализом. Этот вид анализа имеет чувствительность к потоку и ограниченную межпроцедурность (распространяются только данные о функциях, завершающих программу). Чувствительность к путям и контексту не реализована, чтобы не слишком замедлять эту фазу.

В консервативном анализаторе реализован детектор поиска недостижимого кода. Анализ недостижимого кода используется далее в основном анализаторе svace, чтобы исключить влияние на анализ инструкций, которые недостижимы. Анализатор реализован в консервативной фазе, т.к. этот анализ может существенно влиять на работу других анализаторов.

Обычно наличие недостижимого кода не влияет на производительность программы, оптимизирующий компилятор всё равно удалит этот код из программы. Но наличие такого кода может свидетельствовать об опечатках, либо неправильном понимании программы. Программисты редко пишут код, который никогда не может быть выполнен.

Предупреждение UNREACHABLE_CODE выдаётся для участков кода, которые не могут быть достигнуты при выполнении программы. Написано 4 вида анализов потока данных для поиска недостижимого кода: на основе интервалов, ненулевых значений, простых предикатов и функций, завершающих программу. Анализ на основе интервалов для переменных программы сопоставляет интервал возможных целочисленных значений [a; b], что означает, что во время выполнения все возможные значения переменной принадлежат этому интервалу. После завершения анализа осуществляется проход по графу потока управления и для всех условных инструкций проверяются возможные интервалы значений. Если одна из веток условной инструкции при этом не может быть выполнена, то выдаётся предупреждение.

С помощью интервала нельзя описать интервал с выколотой точкой, и в частности ситуацию, что некоторая переменная имеет любое ненулевое значение. Так как сравнение с нулём частая операция, используемая в Си, то был реализован анализ ненулевых значений, который для каждой переменной проверяет, что переменная точно имеет ненулевое значение. После чего также

выполняется проход по графу потока управления. На рис. 3 показан пример предупреждения, найденный описанным анализом для проекта tizen/external-swig/allegrocl.cxx:2699. В примере содержится лишнее сравнение указателя tm с нулём, поэтому код на else-ветке будет недостижимым. После первого условия анализ пометит переменную tm, как имеющую ненулевое значение, далее при анализе второго условия, анализатор выдаст предупреждение, т.к. tm не может быть нулём и сравнивается с нулём. Эта ошибка не обязательно является безобидной лишней проверкой на ноль, которую легко выкинет компилятор. В данном случае, если tm равно нулю, функция не завершит программу, а продолжит выполнение. При этом в отладочном логе не будет содержаться запись о дефекте.

```
if (!is_void_return && tm) {
   if (tm) {
     Replaceall(tm, "$result", "lresult");
     Printf(f->code, "%s\n", tm);
     Printf(f->code, "        return lresult;\n");
     Delete(tm);
   } else {
   Swig_warning(WARN_TYPEMAP_OUT_UNDEF, input_file, line_number,
         "Unable to use return type %s in function %s.\n",
         SwigType_str(t, 0), name);
}}
```

Рис. 3. Недостижимый код

Анализ предикатов в качестве свойств потока данных использует конъюнкции простых предикатов вида “a op b”, где a и b - переменные, либо константы, a op - следующие операции: ==, !=, >, <, >=, <=. Если для некоторой точки оказывается, что предикат тривиально равен лжи, значит данная точка недостижима. Предыдущий пример также может быть найден этим анализом, т.к. в точке вызова функции Swig_warning предикат имеет вид “is void_return==0 & tm!=0 & tm==0”.

Последний вид анализов осуществляет анализ функций, которые завершают выполнение программы, и распространяет вдоль графа потока управления свойство “завершающая функция точно была вызвана в данной точке”. Если это свойство истинно, то код не может быть выполнен.

7. Потоково-чувствительный анализ

Основной анализатор Svace является неконсервативным потоково-чувствительным. При необходимости отдельные детекторы могут использовать общую инфраструктуру для межпроцедурного анализа и анализа с чувствительностью к путям.

