Статический анализатор Svace для поиска дефектов в исходном коде программ

В.П. Иванников, А.А. Белеванцев, А.Е. Бородин,
В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов
{ivan, abel, alexey.borodin, valery.ignatyev, zhur, arut, maksim.leonov}@ispras.ru

Аннотация. В работе описывается разрабатываемый в ИСП РАН инструмент автоматического статического анализа Svace. Инструмент позволяет находить ошибки и потенциальные уязвимости в исходном коде программ на языках Си/Си++. Особенностью инструмента являются простота использования, широкий набор поддерживаемых типов предупреждений, масштабируемость до программ в миллионы строк кода и приемлемое качество анализа (30-80% истинных предупреждений).

Ключевые слова: статический анализ; анализ потока данных; уязвимости; межпроцедурный анализ; анализ на основе аннотаций.

1. Введение

Высокая сложность программ делает практически невозможным создание программного продукта без дефектов. Причём с увеличением размера программного обеспечения возрастает не только количество дефектов, но и их плотность [1]. Поэтому растет необходимость в инструментах и методах поиска дефектов. Одним из таких методов является статический анализ текстов программ. Анализ программ осуществляется без их реального выполнения. При этом происходит исследование всего кода программы, в том числе редко достигаемых участков кода, что позволяет найти ошибки, которые сложно воспроизвести, и которые обычно остаются незамеченными в ходе тестирования.

В статье описывается инструмент статического анализа Svace, разрабатываемый в ИСП РАН для анализа программ, написанных на языках C и C++. В настоящее время ведется добавление поддержки анализа программ, написанных на языке Java. Более ранние описания инструмента можно найти в статьях [2] и [3].

2. Постановка задачи

При разработке инструмента учитываются следующие основные требования:

- От пользователя требуются минимальные действия для интеграции Svace в систему сборки программы. Не нужно как-либо менять либо аннотировать исходный код программы, либо вмешиваться в процессе анализа.
- Выполняется глубокий межпроцедурный анализ, т.е учитывается влияние разных функций на поведение программы при поиске заданных ситуаций.
- Анализ должен быть масштабируемым. Размер анализируемых программ может достигать миллионов строк кода и сотен тысяч функций.
- Инфраструктура анализа должна быть расширяемой. В настоящее время анализатор позволяет находить довольно широкий класс ошибок: разыменование нулевого указателя, использование неинициализированных данных, переполнение буфера, уязвимости форматной строки, утечки памяти, некорректная работа с динамической памятью и др. Этот список постоянно дополняется.
- Доступ к полному исходному коду программы не является необходимым. Для стандартных библиотечных функций Svace имеет внутренние спецификации, описывающие их действия.
- Значительная часть выдаваемых предупреждений должна быть истинной, т.е. необходимо находить ситуации, действительно подпадающие под описание выданного типа предупреждения как ошибочные или опасные; это не означает, что такие ситуации обязательно будут уязвимостями или проявятся во время выполнения программы.

Дополнительными практическими задачами являются обеспечение удобного пользовательского интерфейса для просмотра результатов анализа и настройки инструмента, а также наличие дополнительных возможностей, таких как удаленный анализ и ведение истории результатов анализа.

Естественным ограничением, вытекающим из поставленных требований, является то, что не гарантируется нахождение всех ошибок и допускается наличие ложных срабатываний. Все известные коммерческие инструменты для автоматического статического анализа, не требующего трудоемкой подготовки анализируемых программ или требований к ним (SAVE компании Coverity [4] и Insight компании Klocwork [5]), обладают таким же ограничением.

3. Схема проведения анализа

Анализ с помощью Svace проходит в несколько этапов, как показано на рис. 1. Прежде всего необходимо подготовить внутреннее представление для исходного кода анализируемой программы. Для этого в процессе сборки анализируемого приложения перехватываются запуски компилятора и других утилит, влияющих на процесс сборки. При этом перехват проводится прозрачно, то есть не требует менять файлы системы сборки. С помощью перехваченных команд генерируются новые командные строки для запуска внутреннего компилятора Svace, который генерирует внутреннее представление для исходного кода приложения. Этот компилятор основан на модифицированном компиляторе Clang с открытым исходным кодом. В качестве внутреннего представления, соответственно, используется биткод LLVM. Он содержит описания всех используемых типов, глобальных и локальных переменных модуля компиляции, а также текст функций кода для абстрактной регистровой машины.

