# Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ

*Арутюн Аветисян <arut@ispras.ru>, Андрей Белеванцев <abel@ispras.ru>,*
*Алексей Бородин <alexey.borodin@ispras.ru>,*
*Владимир Несов <nesov@ispras.ru>*

**Аннотация.** Статический анализ является популярным средством поиска в исходном или двоичном коде программ определенных шаблонов или ситуаций (ошибок стиля кодирования, нарушений проектных соглашений об использовании определенных библиотек или свойств языка программирования, критических ошибок, уязвимостей, закладок). В данной статье предлагается обзор инструмента статического анализа исходного кода программ на языках Си/Си++, разработанного в ИСП РАН для поиска критических ошибок и уязвимостей. Применение межпроцедурного анализа потока данных, не гарантирующего нахождение всех заданных ситуаций, позволяет проводить автоматический анализ с долей истинных предупреждений в 40-80%, что находится на уровне лучших коммерческих инструментов статического анализа.

**Ключевые слова:** статический анализ, анализ потока данных, интервальный анализ, межпроцедурный анализ, уязвимости.

## *1. Введение*

Высокая сложность программ влечет необходимость механизмов защиты от атак, в том числе поиска уязвимостей и ошибок, через которые могут происходить такие атаки. Одним из средств обнаружения критических ошибок, не выявляемых обычным тестированием, является статический анализ программ, а именно поиск ситуаций в исходном коде программы, которые могут означать наличие уязвимости. Примером простых ситуаций подобного типа являются выдаваемые компилятором предупреждения, например, об использовании указателя одного типа для манипулирования объектом другого типа. В более сложных ситуациях требуется применение глубокого анализа, который не может быть выполнен компилятором из-за практических ограничений (например, по времени анализа). Обычно такой анализ реализуется как отдельный инструмент с широким привлечением компиляторных технологий.



Данная статья предлагает обзор инструмента статического анализа Svace, разработанного в Институте системного программирования РАН для анализа программ на языках Си/Си++ и подробно описанного в статьях [1-9]. Инструмент генерирует список предупреждений с указанием типа предупреждения и данных о том, как именно может произойти ситуация, описанная в предупреждении (например, указатель на объект, созданный в одной функции программы, может принять нулевое значение в другой функции и после использоваться в третьей).

Инструмент разработан с учетом следующих требований:

- анализ производится автоматически, не требуется специально подготавливать исходный код либо вмешиваться в ходе анализа;
- анализ является межпроцедурным, то есть учитывает влияние разных функций программы на ее поведение при поиске заданных ситуаций;
- размер анализируемых программ может достигать миллионов строк кода и сотен тысяч функций (например, ядро ОС Linux, браузер Mozilla Firefox);
- инфраструктура анализа расширяема, т.е. возможно разработать дополнительные алгоритмы для поиска новых видов потенциально опасных ситуаций в коде программы с использованием имеющихся алгоритмов и подготовленных ими данных;
- доступ к полному исходному коду программы и коду используемых программой библиотек не является необходимым – достаточно иметь спецификации лишь некоторых часто используемых стандартных библиотечных функций;
- значительная часть найденных предупреждений должна быть истинной (т.н. true positive), т.е. необходимо находить ситуации, действительно подпадающие под описание указанного типа предупреждения как ошибочные или опасные; это не означает, что такие ситуации обязательно будут уязвимостями или проявятся во время выполнения программы [15].

Для достижения описанных целей (достаточно хорошее качество достаточно глубокого автоматического анализа при сохранении масштабируемости) необходимостью является применение такого анализа, который не гарантирует нахождение всех ситуаций заданных типов в исходном коде (англоязычный термин – unsound analysis), то есть качество анализа повышается за счет пропуска некоторого числа ошибок. Все известные коммерческие инструменты для автоматического статического анализа, не требующего трудоемкой подготовки анализируемых программ или требований к ним (Prevent компании Coverity [12] и K9 компании Klocwork [13]), используют тот же подход. При этом доля истинных предупреждений этих коммерческих инструментов, так же, как и разработанного нами инструмента, составляет обычно от 30% до 80% (в зависимости от типа

обнаруживаемой ситуации в исходном коде; дополнительная информация по этим инструментам доступна в сравнении [16]).

Текущая версия разработанного нами инструмента поддерживает работу в ОС Linux и Windows для анализа программ, написанных для работы на ОС Linux и предназначенных для сборки компиляторами GCC или ARMCC. Пользователю доступен интерфейс командной строки и графический интерфейс, реализованный в среде Eclipse [14].

