Методы инструментирования Си-программ для поиска ошибок с помощью статического анализа кода

Новиков Евгений
аспирант ИСП РАН 3-го года обучения

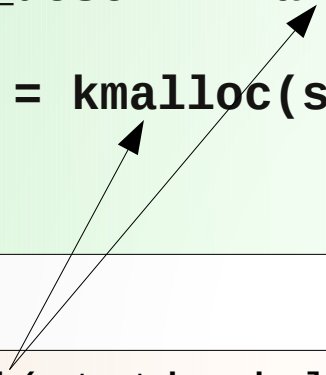
Научный руководитель
д.ф.-м.н., проф. А.К.Петренко

Аспектно-ориентированное программирование (1)

```
static int download_fw(struct edgeport_serial *serial)
{
  ...
  ti_manuf_desc = kmalloc(sizeof(*ti_manuf_desc), GFP_KERNEL);
  if (!ti_manuf_desc)
      return -ENOMEM;
  ...
  rom_desc = kmalloc(sizeof(*rom_desc), GFP_KERNEL);
  if (!rom_desc)
      return -ENOMEM;
  ...
}
```

Аспектно-ориентированное программирование (2)

```
static int download_fw(struct edgeport_serial *serial)
{
  ...
  ti_manuf_desc = kmalloc(sizeof(*ti_manuf_desc), GFP_KERNEL);
  ...
  rom_desc = kmalloc(sizeof(*rom_desc), GFP_KERNEL);
  ...
}
```



```
after: call(static inline void *kmalloc(size_t, gfp_t)) {
  if (!result)
    return -ENOMEM;
}
```

Аспектно-ориентированное программирование (3)

- Идея АОП не получила широкого распространения на практике
  - По мере своего развития языки программирования предоставляли больше средств для декомпозиции программ на модули
  - Для большинства языков программирования нет достаточно хорошей реализации АОП
  - Использование АОП приводит к дополнительным накладным расходам как во время сборки программ, так и во время их выполнения

Типовые общие ошибки в Си программах

Правило: нельзя разыменовывать NULL указатель

```
int var, *ptr = NULL;
...
var = *ptr; // Ошибка!
```

Типовые специфичные ошибки в Си программах

Правило: при удержании спин блокировки необходимо вызывать функции выделения памяти с флагом GFP_ATOMIC

```
spin_lock(lock);
buf = kmalloc(size, GFP_KERNEL); // Ошибка!
spin_unlock(lock);
```

Типовые синхронизационные ошибки в Си программах

Правило: в программе не должно быть взаимных блокировок

```
mutex_lock(lock_a);
mutex_lock(lock_b);
```

```
mutex_lock(lock_b); // Ошибка!
mutex_lock(lock_a); // Ошибка!
```

Типовые ошибки в драйверах ядра ОС Linux

- Анализ изменений в драйверах стабильных версий ядра ОС Linux с 26 октября 2010 года по 26 октября 2011 года

<table>
<tr><td colspan="5">Изменения в драйверах (1503)</td></tr>
<tr><td>Расширение функциональности (321 ~ 20%)</td><td colspan="4">Исправление ошибок (1182 ~ 80%)</td></tr>
<tr><td></td><td>Нетиповые ошибки (833 ~ 70%)</td><td colspan="3">Типовые ошибки (349 ~ 30%)</td></tr>
<tr><td></td><td></td><td>Общие (102 ~ 30%)</td><td>Специфичные (176 ~ 50 %)</td><td>Синхронизационные (71 ~ 20 %)</td></tr>
</table>

Проблемы поиска типовых специфичных ошибок

- Многообразие интерфейсов различных библиотек
  - Для обнаружения 176 типовых специфичных ошибок в драйверах Linux требуется 104 различных правила, причем для 60% ошибок достаточно 30% правил
  - Всего для драйверов Linux требуется порядка 1500 правил, из которых 500 правил позволят обнаружить примерно 60% всех типовых специфичных ошибок
- Изменение интерфейсов библиотек и их реализаций