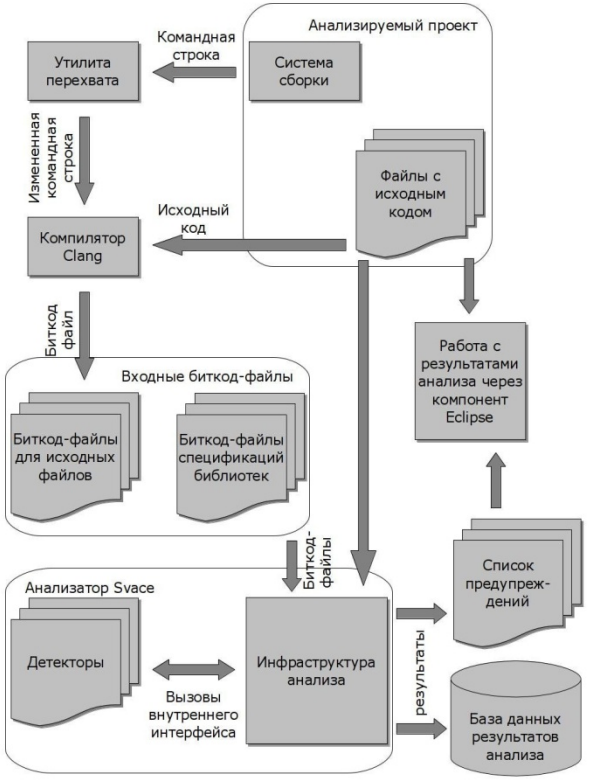

Рис. 1. Схема анализа

Файлы внутреннего представления исходного кода анализируемой программы сохраняются на диск для последующего анализа. При запуске анализа они считываются с диска вместе с файлами биткода, содержащими спецификации стандартных библиотечных функций. Инфраструктура анализа проводит анализ внутреннего представления, используя набор правил нахождения определенных видов дефектов, предоставляемых подключенными детекторами, и в результате выдает список предупреждений. Этот список может быть импортирован в базу данных истории результатов предупреждений для отслеживания предупреждений об исправленных или внесенных дефектах с предыдущего анализа. Также этот список может быть загружен в среду разработки Eclipse с возможностью навигации по участкам кода, соответствующим предупреждениям.

4. Межпроцедурный анализ на основе аннотаций

В основе Svace находится анализ “снизу-вверх”, основанный на параметризованных аннотациях. Во время анализа производится обход функций программы в соответствии с графом вызовов таким образом, чтобы вызываемая функция анализировалась перед вызывающей[1]. Каждая функция анализируется только один раз и независимо от других функций, что позволяет добиться почти линейной масштабируемости от количества функций в анализируемой программе. В результате анализа функции (помимо предупреждений о найденных дефектах) создаётся аннотация — структура данных ограниченного размера, приближенно описывающая наиболее важные для анализа эффекты от вызова этой функции в произвольном контексте. Различные детекторы могут сохранять в аннотациях необходимые им данные. Аннотации являются параметризованными — все свойства описываются относительно входных параметров и их отношений. В терминах анализа потока данных аннотация является структурой, достаточной для создания передаточной функции вызова процедуры.

Во время анализа кода каждой функции при обработке инструкции вызова функции используется её аннотация. Производится процесс трансляции аннотации: формальные параметры сопоставляются фактическим, побочные эффекты аннотации отражаются в контексте точки вызова. При этом анализ не меняет аннотацию вызываемой функции.

При анализе функции предполагается, что входные параметры указывают на непересекающиеся области памяти. Подобное допущение соответствует семантике большинства функций, и позволяет резко сократить сложность анализа указателей и повысить точность, а также скорость анализа [6].

1 Если подобное невозможно (например, при использовании рекурсивных вызовов), из графа вызовов удаляется одно из ребер.

Все аннотации функций по возможности хранятся в оперативной памяти, но при ее нехватке сохраняются на диск и считываются при необходимости. Сохранение на диск позволяет снять ограничение на размер анализируемого кода и выполнять анализ программ размером во много миллионов строк кода. Тем не менее, обычного современного объема оперативной памяти (1-4 Гб) хватает для анализа программ из сотен тысяч строк кода без использования сохранения на диск.

5. Внутрипроцедурный анализ

Во время анализа одной функции анализатору доступно внутреннее представление анализируемой функции, а также аннотации всех непосредственно вызываемых ей функций. Для функции строится граф потока управления, в узлах которого лежат инструкции анализируемой функции, а ребра отражают возможный переход управления между ними. Затем этот граф потока управления обходится во время символьного выполнения в топологическом порядке. После нескольких прямых проходов, производится дополнительный обратный проход.

С каждым ребром графа потока управления ассоциируется контекст — информация о потоке данных, установленная для путей исполнения, проходящих через данное ребро. Контекст описывает взаимосвязь между ячейками памяти, идентификаторами значений и атрибутами. Абстрактные ячейки памяти моделируют ячейки памяти, к которым происходит обращение в программе на различных путях исполнения. Идентификаторы значений обозначают значения, разделяемые различными ячейками памяти без изменения (схожей цели служат поколения переменных в представлении с единственным присваиванием, SSA). Например, для значения, скопированного из одной переменной в другую, поддерживается один идентификатор значения, так что этот идентификатор становится общим для нескольких абстрактных ячеек памяти. Атрибуты описывают анализируемые свойства абстрактных ячеек памяти и идентификаторов значений, например, интервал возможных целочисленных значений или зависимость от данных, полученных из сети.