Далее в статье мы рассмотрим архитектуру и поведение разработанного инструмента, а также характеристики результатов, достигнутых при анализе программ с открытым исходным кодом. Мы будем называть найденную ошибочную ситуацию в исходном коде *предупреждением*, которое является *истинным* в случае действительного наличия описываемой данным типом предупреждения ситуации в коде и *ложным* в обратном случае. *Детектором* (checker) будем называть компонент инструмента анализа, ответственный за поиск предупреждений определенного типа (или схожих типов). Детекторы используют остальные подсистемы анализа (в т.ч. другие детекторы) и обычно малы по сравнению со всем инструментом.

## 2. Архитектура инструмента статического анализа

Инструмент анализа оперирует файлами с *внутренним представлением* (IR), сгенерированными компилятором из файлов исходного кода программы и содержащими описания всех используемых типов, глобальных и локальных переменных модуля компиляции, а также текст функций в кода для абстрактной регистровой машины. Нами используется компилятор LLVM-GCC на основе открытого компилятора GCC [10], генерирующий внутреннее представление для LLVM [11] – популярной системы для построения компиляторов. Представление LLVM (т.н. биткод) компактно, его можно сохранять на диске как в двоичном, так и в текстовом виде, добавлять к коду программы метаданные, компоновать и т.п. Такое представление хорошо подходит для использования в системах статического анализа.

Для построения файлов с LLVM-биткодом (биткод-файлы) ипользуется специальная утилита *перехвата сборки*, поставляемая вместе с инструментом анализа, которая перехватывает все запуски указанного ей компилятора в ходе исполнения произвольной системы сборки анализируемой программы, разбирает командную строку запуска компилятора, формирует команду для выполнения компилятора LLVM-GCC и использует ее для построения биткод файлов анализируемой программы. Утилита перехвата сборки поддерживает как операционные системы на базе ядра Linux, так и ОС Windows, при этом способы ее реализации различны для этих ОС.