Статический анализ кода

- Статический анализ кода позволяет
  - Находить редко встречающиеся ошибки
  - При некоторых ограничениях доказывать полное отсутствие ошибок некоторого вида
- Реализованы различные подходы статического анализа кода
  - «Легковесные» (Coverity, Klockwork Insight, Svace, ...)
  - «Тяжеловесные» (BLAST, CPAchecker, LLBMC, Predator, Treader, ...)
- Для поиска ошибок с помощью статического анализа кода нужно некоторым формальным образом задавать проверяемые правила

Существующие способы формального задания проверяемых типовых правил

- Ручное инструментирование исходного кода Си-программ
  - Позволяет задавать правила наиболее точным образом
  - Неприменимо для индустриальных программ
- Разработка детекторов для инструмента статического анализа кода
  - Позволяет использовать возможности соответствующего инструмента наиболее эффективным образом
  - Детекторы непереиспользуемы между различными инструментами
- Использование специального языка спецификаций для проверки интерфейса
  - Формализация правил в виде спецификаций независимым от инструмента статического анализа способом
  - Автоматическое инструментирование Си-кода на основе спецификаций
  - Ограничения по формальному заданию правил

Цель работы

Разработать метод инструментирования исходного кода Си-программ для поиска нарушений произвольных типовых правил, в особенности специфичных, с помощью различных инструментов статического анализа кода

Предложенный подход

- Использовать реализацию аспектно-ориентированного программирования
  - Формализация правил в виде аспектов независимым от инструмента статического анализа способом
  - Автоматическое инструментирование Си-кода на основе аспектов
  - Широкие возможности по формальному заданию правил

Существующие реализации АОП для языка Си

Характеристики реализации АОП для языка Си	ACC	InterAspect	SLIC
Поддержка традиционных средств АОП	+	±	±
Поддержка средств АОП, специфичных для языка Си	-	-	-
Поддержка языка Си с расширениями GCC	±	+	±
Возможность задания модельных состояния и функций	-	+	±
Выход на языке Си	±	-	±
Возможность расширения	±	+	-

Пример типового специфичного правила

Правило: при удержании спин блокировки необходимо вызывать функции выделения памяти с флагом GFP_ATOMIC

```
spin_lock(lock);
buf = kmalloc(size, GFP_KERNEL); // Ошибка!
spin_unlock(lock);
```

Формализация правила (1)

Модельные состояние и функции
(1-я часть аспектного файла)

```
#include <linux/slab.h> // Определения gfp_t и GFP_ATOMIC

static int spinlock_held = 0;
void model_spin_lock() {
  spinlock_held = 1;
}
void model_spin_unlock() {
  spinlock_held = 0;
}

void check_flags(gfp_t flags) {
  if (spinlock_held && flags != GFP_ATOMIC) {
    ERROR: goto ERROR;
  }
}
```

Формализация правила (2)

Привязка модельных функций к использованиям интерфейса ядра ОС Linux (2-я часть аспектного файла)

```
around: call(static inline void spin_lock(..)) {
  model_spin_lock();
}
around: call(static inline void spin_unlock(..)) {
  model_spin_unlock();
}

before: call(static inline void *kmalloc(.., gfp_t flags)){
  check_flags(flags);
}
```

Пример нарушения правила

```
spin_lock(lock);
buf = kmalloc(size, GFP_KERNEL); // Ошибка!
spin_unlock(lock);
```

Инструментированный код

```
model_spin_lock();
buf = aux_kmalloc(size, GFP_KERNEL); // Ошибка!
model_spin_unlock();
...
static inline void *aux_kmalloc(size_t size, gfp_t flags) {
  check_flags(flags);
  return kmalloc(size, flags);
}
…
#include <linux/slab.h>
int spinlock_held = 0;
void model_spin_lock() {
  spinlock_held = 1;
}
void model_spin_unlock() {
  spinlock_held = 0;
}
void check_flags(gfp_t flags) {
  if (spinlock_held && flags != GFP_ATOMIC) {
    ERROR: goto ERROR;
  }
}
```

