

Десятая независимая научно-практическая  
конференция «Разработка ПО 2014»

23 - 25 октября, Москва



# Использование статического анализатора Clang для обнаружения проблем в исходном коде

**Павлов Евгений**

Samsung R&D Institute Russia

# Что такое статический анализ?

**Статический анализ** кода – это анализ программного обеспечения без реального выполнения используемых программ.

# Для чего используется статический анализ?

Статический анализ позволяет:

- **обнаруживать ошибки** в исходном коде;
- **собирать метрику** программного обеспечения;
- **проверять соответствие** программы определенным **свойствам**.

# Какие инструменты для статического анализа используются в Samsung?

- **Coverity Code Advisor** (C/C++, C#, Java)



- **Klocwork Insight** (C/C++, C#, Java)



- **FindBugs** (Java, open source)



- **Clang Static Analyzer** (C/C++/Objective-C, open source)



- ...

# Зачем Samsung открытый статический анализатор?

- Легче модифицировать анализатор под нужды компании
- Проще разрабатывать новые правила
- Проще интегрировать в процессы разработки

# Основные особенности статического анализатора Clang

- Входит в компиляторную инфраструктуру LLVM (open source)
- Использует символическое выполнение для анализа
- Поддерживает расширение набора правил для обнаружения новых типов дефектов



# Типы обнаруживаемых дефектов

Ошибки работы  
с памятью

Нарушение  
безопасности

Использование  
небезопасного  
UNIX API

Переполнение  
буфера

Утечка ресурсов

Разыменованние  
нулевого  
указателя

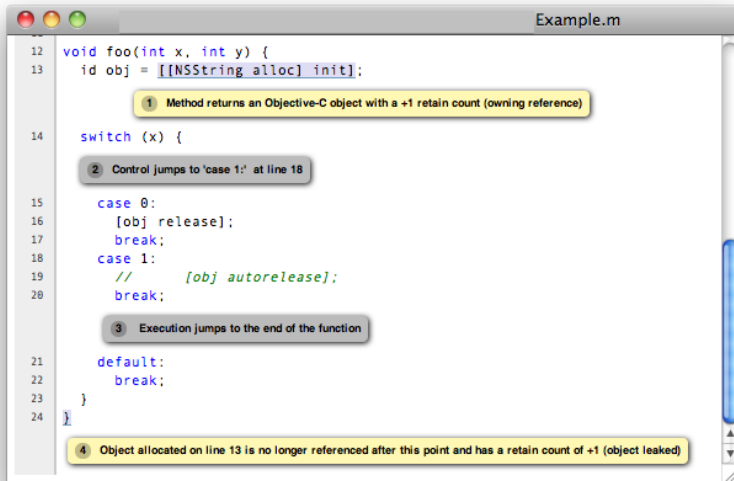
Неправильное  
использование  
OSX и iOS API

Недостижимый  
код

Others...

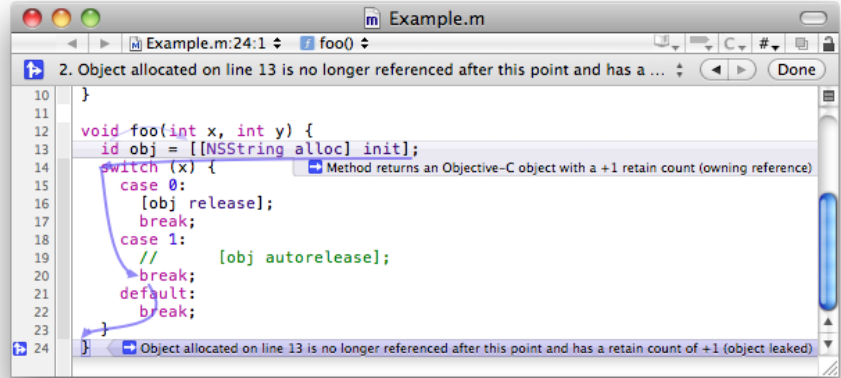
# Варианты использования

- Интегрирован в Xcode



The screenshot shows the Xcode IDE with a file named 'Example.m'. The code defines a function 'foo' that takes two integers 'x' and 'y'. It allocates an Objective-C string object 'obj' on line 13. A switch statement follows, with three cases: 'case 0' where 'obj' is released; 'case 1' where 'obj' is autoreleased; and a 'default' case. Three callouts explain the control flow: 1. 'Method returns an Objective-C object with a +1 retain count (owning reference)' pointing to the allocation. 2. 'Control jumps to 'case 1:' at line 18' pointing to the switch statement. 3. 'Execution jumps to the end of the function' pointing to the end of the function. A fourth callout at the bottom states: 'Object allocated on line 13 is no longer referenced after this point and has a retain count of +1 (object leaked)'.

```
12 void foo(int x, int y) {
13     id obj = [[NSString alloc] init];
14     switch (x) {
15     case 0:
16         [obj release];
17         break;
18     case 1:
19         // [obj autorelease];
20         break;
21     default:
22         break;
23     }
24 }
```



The screenshot shows the Xcode IDE with a file named 'Example.m'. The code is the same as in the previous screenshot. A warning message is displayed at the top: '2. Object allocated on line 13 is no longer referenced after this point and has a ...'. A blue arrow points from the warning to the allocation line in the code. The code is the same as in the previous screenshot.

- Утилита командной строки scan-build



# Что было сделано в SRR?

- Разработано 24 новых правил для обнаружения новых типов дефектов
- Улучшен механизм перехвата вызовов компилятора
- Поправлены баги, улучшен движок анализатора

Некоторые из этих улучшений уже отправлены в сообщество, некоторые в процессе подготовки

# Результаты: Android AOSP 4.2.1\_r1

| Имя правила                     | Количество срабатываний | Процент истинных срабатываний |
|---------------------------------|-------------------------|-------------------------------|
| alpha.different.MissingBreak    | 23                      | 78%                           |
| alpha.different.BadAssert       | 34                      | 76%                           |
| alpha.different.IntegerOverflow | 104                     | 77%                           |



# Результаты: Tizen 2.2

| Имя правила                     | Количество срабатываний | Процент истинных срабатываний |
|---------------------------------|-------------------------|-------------------------------|
| alpha.different.MissingBreak    | 41                      | 60.00%                        |
| alpha.different.BadAssert       | 15                      | 100.00%                       |
| alpha.different.IntegerOverflow | 183                     | 74.00%                        |

# Преимущества и недостатки по сравнению с коммерческими инструментами



# Полезная информация

- Разработка нового правила в среднем занимает от 0,5 до 2 М/М
- Большая часть времени разработки уходит на улучшение точности правила после прогона на тестовых приложениях
- При разработке новых правил лучше почаще общаться с сообществом LLVM

# Полезные ссылки

- ◉ <http://llvm.org/>
- ◉ [http://clang-analyzer.llvm.org/checker\\_dev\\_manual.html](http://clang-analyzer.llvm.org/checker_dev_manual.html)
- ◉ [http://clang-analyzer.llvm.org/open\\_projects.html](http://clang-analyzer.llvm.org/open_projects.html)
- ◉ <http://lists.cs.uiuc.edu/mailman/listinfo/cfe-commits>
- ◉ <http://lists.cs.uiuc.edu/mailman/listinfo/cfe-dev>

Спасибо за внимание!