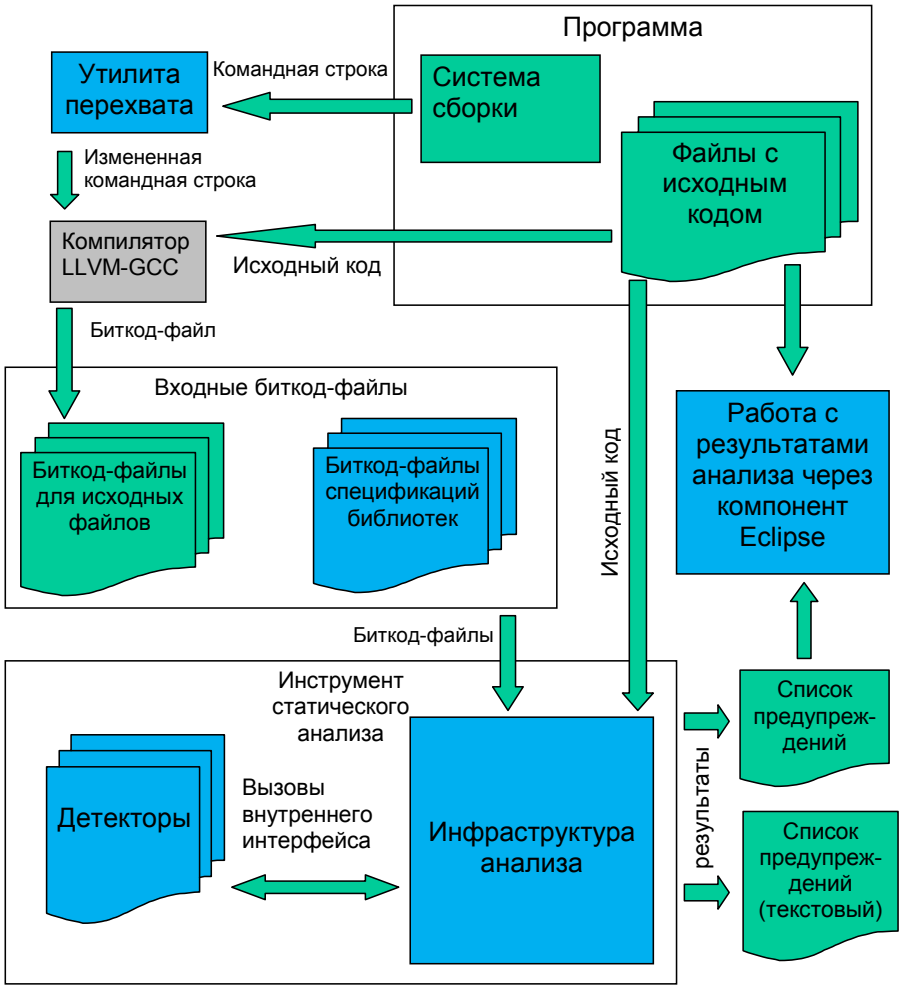


*Рис. 1. Схема работы инструмента статического анализа.*

Инструмент статического анализа состоит из инфраструктуры анализа, управляющей ходом анализа и реализующей набор общих алгоритмов, и набора детекторов, небольших компонент, осуществляющих поиск конкретных типов предупреждений и реализующих специальные алгоритмы анализа, необходимые для обнаружения ситуаций соответствующего вида. Помимо файлов внутреннего представления, при анализе используются

спецификации некоторых стандартных библиотечных функций, входящие в состав инструмента анализа, и (в некоторых случаях) файлы с исходным кодом анализируемой программы. Результаты анализа выдаются как в текстовом виде, так и во внутреннем формате, который может быть просмотрен в среде Eclipse с помощью поставляемого с инструментом анализа встраиваемого компонента – при этом текст предупреждения автоматически показывается в окружении соответствующего исходного кода.

Схема работы инструменты приведена на рис. 1, где компоненты инструмента выделены синим, а обрабатываемые данные – зеленым цветом. Компилятор LLVM-GCC выделен серым, так как он является сторонним компонентом. В начале работы утилита перехвата запуска компилятора запускает систему сборки программы обычным образом, параллельно создавая биткод-файлы. После окончания сборки над множеством сгенерированных биткод-файлов запускается инструмент анализа, который также подгружает спецификации библиотечных функций, необходимые для анализа и хранящиеся в виде биткод-файлов (далее работа со спецификациями ведется так же, как и с биткод-файлами анализируемой программы). Управление ходом анализа ведет инфраструктура анализа, вызывающая детекторы по мере надобности (детекторы регистрируют обработчики интересующих их событий, возникающих в ходе анализа программы). После окончания анализа сгенерированные предупреждения сохраняются в текстовом виде и внутреннем формате. Компонент Eclipse позволяет просматривать результаты с привязкой к исходному коду программы.

## *3. Ход анализа*

### 3.1. Межпроцедурный анализ и аннотации

Распространяемые между функциями данные о программе хранятся в ходе анализа в виде аннотаций функций. *Аннотация* является структурой данных, связанной с функцией анализируемой программы, которая абстрактно описывает эффект от выполнения данной функции в произвольном контексте вызова – побочные эффекты, возвращаемые значения, способы использования и изменения параметров и других доступных функции данных. Аннотация создается автоматически как результат анализа функции. Детекторы могут сохранять в аннотации необходимые им данные для последующего анализа и выдачи предупреждений.

Необходимые инструменту анализа спецификации стандартных библиотечных функций (например, языка Си) пишутся также на Си в виде заглушек, в которых указаны только полезные для анализа операции. Исходный код спецификаций транслируется компилятором LLVM-GCC, и результат поставляется в виде биткод-файлов. Эти файлы анализируются до начала анализа остальных файлов, в результате чего создаются аннотации соответствующих библиотечных функций. Например, для функции `printf`



достаточно указать, что ее первый аргумент всегда разыменовывается (это свойство полезно для поиска ошибок разыменования нулевого указателя; для указания этого свойства первый параметр явно разыменовывается в исходном коде спецификации) и является форматной строкой (это свойство важно для поиска уязвимостей форматной строки), а также то, что данная функция является представителем семейства функций обработки форматной строки и не меняет свои аргументы. Для пометки о том, что некоторый указатель есть форматная строка, используется специальная интерфейсная функция `sf_use_format`, предоставляемая детектором уязвимостей форматной строки (аналогично для остальных функций). Использование этой функции перехватывается в ходе анализа. Сама аннотация на языке Си выглядит, как показано на рис. 2.

```
int printf(const char *format, ...) {
   char d1 = *format;
   sf_use_format(format);

   sf_fun_printf_like(0);
   sf_fun_does_not_update_vargs(1);
}
```

*Рис. 2. Пример спецификации библиотечной функции.*

Статический анализ программы происходит в четыре этапа. На первом этапе все биткод-файлы считываются по-очереди в произвольном порядке, и строится общий граф вызовов программы. На втором (основном) этапе граф вызовов обходится в обратном топологическом порядке («снизу вверх»), так что (насколько позволяет отсутствие циклов в графе вызовов) каждая функция посещается после того, как были посещены все вызываемые из нее функции. Для каждой посещенной в этом порядке обхода функции программы выполняется внутрипроцедурный анализ. На третьем этапе выполняется специфический для Си++ анализ кода, исследующий взаимодействие методов классов. На четвертом этапе принимаются решения о выдаче либо исключении некоторых предупреждений на основании собранных в ходе основного анализа статистических данных и фрагментов исходного кода, соответствующих местам потенциальной выдачи предупреждений.

