Enforcing the Use of API Functions in Linux Code

Julia Lawall

Joint work with
Gilles Muller (EMN/INRIA) and Nicolas Palix (DIKU)

April 16, 2009

Our context: bug finding

Traditional automatic bug-finding tools:

- Start with patterns, either canned or user-provided.
- Scan code for matches.
- Report matches as possible bugs.

Examples:

- Memory leaks.
- Null pointer dereferences.
- Confusion of boolean and bit values (!x&y).
- etc.

Our bug-finding tool

Coccinelle: program matching and transformation for C code.

Guiding principle:

- Matching/transformation rules should look like C code.
- Or, more precisely, like C code patches.

Example:

```
@@
expression E;
constant C;
@@
- !E & C
+ !(E & C)
```

A more complex example

Memory leaks:

```
@r exists@
local idexpression x; statement S;
expression E; identifier f; position p1,p2;
@@

x@p1 = kmalloc(...);
...
if (x == NULL) S
<... when != x
     when != if (...)  <+...x...+>
x->f = E
...>
(
 return <+...x...+>;
|
 return@p2 ...;
)
```

But are these the bugs we should be looking for, or is there something else?

Our (very preliminary) contribution

Observation: Linux header files define many small API functions.

- These functions raise the abstraction level, and impose type safety.
- Sometimes these functions are used, sometimes not.

Goal:

- Find header-file functions that follow a common pattern.
- Generate bug-finding rules based on header-file function definitions.

Example

Some constant definitions:

```
#define LM_ST_UNLOCKED   0
#define LM_ST_EXCLUSIVE  1
#define LM_ST_DEFERRED   2
#define LM_ST_SHARED     3
```

Later, in the same file, some function definitions:

```
static inline int gfs2_glock_is_held_excl(struct gfs2_glock *gl) {
  return gl->gl_state == LM_ST_EXCLUSIVE;
}

static inline int gfs2_glock_is_held_dfrd(struct gfs2_glock *gl) {
  return gl->gl_state == LM_ST_DEFERRED;
}

static inline int gfs2_glock_is_held_shrd(struct gfs2_glock *gl) {
  return gl->gl_state == LM_ST_SHARED;
}
```

Use and non-use of these functions

A use:

```
if (gfs2_assert_withdraw(sdp, gfs2_glock_is_held_excl(ip->i_gl)))
  goto out;
```

A non-use:

```
BUG_ON(gl->gl_state != LM_ST_EXCLUSIVE);
```

Using Coccinelle to impose the use of the API

A semantic patch to convert non-uses to uses:

```
@@
struct gfs2_glock *gl;
@@

(
- gl->gl_state == LM_ST_EXCLUSIVE
+ gfs2_glock_is_held_excl(gl)
|
- gl->gl_state == LM_ST_DEFERRED
+ gfs2_glock_is_held_dfrd(gl)
|
- gl->gl_state == LM_ST_SHARED
+ gfs2_glock_is_held_shrd(gl)
)
```

Using Coccinelle to impose the use of the API

Another semantic patch to convert non-uses to uses:

```
@@
struct gfs2_glock *gl;
@@

(
- gl->gl_state != LM_ST_EXCLUSIVE
+ !gfs2_glock_is_held_excl(gl)
|
- gl->gl_state != LM_ST_DEFERRED
+ !gfs2_glock_is_held_dfrd(gl)
|
- gl->gl_state != LM_ST_SHARED
+ !gfs2_glock_is_held_shrd(gl)
)
```

Results

	excl	dfrd	shrd
Original calls	1	0	0
Introduced pos calls	2	1	1
Introduced neg calls	2	0	0

Remaining references to the `gl_state` field

2	initialization
4	comparison to another expression
10	comparison to a constant (`LM_ST_UNLOCKED`)
3	other

Remaining references to the `LM_ST` constants

92	as function arguments
13	as case labels
12	as assignments
30	as comparisons

Assessment

This is only one example of a common phenomenon.

We might notice such inconsistencies in the use of constants and functions and address them, by hand or by writing rules.

But it would be better to fix them up front, without having to notice them one by one.

Goal: Automatically create rules to enforce API usage.

- Or perhaps automatically create rules to eliminate APIs.

Issues

Different kinds of functions can be used in different kinds of ways.

A potential usage site does not always have the same form as the function definition.

- For our equality tests, the potential usage site does not include `return`.

There may be many kinds of potential usage sites for a given function.

- In our examples, both equalities and inequalities are potential usage sites.

Solution: The user describes a class of functions, and a usage that is common to that class.

Example:

A rule to find a kind of small function definition:

```
@@ identifier f; expression E;
constant C; parameter list ARGS; @@

f(ARGS) { return (E == C); }
```

A matching function definition

```
static inline int gfs2_glock_is_held_excl(struct gfs2_glock *gl) {
  return gl->gl_state == LM_ST_EXCLUSIVE;
}
```

A rule we hope to generate:

```
@@ struct gfs2_glock *gl; @@

- gl->gl_state == LM_ST_EXCLUSIVE
+ gfs2_glock_is_held_excl(gl)
```

Generative rule notation

```
@pat@
identifier f; expression E;
constant C; parameter list ARGS;
@@

f(ARGS) { return (E == C); }

@generative@
identifier pat.f; expression pat.E;
constant pat.C; expression list pat.ARGS;
@@

(
- (E == C)
+ f(ARGS)
|
- (E != C)
+ !f(ARGS)
)
```

Issues

The generated rule should not be as generic as the generative rule

- `E == C` is too generic.