Многие ошибки, встречающиеся в коде, являются локальными и часто вообще находятся на одной строке, поэтому имеет смысл реализовать простые детекторы для поиска таких ошибок. В Svace реализовано несколько относительно простых детекторов, осуществляющих поиск ошибок разыменования нулевых указателей. Каждый детектор осуществляет поиск некоторого конкретного вида ошибки. Преимуществом использования таких детекторов помимо скорости работы и небольшого использования памяти, является простота их реализации и, следовательно, минимизация ошибки при реализации детектора.

Каждая функция программы анализируется сама по себе без конкретного контекста, где она может быть вызвана. Серьёзной проблемой при этом является то, что некоторые функции накладывают определённые требования к контексту использования, и не могут быть вызваны в произвольном контексте. Примером такой функции является функция стандартной библиотеки Си memcpy, осуществляющая копирование из одной области памяти в другую. Функция имеет 3 параметра: 2 указателя на области памяти и количество копируемых байт. Если количество копируемых байт не нулевое, то 2 указателя не могут иметь нулевые значения. Если вызвать функцию memcpy с параметрами (0, 0, 10), то произойдёт ошибка разыменования нулевого указателя. Но это будет ошибка вызывающего кода, а не кода memcpy. Поэтому нельзя выдавать предупреждение каждый раз, когда происходит разыменование указателя без проверки на нулевое значение без информации о том, в каком контексте функция может быть вызвана.

Сформулируем следующий принцип написания функции: «каждая инструкция функции должна быть достижима хотя бы для одного потенциального контекста вызова». Потенциальный контекст вызова не обязательно присутствует в анализируемой программе. Если инструкция недостижима ни из одного потенциального контекста вызова, то возникает вопрос, зачем этот код вообще писали. Данный принцип позволяет находить большое количество дефектов при анализе функций самих о себе. Предупреждение выдаётся, если все пути, проходящие через некоторую инструкцию, содержат ошибку. Если инструкция достижима во время выполнения программы, то предупреждение соответствует дефекту во время выполнения программы. Если инструкция не достижима, то это является дефектом исходного кода программы. Таким образом, если из гипотезы о достижимости инструкции следует наличие в коде ошибки, то выдаётся предупреждение, если гипотеза не верна, то в теле функции находится инструкция, которая ни при каких условиях не будет исполнена, что само по себе является дефектом в исходном коде.

Предупреждение DEREF_OF_NULL описывает ситуации, когда указатель сравнивается с нулём и затем разыменовывается, в точке разыменования указатель может иметь только нулевое значение. Ошибка выглядит довольно глупой, но тем не менее, встречается в реальных проектах. Детектор ассоциирует с переменными свойство «переменная точно была положительно

сравнена с нулём» и распространяет это свойство по графу потока управления. На рис. 4 показаны примеры ошибок, которые могут быть найдены этим детектором:

```
if(!p) {
  *p = 0;//p может иметь только нулевое значение
}
if(q) {
  exit(0);
}
*q = 0;//q может иметь только нулевое значение
```

Рис. 4. Паттерны детектора DEREF_OF_NULL

Рис. 5 содержит пример подобной ошибки из того же проекта tizen/framework-uifw. Ошибка довольно тривиальна. Место разыменования находится всего через 2 строки от проверки на ноль. Не стоит недооценивать такие предупреждения. Если по какой-то причине функция NextIndicatorName вернёт нулевой указатель, то пользователь вместо сообщения об ошибке получит падение программы. Для поиска таких ошибок нет смысла реализовывать сложный детектор, имеющий чувствительность к путям или контексту.

```
260| old = new
261| new = NextIndicatorName(info);
262| if (!new)
263| {
264|   WSGO1("Couldn't allocate name for %d\n", new->ndx);
265|       ACTION("Ignored\n");
266|       return False;
267| }
```