В ходе анализа конкретной функции инструменту доступно внутреннее представление лишь этой функции и данные о вызываемых ей функциях, присутствующие в виде аннотаций. В результате анализа функции создается ее аннотация, которая может использоваться в дальнейшем. Размер аннотаций ограничен сверху (при превышении лимита из аннотации исключается часть данных о поведении функции, в порядке уменьшения приоритета), поэтому в

ходе анализа каждой функции обрабатывается ее внутреннее представление и ограниченный дополнительный объем информации, не зависящий от полного размера программы (количества составляющих всю программу функций). Таким образом, анализ всей программы масштабируется линейно. Все аннотации функций по возможности хранятся в оперативной памяти, но при ее нехватке сериализуются на диск и считываются при необходимости. Сериализация позволяет снять ограничение на размер анализируемого кода и выполнять анализ программ размером в несколько 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 сериализации.

## 3.2. Внутрипроцедурный анализ

При анализе функции строится ее граф потока управления, после чего проводится потоково-чувствительный анализ, аналогичный анализу потока данных (т.е. для разных точек функции вычисляются разные значения атрибутов потока данных), при этом после нескольких проходов сверху вниз анализ завершается проходом снизу вверх. С каждой дугой графа потока управления ассоциируется *контекст* – информация о потоке данных, установленная для путей выполнения, проходящих через данную дугу. Например, после выполнения строки `if (x >= 4 && x <= 7)` в контексте подчиненного условному выражению ребра будет отражена информация о том, что значение переменной x находится в отрезке `[4,7]`. Анализ потока данных происходит путем «продвижения» контекста по дуге, входящей в инструкцию, через эту инструкцию и построения контекста на выходе из инструкции, основываясь на том, как инструкция манипулирует данными и памятью (при этом используются компактные структуры данных, не требующие чрезмерного дублирования данных). Для инструкции вызова продвижение контекста заключается в получении аннотации вызываемой функции (описывающей эффект от вызова вызываемой функции) и использовании ее для построения выходного контекста.

Контекст описывает взаимосвязь между элементами трех видов – абстрактными 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). Наконец, атрибуты описывают свойства значения, отслеживаемого некоторым идентификатором значения. Например, после обработки следующих инструкций контекст выглядит как показано ниже (в упрощенном виде):

```
s = 6;
*p = s;
```



| Ячейка памяти | `p` | `*p` | `s` |
|---|---|---|---|
| Идентификатор значения | `vid1` | `vid2` | |
| Атрибуты | `PT-TO: *p`<br>`NOT-NULL` | `POSITIVE` | |

*Элементами* этого контекста являются три ячейки памяти (`p`, `s` и `*p`, при этом `p` и `s` соответствуют явно указанным *переменным*), два идентификатора значений (один соответствует значению `6`, другой соответствует адресу ячейки `*p`), и 3 атрибута. Ячейка `p` хранит значение с идентификатором `vid1`, адресом ячейки `*p`. Это отражено в атрибутах `PT-TO` (куда указывает `p`) и `NOT-NULL` (если бы указатель был нулевым, то присваивание `*p=s` привело бы к ошибке времени выполнения). Ячейки `*p` и `s` хранят одно и то же значение `vid2` (этот факт устанавливается после обработки того же присваивания), которое имеет атрибут “положительный”. Кроме этого, например, детектор уязвимости переполнения буфера может отметить в атрибуте значения `vid2`, что оно равно в точности `6`.

## 3.3. Контекстная чувствительность и аннотации

При анализе функции не используется информация о том, откуда она могла быть вызвана. Такой вид межпроцедурного анализа называется *контекстно-нечувствительным*, так как разные точки вызова функции не различаются, и возможно появление информации о данных на “ложных” путях выполнения, которые входят в функцию в одной точке вызова и выходят в другой (анализ не может отбросить эти данные как невозможные).