Больше всего Svace оперирует с атрибутами идентификаторов значений, т.к. наиболее интересные свойства являются свойствами значений. Например, при поиске ошибок разыменования нулевого указателя необходимо проверить, что значение переменной, которую разыменовывают, равно нулю. Для этого можно создать специальный атрибут, который будет обозначать свойство, что значение равно нулю (противоположное свойство - “значение не обязательно равно нулю”), и помечать им идентификаторы значений переменных, которые заведомо равны нулю. После этого для выявления дефекта достаточно при обработке инструкции разыменования проверить значение такого атрибута у идентификатора значений, соответствующего разыменовываемой абстрактной ячейке памяти.

Для каждого типа дефектов определяются свои атрибуты, необходимые для обнаружения искомых ситуаций, и правила их распространения. В основную задачу инфраструктуры анализа входит распространение атрибутов между контекстами на ребрах (в зависимости от инструкций в узлах) графа потока управления в пределах одной функции во время внутрипроцедурного анализа и между различными функциями с помощью аннотаций.

Несмотря на то, что аннотация функции создается без учета вызываемого контекста, обработка вызова функции производится контекстно-чувствительным образом. Инфраструктура анализа производит трансляцию аннотации в контекст вызова, при этом элементы вызывающего контекста изменяются в зависимости от содержания аннотации.

На рис. 3 схематично показан пример аннотации для функции, изображённой на рис. 2. На схеме p и x - формальные параметры функции, относительно которых параметризуется аннотация; ret - возвращаемое значение функции; Vp, Vx - идентификаторы значений для формальных параметров; *p - абстрактная ячейка памяти, обозначающая область памяти, на которую указывает формальный параметр p. В аннотации хранится информация о том, что функция возвращает значение своего формального параметра Vx, в ячейку памяти *p записано значение Vx. Дополнительно сохраняются атрибуты deref и pt-to, которые означают, что значение Vp было разыменовано и указывает на абстрактную ячейку памяти *p.

```
int set_p(int *p, int x) {
    *p = x;
    return x;
}
```

Рис. 2. Код модельной функции

Идентификаторы значений		Ячейки памяти	
p	Vp	*p	Vx
x	Vx	Атрибуты	
ret	Vx	Vp	deref, pt-to { *p }

Рис. 3. Аннотация модельной функции

6. Чувствительность к путям

Одним из полезных качеств анализа является чувствительность к путям, когда каждый возможный путь выполнения кода функции анализируется независимо. Однако из-за практических требований такой анализ нельзя проводить полноценно. В некоторых случаях это может приводить к ложным срабатываниям.

```
1: if(!x)
2:     y = 0;
3: if(x)
4:     *y;
```

Рис. 4. Зависимые условия.

В примере на рис. 4 из 4 путей выполнения программы (1→2→3→4, 1→3→4, 1→3→4, 1→3) два являются невыполнимыми (1→2→3→4 и 1→3), так как условия x и !x взаимно исключают друг друга. Поэтому, если выполнение достигло строки 4, то инструкция присваивания нулевого значения в строке 2 не могла быть выполнена. Для повышения точности анализа при сохранении масштабируемости в Svace используется два механизма: зависимые атрибуты и обратный анализ.

Зависимые атрибуты — это атрибуты, имеющие ссылки на идентификаторы значений. Можно создать атрибут IsNull, означающий, что переменная имеет нулевое значение, если другая переменная имеет такое значение. В примере у имеет нулевое значение, если x равно нулю. Зависимые атрибуты позволяют добавить чувствительность к путям для отдельных типов дефектов, для которых чувствительность к путям особенно критична. Svace использует зависимые атрибуты, например, для поиска ошибок двойных блокировок. Это обусловлено тем, что, во-первых, блокировки довольно часто находятся под некоторым флагом, а во-вторых, блокировки используются довольно редко, поэтому их более детальное рассмотрение не замедлит анализ в целом.

Другим механизмом подавления ложных срабатываний на несуществующих путях является обратный анализ. Цель обратного анализа — выделить ситуации, в которых, если происходит одно событие, то другое будет происходить для всех возможных путей выполнения. В примере выше после выполнения инструкции присваивания в строке 2, сравнение в строке 3 выполнится в любом случае, а разыменование в строке 4 только, если условие будет истинным.

На рис. 5 приведены некоторые возможные варианты использования присваивания нулю и разыменования. Инструкции обозначены прямоугольниками, пустые прямоугольники обозначают любые другие

инструкции, которые не изменяют переменную x и не разыменовывают её. На рисунках А, Б, Г и Д после присваивания переменной нуля всегда происходит разыменование нулевого указателя. При этом на рисунке Г разыменование происходит в разных местах, а на рисунке Д дважды встречается присваивание нулю. На рисунке В, если управление дошло до инструкции присваивания, то разыменование выполнится в любом случае. А на рисунке Е наоборот, если управление попало в инструкцию разыменования, то присваивание обязательно было выполнено. Во всех выше перечисленных случаях выдается ошибка разыменования нулевого указателя. На рисунке Ж и присваивание и разыменование выполняются условно, если условия зависимые, то без дополнительного анализа нельзя сказать, будет ли разыменование нулевого указателя в этом случае.