Рис. 5. Разыменование нулевого указателя

Детектор DEREF_AFTER_NULL находит ситуации, в которых указатель сравнивается с нулём, а позже разыменовывается. В отличии от DEREF_OF_NULL в точке разыменования указатель не обязательно имеет нулевое значение. Детектор использует вышеописанный принцип, что каждая инструкция должна быть достижима хотя бы для некоторого контекста вызова. Детектор находит ошибки для следующего паттерна:

```
if(p) { /*..*/}
*p = 0;//здесь p, может иметь нулевое значение. Точка разыменования постдоминирует над сравнением.
```

Т.к. в точке разыменования указатель не обязательно имеет нулевое значение, то для поиска таких ошибок нужна чувствительность к путям, чтобы

проверить, что такой путь может существовать. Детектор DEREF_AFTER_NULL не использует чувствительность к путям, и чтобы не выдавать ложные срабатывания, связанные с тем, что путь не существует, детектор выдаёт предупреждение только, если точка разыменования постдоминирует над точкой сравнения, т.е. если после положительного сравнения с нулём указатель обязательно будет разыменован. Для реализации детектора используется прямой анализ, распространяющий свойство, что указатель был положительно сравнен с нулём, и обратный анализ для свойства, что указатель будет разыменован.

Большое количество найденных ошибок связано с неправильным использованием конъюнкций и дизъюнкций, их часто путают друг с другом. Наиболее типичной является ситуация, где с помощью конъюнкции проверяют более узкое условие P && E, вместо проверки E.

Пример ошибки, найденный детектором для tizen/external-eglibc показан на рис. 6. Если указатель namehashent имеет нулевое значение, а переменная replace имеет значение true, то не произойдёт выход из функции, и указатель будет разыменован. По всей видимости проверка значения replace здесь лишняя, возможно “&&” надо заменить на “||”.

```
   struct namehashent *namehashent = insert_name (ah, alias, strlen (alias), replace);
720:  if (namehashent == NULL && ! replace)
   return;
722:  if (namehashent->name_offset == 0) {
```

Рис. 6. Разыменование указателя после сравнения с нулём.

8. Чувствительность к путям

Анализ с чувствительностью к путям учитывает предикаты в инструкциях ветвления и способен отсеять несуществующие пути, зависящие от предикатов, которые не могут одновременно выполняться. Как правило такие анализы используют SMT-решатели для определения выполнимости предикатов.

Проблему зависимых предикатов можно проиллюстрировать с помощью паттерна «двойного ромба». Паттерн получил такое название, т.к. в графе потока управления выглядит как два последовательных ромба. На рис. 7 показан пример кода, имеющего 2 условия и 4 возможных пути выполнения. Если не рассматривать условия от которых зависит выполнения программы, то нельзя определить содержит ли данный код ошибку разыменования нулевого указателя Поэтому анализатору без чувствительности к путям остаётся либо выдавать предупреждение об ошибке, что приведёт к ложным срабатываниям во многих случаях, либо не выдавать, т.к. недостаточно данных. При использовании более сложных алгоритмов можно определить, что разыменование нулевого указателя произойдёт только если (!(a > b) &&

(a>b+1)), и с помощью SMT-решателя определить, что данная формула не имеет решений.

```
int g;
void func(int a, int b) {
      int*p = 0;
       if(a > b) { p = &g; }
       if(a > b + 1) {*p  = 6;}
}
```

Рис. 7. «Двойной ромб»

Даже при запоминании всех условий и использовании SMT-решателей не всегда возможно сказать может ли существовать некоторый путь или нет, что является следствием проблемы неразрешимости. Например, многие анализируемые условия зависят от решения задачи алиасов и без её решения консервативно можно только сказать, что условия могут быть зависимыми. Существующие лучшие алгоритмы анализа указателей, как правило, потоково-нечувствительные и часто имеют не лучшую скорость работы. Анализ Андерсона является одним из наиболее точных потоково-нечувствительных алгоритмов, но при этом алгоритм имеет кубическую сложность анализа [10] и поэтому плохо масштабируется для больших программ. Поэтому даже при использовании SMT-решателей может быть оправдан неконсервативный анализ и использование эвристик. К примеру можно считать, что значения глобальных переменных не являются алиасами входных параметров и не могут быть косвенно изменены другими функциями. Подобные допущения позволяют существенно ускорить анализ, и обычно не сильно ухудшают результаты анализа.