Тем не менее, некоторая контекстная чувствительность оказывается возможной при использовании параметризованных аннотаций [17].

На рис. 3 аннотация функции `foo` отражает тот факт, что ее возвращаемое значение равно аргументу, через использование одного идентификатора значения для обеих этих ячеек памяти, при этом идентификатор значения не привязан к конкретным атрибутам.

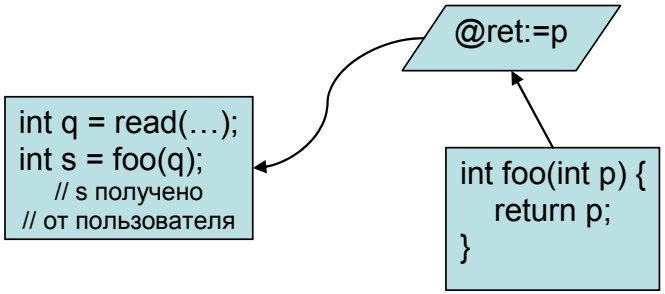


*Рис 3. Параметризованная аннотация.*

При продвижении контекста через вызов функции `foo` в примере используется фактический параметр `q`, значение которого получено от ввода пользователя (“испорчено”, т.е. не может быть использовано в потенциально опасных операциях без проверки на безопасность ввода). В ходе продвижения идентификатор значения для ячейки `s` устанавливается в идентификатор значения для ячейки `q`, тем самым значение `s` будет также помечено атрибутом “получено от пользователя”. При этом анализ вызова функции `foo` не влияет на ее аннотацию, и таким образом, информация об испорченности возвращаемого значения в данном вызове не будет влиять на анализ других вызовов той же функции. Такой анализ одновременно контекстно-чувствителен и позволяет распространять атрибуты через вызовы функций. Более подробно данная процедура описана в статье [9].

Процесс продвижения контекста через точку вызова функции называется *трансляцией аннотаций*, т.к. при этом элементы аннотации (по своей структуре аннотация очень схожа с контекстом) ставятся в соответствие элементам контекста в точке вызова – побочные эффекты вызываемой функции отражаются в контексте точки вызова, и новая информация, полученная об уже существующих в контексте вызова идентификаторах значений (указывающая, как фактические параметры вызова и доступные через них значения изменялись или использовались в вызываемой функции), должна быть объединена с имеющейся. В ходе трансляции возможно создание новых элементов контекста точки вызова.

На примере, показанном на рис. 4, аннотация функции `foo` содержит ячейки `z` и `*z`, у ячейки `z` есть идентификатор значения `v1`, имеющий атрибут, говорящий о том, что ячейка указывает на `*z`, а у ячейки `*z` есть идентификатор `v2`, имеющий атрибут, показывающий, что значение идентификатора равно четырем. В общем случае ячейки, идентификаторы и атрибуты составляют лес, в котором происходит поиск соответствующих элементов контекста в точке вызова и их создание в случае неудачи поиска. В нашем примере, контекст на ребре, входящем в инструкцию вызова, содержит



лишь ячейку `x`, которая связывается с `z`. Остальные элементы должны быть созданы в контексте исходящего из инструкции вызова ребра графа потока управления, где создается ячейка памяти для `*x` и соответственно устанавливается атрибут для идентификатора значения ячейки `x` (описывающий тот факт, что `x` указывает на `*x`), а также создается идентификатор значения для `*x`. После того, как все идентификаторы созданы, атрибуты идентификаторов `*x` и `*z` должны быть объединены (точнее, атрибуты идентификатора `v2` должны быть перенесены с объединением на атрибуты идентификатора значения соответствующей ячейки `*x`). Так как ранее ячейки `*x` не было в контексте, то информация из аннотации просто копируется в виде атрибута, показывающего, что значение `*x` равно четырем. Если бы, например, идентификатор `x` в точке вызова содержал атрибут “получен от пользователя”, атрибут идентификатора `z` об указывании `z` на `*z` был бы объединен с имеющимся.