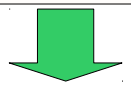
Процесс инструментирования

- Инструментирование выполняется автоматически с помощью C Instrumentation Framework (CIF) за 5 этапов
  - «Аспектное препроцессирование»
  - «Подготовка кода»
  - «Инструментирование макросов»
  - «Инструментирование»
  - «Компиляция»

«Аспектное препроцессирование»

```
@include <kernel-model/spinlock.aspect>

before: call(static inline void *kmalloc(.., gfp_t flags)) {
  check_flags(flags);
}
```



```
around: call(static inline void spin_lock(..)) {
  model_spin_lock();
}
around: call(static inline void spin_unlock(..)) {
  model_spin_unlock();
}

before: call(static inline void *kmalloc(.., gfp_t flags)) {
  check_flags(flags);
}
```

«Подготовка кода»

```
before: file("$this") {
  void model_spin_lock();
}
```

```
#include <linux/spinlock.h>
#include <linux/module.h>
```

```
void model_spin_lock();

#include <linux/spinlock.h>
#include <linux/module.h>
```

«Инструментирование макросов»

```
around: define(mutex_lock(lock)) {
  model_mutex_lock(lock);
}
```

```
mutex_lock(&driver_lock);
```

```
model_mutex_lock(&driver_lock);
```

«Инструментирование» (1)

```
before: call(static inline void *kmalloc(.., gfp_t flags)) {
  check_flags(flags);
}
```

```
buf = kmalloc(size, GFP_KERNEL);
```

```
buf = kmalloc(size, GFP_KERNEL);
...
static inline void *aux_kmalloc(size_t size, gfp_t flags) {
  check_flags(flags);
  return kmalloc(size, flags);
}
```

«Инструментирование» (2)

```
before: get(struct mutex *i_mutex) {
  check_mutex(i_mutex);
}
```

```
j_mutex = i_mutex;
```

```
j_mutex = i_mutex;
...
struct mutex *aux_i_mutex(struct mutex *i_mutex) {
  check_mutex(i_mutex);
  return i_mutex;
}
```

«Инструментирование» (3)

```
after: introduce(struct mutex) {
  int islocked;
}
```

```
struct mutex { … };
```

```
struct mutex {
  …
  int islocked;
};
```

«Компиляция»

- Оригинальные конструкции из исходного кода программы связываются с вспомогательными функциями
- На выходе генерируется инструментированный Си-код

Результаты

Формализовано правил проекта верификации драйверов ОС Linux	44 (из ~500)
Инструментирован исходный код драйверов ядра ОС Linux версий от 2.6.31.6 до 3.7-rc1	>95%
Использовано инструментов статического анализа кода	BLAST и CPAchecker

Сравнение с существующими реализациями АОП для языка Си

Характеристики реализации АОП для языка Си	ACC	InterAspect	SLIC	**CIF**
Поддержка традиционных средств АОП	+	±	±	±
Поддержка средств АОП, специфичных для языка Си	-	-	-	±
Поддержка языка Си с расширениями GCC	±	+	±	+
Возможность задания модельных состояния и функций	-	+	±	+
Выход на языке Си	±	-	±	±
Возможность расширения	±	+	-	+

Дальнейшие планы

- Проанализировать, что нужно для формализации оставшихся (из 104) типовых специфичных правил с помощью предложенного подхода
- Доработать C Instrumentation Framework
  - Добавить поддержку наиболее важных возможностей среди выявленных
  - Реализовать неподдерживаемые традиционные и специфичные для языка Си средства АОП
  - Исправить ошибки в реализации
- Использовать CIF в качестве общецелевой реализации АОП для языка Си для решения других задач

Спасибо!

http://forge.ispras.ru/projects/cif (GPL v3)