В Svace реализовано несколько детекторов, имеющих чувствительность к путям. DEREF_AFTER_NULL.EX - версия DEREF_AFTER_NULL с чувствительностью к путям. Детектор DEREF_AFTER_NULL из-за отсутствия анализа выполнимости условий на путях выдаёт предупреждение только, если на всех путях будет разыменование, благодаря чему удаётся избежать ложных срабатываний, связанных с отсутствием чувствительности к путям. Предупреждение DEREF_AFTER_NULL.EX выдаётся, если переменная сравнивается с нулём, а затем на некотором пути разыменовывается. Для поиска ошибки Svace осуществляет проход по графу потока управления и для каждой точки разыменования собирает необходимые условия достижимости точки разыменования. Затем начиная с каждой точки сравнения указателя с нулём Svace осуществляет проход по графу потока управления и собирает условия, при которых можно попасть из точки сравнения во все остальные точки. Затем для инструкции разыменования проверяется все условия с помощью SMT-решателя Z3[11]. Если условия могут быть выполнимы, значит может существовать путь, где указатель сначала сравнивается с нулём, а затем разыменовывается, в этом случае выдаётся предупреждение.

Пример предупреждения для проекта tizen-2.3 (gdhcp/client.c:2124). На строке 2096 переменная “message_type” сравнивается с нулём, но выход из функции произойдёт только если “client_id” тоже имеет нулевое значение. В противном случае если dhcp_client->state имеет одно из значений REQUESTING, RENEWING, REBINDING, то произойдёт разыменование на строке 2124. Детектор DEREF_AFTER_NULL не может обнаружить подобную ошибку, т.к. разыменование происходит не на всех путях, а для путей, на которых происходит разыменование детектор не может определить их выполнимость. Разыменование произойдёт, если “client_id != 0 && dhcp_client->state==REQUESTING”, но для детектора без чувствительности к путям это условие не отличимо от условия “client_id != 0 && message_type != 0” и от условия “client_id != 0 && client_id == 0”, при которых путь будет невыполним.

```
2096|   if (message_type == NULL && client_id == NULL)
2098|       return TRUE;
2100|   debug(dhcp_client, "received DHCP packet xid 0x%04x "
2101|           "(current state %d)", xid, dhcp_client->state);
2102|
2103|   switch (dhcp_client->state) {
2121|   case REQUESTING:
2122|   case RENEWING:
2123|   case REBINDING:
2124|
      if (*message_type == DHCPACK) {
2125|           dhcp_client->retry_times = 0;
```

Рис. 8. Разыменование нулевого указателя на одном из путей.

9. Межпроцедурный анализ

Часть ошибок является результатом неправильного использования функций, либо неправильной реализации функций. Поиск таких ошибок представляет сложность как для человека, так и для статического анализатора. Вручную найти такие ошибки сложно из-за того, что код, который вызывает ошибку разнесён в разные функции или даже в разные файлы. Статическому анализатору необходимо учитывать межпроцедурные эффекты функций. Инструмент Svace осуществляет межпроцедурный анализ на основе аннотаций. При таком подходе после анализа функции создаётся её аннотация, описывающая интересующие эффекты вызова функции, а затем аннотация транслируется в вызывающий контекст, эмулируя вызов функции. Создание аннотаций требует процессорное время, а их хранение занимает место в оперативной памяти. Аннотации можно сохранять на диск, чтобы снизить требования к оперативной памяти, но сохранение и чтение с диска

замедляет анализ. Поэтому межпроцедурные детекторы потребляют больше памяти и процессорного времени по сравнению с внутрипроцедурными.