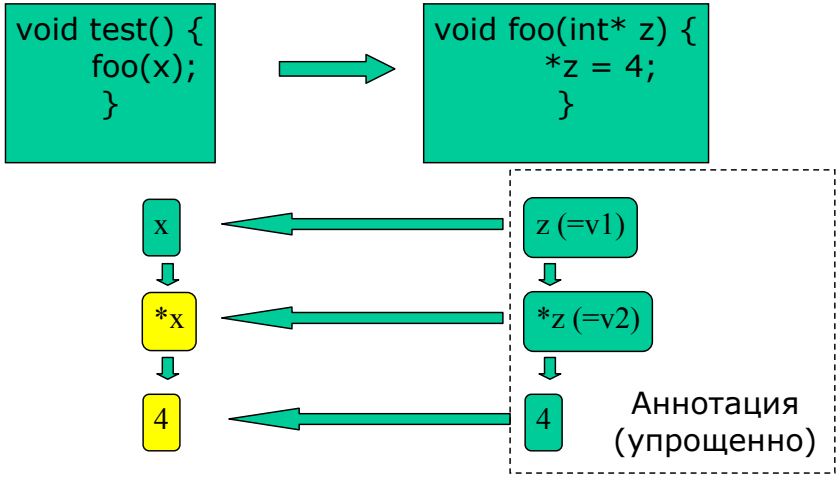


*Рис 4. Трансляция аннотаций.*

### 3.4. Использование атрибутов детекторами

Каждый детектор может вводить новые атрибуты для отслеживания необходимых свойств данных программы и выдачи предупреждений. Чтобы определить способы продвижения и слияния новых атрибутов, детекторы описывают обработчики этих событий, которые вызываются инфраструктурой анализа, и специальные функции манипуляции атрибутами, которые используются при написании спецификаций библиотечных функций (как на рис. 2). Обработчики событий имеют доступ к инструкции внутреннего

представления, через которую продвигается контекст, и всем элементам контекста, на которые ссылаются используемые в инструкции переменные и константы.

```
char buf[10];
buf[i]=0; // i∉[0,9] ⇒OVERFLOW
if(i>9) {
        // i ∈[10,+INF); i∉[0,9] ⇒OVERFLOW
        // ⇒ OVERFLOW
        exit(1);
}
```

*Рис 5. Продвижение атрибутов.*

На рис. 5 описаны шаги анализа, используемые для нахождения ситуации переполнения буфера. В ходе анализа приведенного фрагмента кода вводятся два атрибута. Первый атрибут описывает отрезок, внутри которого должно лежать значение, чтобы в данной точке программы не было переполнения буфера. Второй атрибут описывает отрезок, в котором гарантированно находится данное значение (т.н. *отрезок значения*). Во второй строке рассматриваемого примера первый атрибут устанавливается в отрезок `[0,9]`. Атрибут, связанный с идентификатором значения, распространяется до вызова функции `exit`. В точке вызова этой функции отрезок значения для этого идентификатора будет установлен в `[10,+INF)` как результат продвижения через условное выражение. Так как второй атрибут не содержит значений, находящихся в первом, детектор выдает предупреждение о переполнении буфера (на путях исполнения, проходящих через вызов функции `exit`, неизбежно происходит переполнение буфера).

В редких случаях детекторы могут обращаться к файлам с исходным кодом программы для отсечения ложных предупреждений с помощью информации, которая не сохранена в файлах с внутренним представлением. Например, некоторые предупреждения могут выдаваться в тексте макросов в местах, которые являются мертвым кодом. Знание о том, сгенерирован ли некоторый код с помощью макроса или написан программистом вручную, позволяет отсеять такие случаи.

### 3.5. Управление результатами анализа

После окончания анализа его результаты могут быть просмотрены пользователем через компонент в среде 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 функции программы, приводящая к ошибке). Для



удобства пользователя реализованы следующие дополнительные возможности:

- возможность запуска анализа некоторого проекта через среду Eclipse (а не только просмотр результатов);
- возможность размечать результаты анализа как истинные или ложные;
- поддержка истории запусков анализа. Эта часть инструмента позволяет сохранять результаты анализа и соответствующие исходные коды в базу данных и просматривать по мере необходимости. Кроме того, история запусков в совокупности с разметкой истинных и ложных предупреждений позволяет не выдавать пользователю повторно предупреждения, которые им были помечены как ложные (в т.ч. в случае, когда исходный код программы в ходе разработки был изменен, и предупреждение было выдано на другой строке).

Пример просмотра результатов анализа в Eclipse приведен на рис. 6.