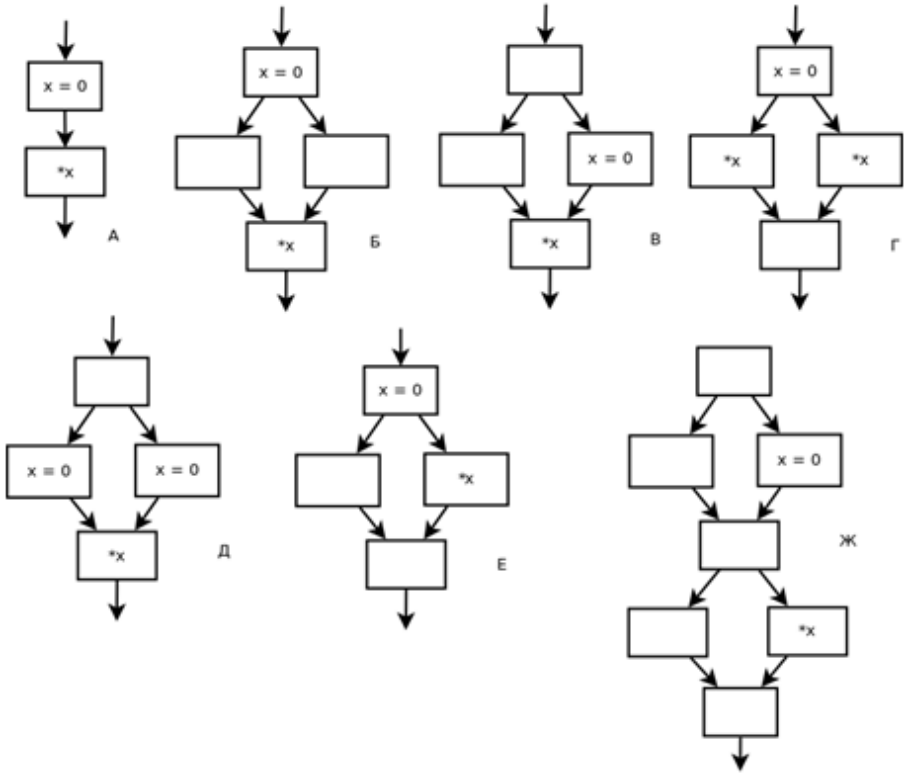

Рис. 5. Использование обратного анализа

Обратный анализ позволяет различить ситуации А-Е от Ж. Общая идея для рассматриваемого типа дефекта заключается в использовании двух булевых

атрибутов IsNull и IsDeref. Во всех контекстах, соответствующих инструкциям программы, которые выполняются после присваивания переменной нулевого значения, атрибут IsNull для этого идентификатора значений будет иметь истинное значение. Этот атрибут вычисляется только при прямом анализе. На обратном проходе атрибут IsDeref получает истинное значение для тех идентификаторов значений, которые были разыменованы на все путях ниже по графу потока управления. В точке, где после обратного анализа идентификатор значения переменной имеет одновременно истинные атрибуты IsDeref и IsNull, выдаётся ошибка разыменования нулевого указателя, т.к. на всех путях выполнения, проходящих через данную точку, было присваивание переменной нулю, и в то же время будет разыменование этой переменной. Таким образом, благодаря обратному анализу можно эффективно и с минимальными накладными расходами получить частичную чувствительность к путям.

7. Пример поиска предупреждения DEREF_AFTER_NULL

Опишем детектор, осуществляющий поиск следующего дефекта: «переменная была разыменована после положительного сравнения с нулём». Приведём пример работы детектора для фрагмента кода на рис. 6. В строке 4 переменная x сравнивается с нулём, а затем разыменовывается либо в строке 8, либо в строке 10. Таким образом пример содержит либо лишнее сравнение с нулём в строке 4, либо разыменование нулевого указателя.

```
1: func(int*x, int* p, int m) {
2:     if(p==0) {}
3:     else {
4:         if(x==0) {}
5:         else *p = 0;
6:     }
7:     if(m==0)
8:         *x = 1;
9:     else
10:       *x = 2;
11: }
```

Рис. 6. Пример для DEREF_AFTER_NULL

Для поиска ошибки мы используем два булевых атрибута идентификаторов значений: IsNullCompared, имеющий значение истина, если значение переменной было положительно сравнено с нулём на всех путях выше по графу потока управления, и ложь иначе (отсутствие информации); и атрибут IsDeref, имеющий значение истина, когда значение было разыменовано без проверки на ноль, и ложь во всех остальных случаях. Предупреждение

выдаётся, если после положительной проверки на нуль на всех достижимых путях выполнения происходит разыменование нулевого указателя. Нам потребуется обратный анализ, чтобы распространить атрибут IsDeref до точек, где значение указателя всегда равно нулю (IsNullCompared имеет значение истина).

Выполнение анализа показано на рис. 7. Для удобства рисунок содержит точки функции, соответствующие пустым веткам условия в строках 2 и 4, а также две вершины Join1 и Join2, обозначающие слияние путей выполнения. На рисунке показаны детали анализа только для переменных x, p и m. Для обратного анализа стрелки между инструкциями показывают направление анализа, а не выполнение программы. Предупреждение DEREF_AFTER_NULL выдаётся, т. к. в строке 4 идентификатор значения имеет истинными одновременно атрибуты IsNullCompared и IsDeref.