Аннотации, используемые в Svace, основаны на атрибутах, описывающих интересующие свойства. При этом атрибут может использоваться разными детекторами, если их интересует одно и тоже свойство. Выше описывались детекторы DEREF_OF_NULL, DEREF_AFTER_NULL, NULL_AFTER_DEREF, осуществляющие поиск разыменования нулевого указателя. Всем этим детекторам интересно свойство, что указатель разыменовывается в некоторой функции, для этого свойства создаётся атрибут, который записывается в аннотацию один раз и используется всеми этими детекторами, что экономит память по сравнению с версией, если бы каждый детектор самостоятельно писал в аннотацию.

При создании аннотации необходимо решить, что именно сохранять в аннотации. Чем больше информации будет сохранено, тем более медленным будет анализ. Чем меньше информации будет сохранено, тем меньше дефектов будет найдено. В Svace используется 2 подхода: запомнить свойство, которое верно для всех вызовов функций; запомнить при каких именно условиях происходит некоторое событие.

Если некоторое событие происходит при любом вызове некоторой функции, то для его сохранения не требуется много памяти. В минимальном случае достаточно использовать один бит - произошло событие или нет. На практике при сообщении об ошибке пользователю необходимо показать почему именно здесь ошибка, поэтому требуется также сохранить трассу, содержащую точки в программе для описываемого события. Для детекторов DEREF_OF_NULL, DEREF_AFTER_NULL, NULL_AFTER_DEREF это событие - разыменование. Если некоторый указатель разыменовывается на всех возможных путях, проходящих через некоторую функцию, то в аннотацию добавляется атрибут содержащий трассу разыменования - последовательность точек вызовов функций, завершающуюся точкой разыменования. Если есть путь, на котором нет разыменования указателя, то в аннотацию ничего не добавляется и предупреждение не будет выдаваться. Межпроцедурные версии этих детекторов находят те же самые дефекты, что и внутрипроцедурные, но способны выдать предупреждение, о разыменовании, происходящем в вызываемой функции.

Пример межпроцедурного дефекта проекта xorg-server-1.7.6 показан на рис. 9. Сравнение с нулём находится на 232 строке, разыменование же происходит внутри функции __glXDrawableRelease, которая вызвана на строке 242. Рис. 9 содержит код этой функции, где на строке 1147 происходит разыменование входного аргумента. В данном примере уже сложно разобраться что происходит, по этой причине помимо сообщения об ошибке, Svace дополнительно выдаёт трассы, показывающие, почему произошло разыменование. Предупреждение содержит 2 трассы: одну для точки сравнения с нулём, и одну для разыменования, включающую в себя 2 точки

(glxdri.c:242, где происходит вызов функции, и glxcmds.c:1147, где происходит разыменование).

```
232|    if (drawable->pDraw != NULL) {
233|      screen = (__GLXDRIscreen *) glxGetScreen(drawable->pDraw->pScreen);
234|   (*screen->core->destroyDrawable)(private->driDrawable);
236|      __glXenterServer(GL_FALSE);
237|      DRIDestroyDrawable(drawable->pDraw->pScreen,
238|                              serverClient, drawable->pDraw);
239|      __glXleaveServer(GL_FALSE);
240|    }
242| __glXDrawableRelease(drawable);
```

Код функции _glXDrawableRelease:

```
1145|void __glXDrawableRelease(__GLXdrawable *drawable)
1146|{
1147|    ScreenPtr pScreen = drawable->pDraw->pScreen;
1149|    switch (drawable->type) {
1150|    case GLX_DRAWABLE_PIXMAP:
1151|    case GLX_DRAWABLE_PBUFFER:
```

