LinuxTesting.org

Using Aspect-Oriented Programming for Preparing C Programs for Static Verification

Evgeny Novikov
joker@ispras.ru

ISP RAS

Institute for System Programming of the Russian Academy of Sciences

Bugs in C Programs

Rule: might sleep memory allocation should be forbidden when spinlock is held. Therefore, having a spinlock acquired, you should use GFP_ATOMIC flag when calling memory allocation functions

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

Static Verifiers

- A lot of tools implementing different static verification approaches (BLAST, CPAchecker, CBMC, etc.)
- The most of tools are reachability verifiers

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

Challenge

How to prepare C programs for verification against different rules by means of different static verifiers?

Related Approaches

- Manual instrumentation of programs
  - Is the most accurate approach
  - Is not well scalable for industrial software
- Checkers embedded in verifiers
  - Using capabilities of analysis tools in the most efficient way
  - Can not be easily shared between different tools
- Using restricted implementations of aspect-oriented programming (**AOP**) for the C programming language
  - Formalizing rules in the verifier independent way in form of aspects
  - Automatic instrumentation of programs on the basis of aspects
  - Does not support complex instrumentation of programs

Approach Suggested

- Using general purpose AOP implementation specifically designed for preparing C programs for static verification
  - Formalizing rules in the verifier independent way in form of aspects
  - Automatic instrumentation of programs on the basis of aspects
  - Supports complex instrumentation of programs

Rule Example

Rule: might sleep memory allocation should be forbidden when spinlock is held. Therefore, having a spinlock acquired, you should use GFP_ATOMIC flag when calling memory allocation functions

Formalizing Rule (1)

Model state and functions

```
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;
  }
}
```

Formalizing Rule (2)

Relation between program and model functions (aspect)

```
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);
}
```

Program Example

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

Program Instrumentation (1)

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

Program Instrumentation (2)

- Performed automatically with help of C Instrumentation Framework (**CIF**) in 5 stages
  - Aspect preprocessing
  - File preparation
  - Macro instrumentation
  - Instrumentation
  - Compilation

Aspect preprocessing

Preprocesses aspect files as well as standard C files but with „@“ directives instead of „#“ ones

```
@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);
}
```

File preparation

Modifies instrumented file as a plain text

```
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>
```

Macro instrumentation

Affects the standard preprocessing process

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

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

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

Instrumentation (1)

Creates auxiliary functions for function calls and definitions

```
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);
}
```

Instrumentation (2)

Creates auxiliary functions for simple object manipulations

```
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;
}
```

Instrumentation (3)

Extends type declarations (structures, unions, enums)

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

+

```
struct mutex { … };
```

↓

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

Compilation

- Finally binds original constructions with auxiliary ones and produces output by means of back-end specified
    - **Source code**
    - Assembler
    - Object
    - Executable

Summary

- Supported common well-known AOP features
  - Instrumentation of function calls and definitions
  - Instrumentation of simple object manipulations
  - Instrumentation of complex object manipulations (e.g. pointers dereferencing, fields getting, etc.)
  - Instrumentation of type declarations
  - Instrumentation before, after and around corresponding points
  - Processing of several befores, afters and arounds for the same point
  - Wildcard „..“ for matching any parameters
  - Wildcard „$“ for matching any type and name
  - Reflective information on called funciton arguments, return types, etc.
- Supported specific AOP features
  - Aspect preprocessing
  - File preparation
  - Macro instrumentation
  - State definition
  - Reflective information on GCC attributes
  - „Memset“ functionality
  - Non-instrumentation quinformation requests
- Input and output limitations
  - Programs in C with GNU extensions as input
  - C source code as output
  - Binary as output

Results

Formalized rules o f the Linux Driver Verification project	44 (of ~200)
Instrumented drivers of Linux kernels from 2.6.31.6 up to 3.6-rc4	>90%
Used static verifiers	BLAST and CPAchecker

Future Plans

- Support more common well-known AOP features, e.g. instrumentation of complex object manipulations
- Support more specific AOP features, e.g. non-instrumentation information requests
- Extend applications to user-space programs

LinuxTesting.org

Thank you!

Evgeny Novikov
joker@ispras.ru

http://forge.ispras.ru/projects/cif

ISP RAS

Institute for System Programming of the Russian Academy of Sciences