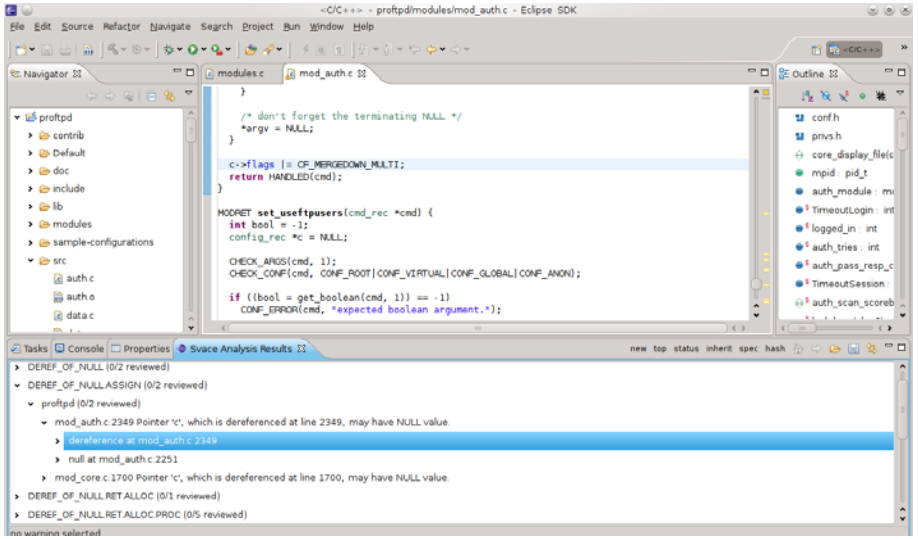


*Рис.6. Просмотр результатов анализа в среде Eclipse.*

## *4. Тестирование компонентов инструмента*

Детекторы тестируются с помощью вручную написанных на языке Си тестов. Тестирование заключается в запуске анализа на данном файле или наборе файлов. Результат запуска тестов определяется функциями-утверждениями, вызываемыми в ходе теста, обычно о том, что предупреждение определенного

типа было выдано (или не выдано) на определенной строке файла. Все тесты могут быть запущены автоматически.

Например, на рис. 7 приведен тест для предупреждения `DEREF_OF_NULL` (разыменование указателя, имеющего нулевое значение). Этот тест проверяет работу межпроцедурного анализа с побочными эффектами, выражающимися в изменении значений глобальных переменных. Функция `get_var` всегда возвращает значение глобальной переменной `var`. В функции `test` первый условный оператор выполняется, когда функция `get_var` возвращает (а переменная `var` имеет) нулевое значение, и предупреждение `DEREF_OF_NULL` должно быть выдано при разыменовании этого значения. Требование о выдаче выражается вызовом функции-утверждения `sf_assert_thrown`, которой передается имя предупреждения. Выдача ожидается на следующей после вызова функции строке. Аналогично, второй условный оператор выполняется при ненулевом возвращаемом значении, и предупреждение не должно быть выдано, что выражается вызовом функции `sf_assert_not_thrown`.

```
#include "specfunc.h"

int* var;

int* get_var(void) {
   return var;
}

void test(int *p, int x, int y)
{
   if(!get_var()) {
      sf_assert_thrown(DEREF_OF_NULL);
      x = *get_var();
   }
   if(get_var()) {
      sf_assert_not_thrown(DEREF_OF_NULL);
      x = *get_var();
   }
}
```

Рис.7. Пример теста.

## *5. Тестирование качества анализа*

Помимо небольших, написанных вручную тестов, для проверки качества общей инфраструктуры анализа и отдельных детекторов, а также для поиска



часто встречающихся ситуаций и разработки новых эвристик используются наборы пакетов программ с открытым исходным кодом. Рассматриваемые пакеты были автоматически собраны для поддерживаемых целевых платформ и содержат в общей сложности около 6 миллионов строк кода на Си и Си++.

Нужно заметить, что большая часть работы над инструментом заключается как раз в обработке результатов анализа больших пакетов, поиске ошибок и идентификации возможных улучшений алгоритмов их поиска. Без этой работы ценность основной инфраструктуры анализа невысока, так как большинство деталей анализа, достигающих конечного пользователя, раскрываются отдельными детекторами при выдаче предупреждений, а они в свою очередь тем точнее и надежнее, чем больше примеров исходного кода было проработано при их создании.

В ходе оценки качества анализа необходимо выработать критерии выбора предупреждений для анализа. На большом объеме исходного кода, как правило, обнаруживается большое количество предупреждений, и просмотреть все выданные предупреждения и для каждого из них вынести суждение об его истинности или ложности является чрезмерно трудоемкой задачей (на полном наборе тестовых пакетов программ может быть выдано до нескольких тысяч предупреждений одного типа). Мы используем псевдослучайную выборку фиксированного количества предупреждений каждого типа (или группы схожих типов), стабильную между различными запусками анализа. Таким образом, количество работы, необходимой для оценки результатов, ограничено, независимо от количества проанализированного исходного кода. Случайность выборки делает результаты оценки репрезентативными для проанализированных пакетов программ, а большой объем анализируемого кода делает результаты оценки репрезентативными среди возможных (не входящих в тестовый набор) пакетов программ.