Рис. 7. Прямой и обратный анализ функции func

8. Расширение анализа

Для расширения возможностей анализатора нужно уметь добавлять новые виды обнаруживаемых дефектов. При разработке Svace требовалось, чтобы добавление нового типа предупреждения не влекло серьёзной переработки всей системы. Для этого в составе Svace выделяется инфраструктура анализа, содержащая общие части для всех видов анализов, а конкретный анализ реализуется с помощью детекторов для каждого типа дефекта или группы схожих дефектов [7].

Инфраструктура анализа создаёт идентификаторы значений и абстрактные ячейки памяти, оповещает детекторы о важных операциях в программе, а детекторы могут подписываться на интересующие их операции, просматривать состояние программы перед выполнением операций, и модифицировать атрибуты выходного состояния. При этом для большинства случаев используются стандартные типы атрибутов: булевы атрибуты, тернарные атрибуты, интервалы, двойные интервалы, или атрибуты, параметризованные идентификаторами значений.

Такая структура позволяет достаточно легко добавлять новые типы предупреждений. Для большинства видов потенциальных ошибок необходимо понять, в чём состоит нежелательная ситуация, какими свойствами должны обладать переменные и их значения для возникновения ошибки. После этого, чтобы реализовать детектор, достаточно описать то, как нужно помечать переменные или их значения некоторыми атрибутами, обозначающими интересующие свойства, а также указать правила продвижения атрибутов по контексту и условия выдачи предупреждения.

Ещё одним преимуществом разделения анализа на общую инфраструктуру и детекторы является низкая стоимость включения отдельного детектора. Большинство необходимых действий производится инфраструктурой анализа, поэтому добавление нового детектора незначительно изменяет общее время анализа.

9. Скорость и масштабируемость

Высокая скорость позволяет чаще производить поиск ошибок, в том числе для вновь сделанных изменений. Медленный анализ может стать причиной отказа от инструмента. Другой важной характеристикой является масштабируемость. Многие инструменты не способны проанализировать большие программы из-за нелинейного роста требований к памяти и процессорному времени. Это сильно ограничивает сферу их применения. При разработке Svace скорость и масштабируемость были одними из главных требований, которые были успешно учтены.

В табл. 1 приведены времена анализа инструментом Svace некоторых проектов различного размера. Для проведения анализа использовались две машины: персональный компьютер с процессором Intel Core i7 (4 ядра) и 12ГБ

оперативной памяти и двухпроцессорный сервер на основе Intel Xeon с 8 ядрами и 70ГБ оперативной памяти, на обоих системах реализована технология гиперпоточность (hyper-threading). Учитывалось только время анализа, время требуемое для компиляции проектов не приведено. При этом были включены все реализованные детекторы. Для операционной системы Android анализ производился только на Intel Xeon. Ядро операционной системы Linux на обычном компьютере было проанализировано быстрее, чем за полчаса. Существует довольно мало проектов большего размера чем ядро Linux и операционная система Android. Поэтому можно считать, что представленный анализ масштабируется для любых существующих проектов.

Пакет	Размер, тыс. строк кода	Количество функций	Скорость анализа для Intel Core i7, 4 ядра, 12 Гб	Скорость анализа для Intel Xeon, 8 ядер, 70 Гб
pcre3-7.8	19	506	2 м. 59 с.	40 с.
openssh-5.3p1	65	1549	1 м. 15 с.	51 с.
openssl-0.9.8k	240	5030	3 м. 53 с.	2 м. 25 с.
ffmpeg-0.5.1	300	7119	6 м. 21 с.	3 м. 33 с.
postgresql-8.4	501	13237	11 м. 35 с.	6 м. 10 с.
binutils-2.20.1	1086	24271	19 м. 20 с.	5 м. 35 с.
linux-kernel-3.10.9	6219	53114	28 м. 12 с.	9 м. 58 с.
android-4.2.1	12250	584743	-	2 ч. 43 м.

Табл. 1. Время анализа проектов с помощью анализатора Svace

В настоящее время широкое распространение получили вычислительные машины с несколькими процессорами или ядрами. Алгоритм анализа на основе аннотаций, применяемый в Svace, довольно хорошо распараллеливается и позволяет эффективно использовать многоядерные и многопроцессорные машины. В качестве атомарной единицы анализа используется анализ функции, поэтому потенциал распараллеливания зависит

от ширины графа вызовов. В табл. 2 приведено время анализа приложения ffmpeg-0.5.1 на упоминавшемся выше сервере Intel Xeon при использовании разного количества потоков.

Количество используемых потоков	**Время анализа, сек.**
1	1275
2	652
3	445
4	343
6	252
8	225
16	213

Табл. 2. Анализ с разным количеством используемых потоков.