Рис. 9. Межпроцедурный дефект разыменования нулевого указателя

Наиболее сложный анализ осуществляется для поиска межпроцедурный ошибок в ситуациях, когда некоторое событие происходит в функции условно. Рассмотрим поиск разыменований нулевых указателей, где разыменование может произойти в функции при некоторых условиях. Необходимо запомнить условия, при которых вызываемая функция разыменовывает входные параметры, а также взаимосвязь всех переменных. Во многих случаях условия становятся слишком громоздкими и требующими много памяти для сохранения в аннотации. Если формула для условия становится слишком сложной, то Svace не сохраняет её в аннотации, и считается, что функция не разыменовывает свой аргумент. Для определения сложности формулы используются несколько параметров, включая высоту дерева разбора формулы, количество используемых атомарных формул, общее количество узлов в дереве разбора. Для увеличения количества формул, которые могут быть сохранены, используется несколько методов их упрощения. Во-первых, используются некоторые факты из логики высказываний, которые позволяют упростить формулы вида «a & (b & a)» в «a & b». Т.к. запись в аннотацию некоторой формулы означает, что если формула имеет решение, то произойдёт разыменование переменной, то корректно разрывать дизъюнкции и вместо формулы «a | b» сохранять «a». Формула «a» может быть проще формулы «a | b» и пройдёт требования на сложность формулы. При этом

произойдёт потеря точности, т.к. не будет сохранена информация, что если «b» истинно, то произойдёт разыменование, но это всё равно лучше, чем отсутствие любой информации из-за слишком сложной формулы.

При анализе вызова функции все формулы в аннотации необходимо оттранслировать в контекст вызова и переименовать все используемые переменные в соответствующие аналоги на стороне вызова. После этого для каждой переменной анализатор имеет условие в виде формулы, что переменная имеет нулевое значение перед вызовом функции, условие, что переменная будет разыменована в вызываемой функции, и также условие что инструкция вызова будет достигнута. Решателю Z3 передаётся конъюнкция всех 3х условий, и если он находит модель результирующей формулы, то вызывающая функция может разыменовать нулевой указатель.

10. Анализ конструкторов и деструкторов

В языке Си++ при создании объекта вызывается конструктор, осуществляющий его инициализацию. При удалении объекта будет вызван деструктор для очистки используемых ресурсов. Деструктор должен гарантировать, что все созданные в конструкторе ресурсы будут освобождены при уничтожении объекта. Несогласованность между конструкторами и деструкторами может приводить к утечке ресурсов, их двойному освобождению, либо освобождению не выделенных ресурсов.

Для поиска таких дефектов в конце анализа запускается фаза парного анализа конструкторов и деструкторов. Эта фаза была написана таким образом, чтобы по максимуму переиспользовать возможности основного анализатора Svace. Для этого в код деструктора в качестве первой инструкции вставляется вызов конструктора. Затем производится обычный анализ модифицированного кода, при обработке вызова конструктора используется его аннотация, также как и при межпроцедурном анализе. Различные детекторы при этом могут проверить несогласованность в конце анализа деструктора. Если у класса есть несколько конструкторов, то такой анализ проводится для каждого конструктора в отдельности. Также парный анализ осуществляется для операторов присваивания.

Детектор MEMORY_LEAK.CTOR выдаёт предупреждение, если в конструкторе выделяется память для некоторого члена класса, а в деструкторе отсутствует освобождение выделенной памяти. Для поиска ошибок в конце анализа модифицированного кода деструктора детектор запускает стандартный анализ утечек памяти, осуществляющий поиск достижимых ячеек памяти. Если в результате поиска находится утечка памяти, то это означает, что после вызова деструктора не все ресурсы будут освобождены. Пример ошибки для проекта android 5.0.2 (AST.cpp:612) приведён на рис. 10.

```
611| FinallyStatement::FinallyStatement()
612|         :statements(new StatementBlock) { }
  Код деструктора :
616| FinallyStatement::~FinallyStatement()
617|{ }
   Определение FinallyStatement (FrameworkListener.h) :
280| struct FinallyStatement : public Statement
281| {
282|    StatementBlock* statements;
```

Рис. 10. Несогласованность конструктора и деструктора.