The generated rule should not be as specific as the matched function definition

- `gl->gl_state == LM_ST_EXCLUSIVE` is too specific.

Solution: The metavariables should be the matches function's parameters.

- These are the point of variability of the matched function's behavior.

Generation process

- Match a function definition.
- Generate a rule having:
  - Metavariables: The parameters of the matched functions.
  - Body: The body of the generative rule instantiated according to the generative rule's metavariable bindings.
- Also generate:
  - A rule to check that the header file is included.
  - A rule to prevent transforming the code in the header-file definition into a call to itself.

Walid has proposed a more elegant generation process relying on metavariables that are subdivided into contexts.

Example

```
@pat@
identifier f; expression E;
constant C; parameter list ARGS;
@@

f(ARGS) { return (E == C); }
```

with metavariable bindings:

- $E \equiv$ `gl->gl_state`
- $C \equiv$ `LM_ST_EXCLUSIVE`
- $f \equiv$ `gfs2_glock_is_held_excl`
- $ARGS \equiv$ `struct gfs2_glock *gl`

```
@generative@
identifier pat.f; expression pat.E;
constant pat.C; expression list pat.ARGS;
@@

(
- (E == C)
+ f(args)
|
- (E != C)
+ f(ARGS)
)
```

Example

```
@pat@
identifier f; expression E;
constant C; parameter list ARGS;
@@

f(ARGS) { return (E == C); }
```

with metavariable bindings:

- $E \equiv$ `gl->gl_state`
- $C \equiv$ `LM_ST_EXCLUSIVE`
- $f \equiv$ `gfs2_glock_is_held_excl`
- $ARGS \equiv$ `struct gfs2_glock *gl`

```
@generative@
identifier pat.f; expression pat.E;
constant pat.C; expression list pat.ARGS;
@@

(
- (gl->gl_state == LM_ST_EXCLUSIVE)
+ gfs2_glock_is_held_excl(gl)
|
- (gl->gl_state != LM_ST_EXCLUSIVE)
+ !gfs2_glock_is_held_excl(gl)
)
```

Example

with metavariable bindings:

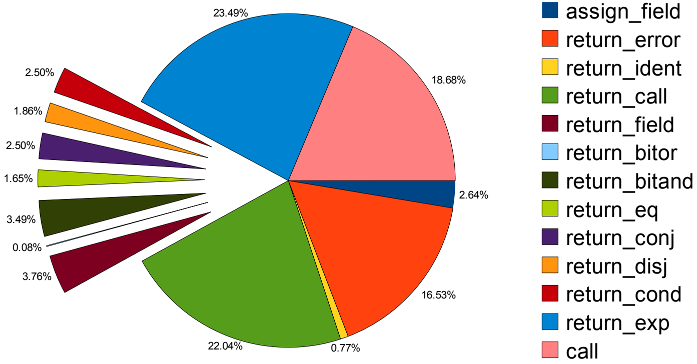
- $E \equiv$ gl->gl_state
- $C \equiv$ LM_ST_EXCLUSIVE
- $f \equiv$ gfs2_glock_is_held_excl
- $ARGS \equiv$ struct gfs2_glock *gl

```
@@
struct gfs2_glock *gl;

@@

(
- (gl->gl_state == LM_ST_EXCLUSIVE)
+ gfs2_glock_is_held_excl(gl)
|
- (gl->gl_state != LM_ST_EXCLUSIVE)
+ !gfs2_glock_is_held_excl(gl)
)
```

Results

- 125 small functions found in 82 files.
- Non-use sites found for 13 functions.
- 26 non-use sites found in 18 files.

Applicability



A larger case study

USB chapter 9 iterface:

- Defined in `include/linux/usb.h`, which is included in over 300 .c files.
- API initially very little used.
- A Linux developer submited a patch that introduced a few uses of this API, which caught our attention.

Function types:

- Call another function.
- Initialize a structure field.
- Return the result of an equality test.
- Return the result of an arbitrary expression.

Results

For four USB chapter 9 functions (one in each category):

	current calls	needed calls	calls updated	% calls updated
usb_set_intfdata	303	4	4	100%
usb_mark_last_busy	15	6	6	100%
usb_endpoint_xfer_isoc	8	11	3	27%
usb_endpoint_is_isoc_in	14	1	0	0%

Issues

Rule ordering

- We follow the definition order in the header file.

The use of constants rather than macros

- We create a matching rule that finds the definition associated with a matched macro use

Rule granularity

- The generative rules express variants at the level of the matched metavariable, not their subterms

Lack of type information

Conclusions

- Linux provides many small functions that raise the level of abstraction and improve type safety
- But these functions are not used systematically
- We propose an approach to automatically address this problem
- In manually checking our usb results, we found one bug in Linux.

Future work

- Consider Walid's more elegant suggestion
- Assess the approach more thoroughly on Linux
- Consider other software: OpenSSL? VLC? Wine?

http://www.x-info.emn.fr/coccinelle