10. Легковесный анализ лексических, синтаксических, семантических и ряда ситуационных ограничений

Для проверки языково-зависимых правил языков C и C++ может использоваться легковесный анализатор на основе Clang [8]. Он позволяет быстро анализировать лексические, синтаксические, семантические и ряд ситуационных правил. В отличие от Svace, доступна полная информация об исходном тексте программы, в том числе, не влияющая на семантику программы, например, комментарии, переводы строк, пробелы и т.д. В качестве внутреннего представления используется аннотированный абстрактный семантический граф (ААСГ), строящийся на основе абстрактного синтаксического дерева, дополненного помеченными ребрами и аннотациями узлов. ААСГ позволяет моделировать не только семантику языка, но и граф потока управления, граф вызовов и используется для анализа потоков данных. Каждый узел графа имеет тип, определяющий его семантику. Множество всех типов ограничено и состоит из подмножеств для каждого этапа компиляции: *Lex={Keyword, Identifier, ...}, AST={IfStmt, BinaryOperator, ...}* и т.д. Для анализа работы с памятью используется понятие *область памяти* $A = StorageCla\,ss \times N_0$, где $N_0 = N \bigcup \{0\}$ – декартово произведение класса памяти и размера в битах, что позволяет учитывать битовые операции и структуры с битовыми полями. Всего определено 10 классов памяти, разделенных на 4 подкласса:

1. стековый (автоматические переменные, параметры функции, выделенная alloca);
2. код (указатели на функцию, метки);
3. данные (глобальная системная память, глобальная статическая, глобальная);
4. динамическая память (выделенная malloc, new).

Для поддержки элементов массива и полей структур используется *подобласть памяти* $A_{\subset} = A_{base} \times N_0$ – декартово произведение базовой области памяти и размера. Состояние памяти задается как $Stmt \times Env \times Mem$, где Stmt – идентификатор оператора программы, $Env = \{\alpha : Var \mapsto A, \alpha^f : A_{base} \times Field \mapsto A_{\subset}, \alpha^e : A_{base} \times N_0 \mapsto A_{\subset}\}$ – состояние переменных, $Mem = \{\lambda : A \mapsto \{0,1\}^*, \hat{\lambda} : A \mapsto A_{\subset}\}$ – состояние памяти. Контекст функции не учитывается для ускорения анализа. В табл. 3 приведены примеры правил вычисления областей памяти для выражений C и C++.

Выражение(E)	l(E)	r(E)
константа C	Ø	C
переменная V	A^V	$\lambda(A^V)$
s.f	$A^f_{\subset}(A^s)$	$\lambda(A^f_{\subset}(A^s))$
*E	$\hat{\lambda}(l(E))$	$\lambda(l(E))$

Табл. 3. Пример вычисления l-value и r-value выражений

Для ускорения анализа необходимо избегать полной символьной интерпретации и вычислять только необходимую информацию. Поэтому лишь минимальное количество часто используемых атрибутов вычисляется заранее. На начальных этапах работа анализатора аналогична компилятору. Основное отличие заключается в вычислении дополнительных атрибутов и проверке ограничений. Правила не модифицируют ААСГ, а только активируют функции вычисления «редких» атрибутов, которые могут быть сохранены.

Поэтому они обрабатываются параллельно. Для ряда типов правил, например, для проверки корректности обработки исключений, разработаны межмодульные алгоритмы анализа, экспортирующие необходимые данные в процессе работы и выполняющие проверку на этапе линковки с помощью интеграции в систему сборки программ. Система состоит из трёх основных компонентов: подсистема определения правил, сбора информации и планировщик правил. Ограничения задаются в виде классов, унаследованных от базового класса правил и содержат список типов необходимых узлов ААСГ, используемый для планирования порядка их проверки. Доступ к атрибутам узлов ААСГ осуществляется с помощью методов соответствующего класса, что позволяет получать доступ к данным по мере вычисления их анализатором или запускать “ленивые” алгоритмы их вычисления. Реализованная подсистема способна проверять более 50 различных правил. Среднее замедление сборки проекта при включенном анализаторе составляет 22 %, что допускает его применение при каждой компиляции и позволяет обнаруживать ошибки на самых ранних стадиях. Результаты работы легковесного анализатора могут быть импортированы в Svace позднее во время его анализа и показаны наряду с предупреждениями, выданными самим Svace.

11. Просмотр результатов в среде Eclipse

После окончания анализа его результаты могут быть просмотрены пользователем через компонент в среде разработки Eclipse. Для каждого предупреждения показывается сообщение и релевантные места в исходном коде программы (таких мест может быть несколько, например, может быть указана полная межпроцедурная трасса по функциям программы, приводящая к ошибке). Пример просмотра результатов анализа в Eclipse приведен на рис. 8.