11. Заключение

Как было показано, инструменты статического анализа позволяют находить ошибки в хорошо протестированных приложениях. Многие найденные ошибки связаны с обработкой ошибочных ситуаций и с редко исполняемыми путями. Для реализации подобных ошибок программа должна попасть в состояние, которое сложно смоделировать с помощью тестирования.

Каждое ложное срабатывание, выданное инструментом, выражается в деньгах, затрачиваемых на зарплату программистам, просматривающих сообщения об ошибках. Поэтому для практического поиска дефектов низкий уровень ложных срабатываний может быть даже важнее, чем общее количество найденных дефектов. Разные инструмента статического анализа используют различные алгоритмы поиска дефектов и способы борьбы с ложными срабатываниями. По этой причине даже после проверки программы статическим анализатором имеет смысл использовать другой анализатор для поиска дефектов.

В отличии от оптимизирующих компиляторов статическим анализаторам не обязательно выдавать консервативный результат. Во многих ситуациях лучше сообщить о проблеме даже, если нет достаточно уверенности в её наличии, что мотивирует использование неконсервативных анализаторов, использующих предположения, которые не обязательно верны для всех программ и функций.

Дефекты в программе имеют разную природу и для их поиска необходимо правильно выбрать алгоритм анализа. Хороший инструмент будет включать в себя как простые детекторы, использующие анализ на основе АСД, так и сложные детекторы, позволяющие найти нетривиальные межпроцедурные ошибки. Инструмент Svace состоит из набора анализаторов реализующих анализы разных типов: анализ на основе абстрактного синтаксического дерева, консервативный анализ потока данных для одной функции, потоко-чувствительный и межпроцедурный неконсервативный анализ с возможностью использовать чувствительность к путям. Построение инструмента на основе нескольких анализаторов позволяет использовать

преимущества этих видов анализаторов и находить больший диапазон ошибочных ситуаций.

Список литературы:

[1]. S. C. Misra, V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality //Computational Science and Its Applications—ICCSA 2003. – Springer Berlin Heidelberg, 2003. – С. 724-732.

[2]. https://msdn.microsoft.com/library/cc307416

[3]. M. Tim Jones. Static and dynamic testing in the software development life cycle. 26 August 2013 (http://www.ibm.com/developerworks/library/se-static/)

[4]. А. С. Марков, В. Л. Цирлов, А. В. Барабанов. Методы оценки несоответствия средств защиты информации //М.: Радио и связь. – 2012.

[5]. T. Kremenek, D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations //Static Analysis. – Springer Berlin Heidelberg, 2003. – С. 295-315.

[6]. В.Н. Игнатьев. Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования. Труды ИСП РАН, том 22, 2012, с. 169–188. DOI: 10.15514/ISPRAS-2012-22-11.

[7]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26. С. 231–250. DOI: DOI: 10.15514/ISPRAS-2014-26(1)-7.

[8]. А.И. Аветисян, А.Е. Бородин. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН, том 21, 2011, с. 39–54.

[9]. А.И. Аветисян, А.А. Белеванцев, А.Е. Бородин, В.С. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, 2011, с. 23–38.

[10]. M. Shapiro, S. Horwitz. Fast and accurate flow-insensitive points-to analysis //Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM, 1997. – С. 1-14.

[11]. L. De Moura, N. Bjørner. Z3: An efficient SMT solver //Tools and Algorithms for the Construction and Analysis of Systems. – Springer Berlin Heidelberg, 2008. – С. 337-340.

A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels

[1]*A. Borodin <alexey.borodin@ispras.ru>*
[1,2]*A. Belevancev <abel@ispras.ru>*
[1]*Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. The paper describes a practical approach for finding bugs in the source code of programs using static analysis. The approach is implemented in the Svace tool that is developed by ISP RAS. Svace performs defect detection for different error types including null pointer dereferences, buffer overruns and underruns, uninitialized variables usages, memory leaks, double locks and missing locks, unreachable code, division by zero, use after free and others.