При анализе каждое предупреждение обычно классифицируется как истинное либо ложное, но дополнительно возникает два случая: а) предупреждение истинное, так как инструмент нашел ровно ту ситуацию в исходном коде, которая была задана при поиске, но из кода видно, что такая ситуация не представляет ошибки и допущена намеренно; б) неясно, является ли предупреждение истинным или ложным, либо из-за нечеткости формулировки предупреждения, либо из-за сложности анализируемого исходного кода.

Полные результаты оценки разработанного инструмента статического анализа не могут быть раскрыты как полученные в ходе работы по коммерческим контрактам. В целом можно сказать, что из 15 групп оцениваемых предупреждений средняя доля истинных предупреждений составила 67%, от 25% до 100%. Более высокая доля истинных предупреждений (50-85%, в зависимости от типа) наблюдается у групп предупреждений о возможном разыменовании нулевого указателя и небезопасном использовании испорченных (контролируемых внешним вводом) данных. Более низкая доля

(40-60%, в зависимости от типа) у предупреждений об утечках памяти, использовании неинициализированных данных или освобожденной памяти, и переполнении буфера.

## 6. Заключение

Инструменты автоматического статического анализа являются полезными для задачи обеспечения безопасности программного обеспечения – с помощью глубокого межпроцедурного анализа можно достичь хорошей доли истинных предупреждений (40-80%) при приемлемом времени анализа (несколько часов для миллионов строк кода). Институтом системного программирования РАН разработан и успешно опробован в коммерческих проектах такой инструмент, качество которого не уступает лучшим коммерческим продуктам в данной области. Залогом успеха является кропотливая работа по просмотру реальных промышленных программ с открытым исходным кодом и доработке инструмента анализа на основе его результатов на этих программах, для достижения приемлемого компромисса между качеством выдаваемых результатов и их количеством. В планах работы над инструментом – поддержка других языков программирования (в первую очередь Java), языков Web-программирования (Javascript), развитие интерфейса пользователя (поддержка удаленного и распределенного анализа, поддержка командной работы), а также предоставление доступа к инструменту как к сервису в рамках систем облачных вычислений.

**Список литературы**


[1] С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова. О некоторых задачах анализа и трансформации программ. Труды ИСП РАН, №5, с.7-41, 2004.

[2] О.Р. Маликов, А.А. Белеванцев. Автоматическое обнаружение уязвимостей в программах. Материалы конференции «Технологии Майкрософт в теории и практике программирования», Москва, 2004.

[3] О.Р. Маликов. Автоматическое обнаружение уязвимостей в исходном коде программ. Известия ТРТУ, №4, с. 48-53, 2005.

[4] В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды ИСП РАН, №9, с. 51-57, 2006.

[5] О.Р. Маликов, В.С. Несов. Автоматический поиск уязвимостей в больших программах. Известия ТРТУ, Тематический выпуск «Информационная безопасность», №7 (62), с. 114-120, 2006.

[6] В.С. Несов. Использование побочных эффектов функций для ускорения автоматического поиска уязвимостей в программах. Известия ЮФУ. Технические науки. Тематический выпуск «Информационная безопасность». Таганрог: Изд-во ТТИ ЮФУ, 2007. № 1(76), с. 134-139.

[7] В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. ун-та, 2008, с.107.

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

[9] Vladimir Nesov. Automatically Finding Bugs in Open Source Programs. Electronic Communications of the EASST 20. ISSN 1863-2122, 2009.

[10] Компилятор GCC. http://gcc.gnu.org

[11] Компиляторная инфраструктура LLVM. http://llvm.org

[12] Инструмент Coverity Prevent. http://www.coverity.com/library/pdf/coverity_prevent.pdf

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

[14] Среда Eclipse. http://www.eclipse.org/

[15] P. Godefroid. The Soundness of Bugs is What Matters (position statement). In BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools), 2005.

[16] P. Emanuelsson & U. Nilsson. A Comparative Study of Industrial Static Analysis Tools (extended version). Tech. rep., Linköping University, 2008.

[17] D. Liang & M. J. Harrold. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In SAS '01: Proceedings of the 8th International Symposium on Static Analysis, pp. 279-298, London, UK, Springer-Verlag, 2001.