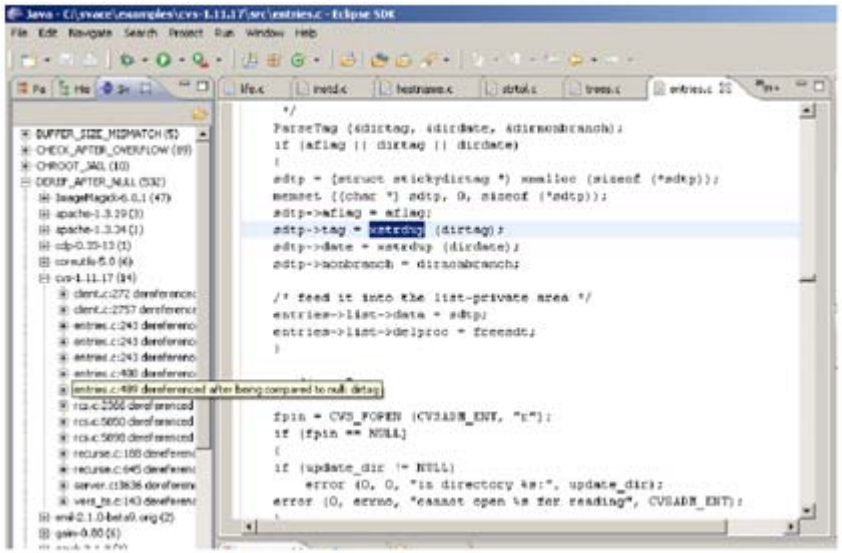

Рис. 8. Графический интерфейс компонента Svace в среде разработки Eclipse

12. База данных результатов анализа

Результаты анализа, представляющие из себя набор предупреждений, могут сохраняться как в специальных файлах, хранящих в себе результаты отдельного анализа, так и в базе данных. База данных позволяет сохранять результаты множества последовательных анализов одного проекта (между которыми исходный код проекта может меняться), и сопоставляет предупреждения с этих различных анализов, что позволяет отслеживать историю изменения отдельных предупреждений. Также база данных позволяет пользователю устанавливать комментарии и различные метки для предупреждений (например, размечать истинные и ложные срабатывания анализатора). Они автоматически переносятся между последовательными результатами анализа. Сопоставление предупреждений на различных результатах анализа производится путём выявления для каждого сопоставляемого нового предупреждения наиболее похожих предупреждений из уже существующих в базе данных. Схожесть предупреждений вычисляется на базе различных параметров, таких как тип предупреждений, контекст положения предупреждения в исходном коде, наборе имён участвующих в проблемной ситуации переменных.

Система контроля истории результатов анализа также позволяет производить синхронизацию хранилищ результатов анализа по сети. Это позволяет получать новые результаты анализа (например, с централизованного сервера анализа) и производить синхронизацию комментариев и меток, созданных различными пользователями.

13. Примеры находимых дефектов

В данном разделе приведено несколько типичных ошибок, найденных в популярных проектах с исходным открытым кодом.

Многие ошибки являются результатом вставки кода из другого места и долгое время остаются незамеченными. На рис. 9 показан пример подобной ошибки. Функция fopen в случае неудачного выполнения возвращает нулевой указатель. В этом случае приведённый код сообщает об ошибке, очищает ресурсы и зачем-то вызывает функцию fclose. Передача нулевого указателя в функцию fclose является неопределённым поведением. Подобную ошибку легко допустить и относительно сложно найти с помощью тестирования, т. к. она происходит на редком пути исполнения. При этом для статических анализаторов реализация детектора для нахождения подобных дефектов не является сложной задачей.

```
outfile = fopen(argv[2], "wb");
if (!outfile) {
      perror(argv[2]);
      fclose(outfile);
      free(moov_atom);
      return 1;
}
```

Рис. 9. Ошибка DEREF_OF_NULL в ffmpeg-0.5.1/tools/qt-faststart.c

Некоторые библиотечные функции могут возвращать нулевой код возврата в качестве индикации ошибки. Например, функция malloc стандартной библиотеки языка Си возвращает нуль, если не удалось выделить память. Программисты часто не проверяют код возврата функции malloc, т.к. нехватка памяти довольно редкое явление, и даже в случае недостатка памяти, непонятно что делать с такой ситуацией. Тем не менее во многих проектах приняты более жёсткие стандарты, в которых необходимо проверять результат malloc. Типичный пример подобной ошибки, найденной в busybox, приведён на рис. 10.


```
sym->user.val = val = malloc(size);
*val++ = '0';
*val++ = 'x';
```

Рис. 10. Ошибка DEREF_OF_NULL.ALLOC в busybox-1.13.3/scripts/kconfig/symbol.c

Ошибки, допущенные в программе, могут приводить не только к падению программы или некорректной работе, но и создавать уязвимости системы. Svace имеет группу предупреждений, проверяющих, что полученные из внешней среды данные не используются в критичных операциях. Пример подобной уязвимости показан на рис. 11, где переменная env указывает на массив, полученный из переменной окружения с помощью функции getenv, которая затем без проверок используется в качестве аргумента форматной строки в функции sprintf.

```
env = getenv(SRCTREE);
if (env) {
 sprintf(fullname, "%s/%s", env, name);
}
```

Рис. 11. Ошибка TAINTED_PTR в busybox-1.13.3/scripts/kconfig/confdata.c

Поиск утечек памяти является сложной задачей, тем не менее, многие ошибки подобного вида могут быть найдены с помощью статического анализа. На рис. 12 показан фрагмент кода, где с помощью функции strdup выделяется память, которая затем присваивается указателю ‘type’ и никогда не освобождается.

```
case 't': {
          char *type;

          type = strdup(optstate->value);
          crlType = atoi (type);
          if (crlType != SEC_CRL_TYPE && crlType
!= SEC_KRL_TYPE) {
            PR_fprintf(PR_STDERR, "%s: invalid crl
type\n", progName);
            PL_DestroyOptState(optstate);
            return -1;
          }
          break;
```