The analysis goal is to find as many defects as possible while minimizing false positives with acceptable analysis time. As a result, on large programs the approach inevitably results in missing some defects.

Even critical program defects exist because of various reasons, and the right analysis approach should be detected based on a defect type. A good analyzer will include both simple detectors using only semantic analysis on an abstract syntax tree (AST) and complex detectors using interprocedural context and path sensitive analyzers. The Svace analyzer is designed for that purpose as a collection of analyzers having various levels: an AST analyzer, a conservative data flow analyzer, flow, context and path sensitive interprocedural analysis that makes a few assumptions losing conservativeness. The interprocedural analysis is annotation based: each function is analyzed only once, and its annotation created to summarize the analysis results is used when simulating this function's call. All described algorithms are presented and illustrated using examples of various detectors and their real warnings found on a number of open source projects.

Keywords: static analysis; C language; defects in source code; abstract syntax tree; flow-sensitivity; path-sensitivity; interprocedural analysis; unsound analysis; null pointer dereference

DOI: 10.15514/ISPRAS-2015-27(6)-8

For citation: Borodin A., Belevancev A.. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

References:

[1]. S. C. Misra, V. C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality //Computational Science and Its Applications—ICCSA 2003. – Springer Berlin Heidelberg, 2003. – pp. 724-732.

[2]. https://msdn.microsoft.com/library/cc307416

[3]. M. Tim Jones. Static and dynamic testing in the software development life cycle. 26 August 2013 (http://www.ibm.com/developerworks/library/se-static/)

[4]. A. S. Markov, V. L. Cirlov, A. V. Barabanov. Metody ocenki nesootvetstvija sredstv zawity informacii [Methods for assessing non-compliance means of information protection] //M.: Radio i svjaz' [Radio and Communication] – 2012. (in Russian)

[5]. T. Kremenek, D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations //Static Analysis. – Springer Berlin Heidelberg, 2003. – pp. 295-315.

[6]. V.N. Ignat'ev. Ispol'zovanie legkovesnogo staticheskogo analiza dlja proverki nastraivaemykh semanticheskikh ogranichenij jazyka programmirovanija [Static analysis usage for customizable checks of programming languages semantic constraints]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 22, 2012, pp. 169–188. DOI: 10.15514/ISPRAS-2012-22-11. (in Russian)

[7]. V.P. Ivannikov, A.A. Belevancev, A.E. Borodin, V.N. Ignat'ev, D.M. Zhurikhin, A.I. Avetisjan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v iskhodnom kode programm [Svace: static analyzer for detecting of defects in program source code] // Trudy ISP RAN [The Proceedings of ISP RAS], volume 26, issue 1, pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7. (in Russian)

[8]. A.I. Avetisjan, A.E. Borodin. Mekhanizmy rasshirenija sistemy staticheskogo analiza Svace detektorami novykh vidov ujazvimostej i kriticheskikh oshibok [Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, 2011, pp. 39–54. (in Russian)

[9]. A.I. Avetisjan, A.A. Belevancev, A.E. Borodin, V.S. Nesov. Ispol'zovanie staticheskogo analiza dlja poiska ujazvimostej i kriticheskikh oshibok v iskhodnom kode programm [Using static analysis for searching vulnerabilities and critical errors in the source code of programs]. Trudy ISP RAN [The Proceedings of ISP RAS], volume 21, 2011, pp. 23–38. (in Russian)

[10]. M. Shapiro, S. Horwitz. Fast and accurate flow-insensitive points-to analysis //Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM, 1997. – pp. 1-14.

[11]. L. De Moura, N. Bjørner. Z3: An efficient SMT solver //Tools and Algorithms for the Construction and Analysis of Systems. – Springer Berlin Heidelberg, 2008. – pp. 337-340.