Рис. 12. Ошибка MEMORY_LEAK.STRDUP в nss-3.12.6/mozilla/security/nss/cmd/crlutil/crlutil.c

14. Заключение

В статье описывается инструмент статического анализа Svace, нацеленный на практическое использование для автоматического поиска дефектов и уязвимостей в программах на языках Си и Си++. В основе анализа лежит межпроцедурный анализ потока данных, что позволяет обнаруживать дефекты, причинами которых являются сложные взаимодействия между функциями разных модулей компиляции. При этом анализ обладает хорошей скоростью, в том числе за счет параллелизации, и масштабируется до проектов из миллионов строк кода. Качество анализа при этом сравнимо с лучшими коммерческими аналогами. Svace прост в использовании и обладает расширенным набором пользовательских возможностей, в том числе, графическим интерфейсом и поддержкой истории результатов анализов. Инструмент находится в развитии и в ближайшее время получит возможность анализировать программы на языке Java.

Список литературы

[1]. S.C. Misra, V.C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality, in: Proceedings of the International Conference on Computational Science and its Applications, ICCSA, in: Lecture Notes in Computer Science, vol. 2667, Springer, Montreal, Canada, 2003, pp. 724–732.

[2]. В.С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции «РусКрипто'2009».

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

[4]. Инструмент Coverity SAVE. http://www.coverity.com/products/coverity-save.html

[5]. Инструмент статического анализа компании Klocwork. http://www.klocwork.com/products/insight/klocwork-truepath

[6]. V. Benjamin Livshits, Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs, 2003.

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

[8]. Игнатьев В.Н. 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.

Svace: static analyzer for detecting of defects in program source code

Viktor Ivannikov <ivan@ispras.ru>, ISP RAS, Moscow, Russia
Andrey Belevantsev <abel@ispras.ru>, ISP RAS, Moscow, Russia
Alexey Borodin <alexey.borodin@ispras.ru>, ISP RAS, Moscow, Russia
Valery Ignatyev <rook@ispras.ru>, ISP RAS, Moscow, Russia
Dmitry Zhurikhin <zhur@ispras.ru>, ISP RAS, Moscow, Russia
Arutyun Avetisyan <arut@ispras.ru>, ISP RAS, Moscow, Russia
Maksim Leonov <maksim.leonov@ispras.ru>, ISP RAS, Moscow, Russia

Abstract. High complexity of present-day programs makes it nigh impossible to write a program without a defect. Thus it is increasingly necessary to use tools for defects detection. This article presents Svace, a tool for static program analysis developed in ISP RAS. This instrument allows to automatically find defects and potential vulnerabilities in programs written in C and C++ languages. Main features of the tool are simplicity of usage, deep interprocedural analysis, wide variety of supported warning types, scalability up to programs of millions lines of code and acceptable quality of analysis (30-80% of true positive warnings).

In the core of the Svace tool lies an engine for interprocedural data-flow analysis based on function annotations. Each function is analyzed once and independently of the other functions which allows to achieve almost linear scalability (Linux kernel can be analyzed within 10 minutes on a relatively powerful machine and analysis of the whole Android source code takes less than 3 hours). Intraprocedural analysis is performed on source code internal representation derived from LLVM bitcode. It operates with value identifiers that are shared between memory locations with same values (similarly to generations in SSA representation). Special attributes of these value identifiers are calculated over the control-flow graph of the function. When specific combination of attributes is observed a defect warning is issued. Svace analysis engine is accompanied by Clang compiler-based lightweight analysis tool for checking of language-dependent rules which allows to quickly check a number of syntactic, semantic and situational rules. Analysis results can be presented to the user with the help of Eclipse IDE plugin. They can also be imported into analysis results database to trace history of program defects over time.

Keywords: static analysis; data-flow analysis; vulnerabilities; interprocedural analysis; annotation-based analysis.

References

[1]. S.C. Misra, V.C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality, in: Proceedings of the International Conference on Computational Science and its Applications, ICCSA, in: Lecture Notes in Computer Science, vol. 2667, Springer, Montreal, Canada, 2003, pp. 724–732.

[2]. V.S. Nesov. Automatic defect detection with the help of interprocedural static analysis of source code, in: Proceedings of International Conference Ruscrypto'2009.

[3]. Coverity SAVE, http://www.coverity.com/products/coverity-save.html

[4]. Klocwork Insight, http://www.klocwork.com/products/insight/klocwork-truepath

[5]. V. Benjamin Livshits, Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs, 2003.

[6]. A. Avetisyan, A. Borodin. Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors, in: Proceedings of the Institute for System Programming of RAS, volume 21, 2011, pp. 39-54.

[7]. V.N. Ignatyev. Using static analysis for checking configurable semantic restrictions on a programming language, in: Proceedings of the Institute for System Programming of RAS, volume 22, 2012, pp. 169-188. DOI: 10.15514/ISPRAS-2012-22-11.