Advanced SmPL: Finding Missing `IS_ERR` tests

Julia Lawall

January 26, 2011

The error handling problem

The C language does not provide any error handling abstractions.

For pointer-typed functions, Linux commonly uses:

- `NULL`
- `ERR_PTR(...)/IS_ERR(...)`

Example:

```
static struct fsnotify_event *get_one_event(struct fsnotify_group *group, size_t count) {
        if (fsnotify_notify_queue_is_empty(group))
                return NULL;
        if (FAN_EVENT_METADATA_LEN > count)
                return ERR_PTR(-EINVAL);
        return fsnotify_remove_notify_event(group);
}
```

Problems:

- The result of a function call may not be tested for an error.
- The result of a function call may be tested for the wrong kind of error.

Our goal: Find missing tests for `ERR_PTR`.

How often is ERR_PTR used anyway?

Strategy:

- Initialize a counter.
- Match calls to ERR_PTR.
- Increment a counter for each reference.
- Print the final counter value.

Initialize a counter

Arbitrary computations can be done using a scripting language:

- Python
- OCaml

Initialize script

- Invoked once, when spatch starts.
- State maintained across the treatment of all files.

Example:

```
@initialize:python@
count = 0
```

Match and count calls to ERR_PTR

Observations:

- SmPL allows matching code, but not performing computation.
- Python/OCaml allows performing computation, but not matching code.

Solution:

- Match using SmPL code.
- Communicate information about the position of each match to Python code.

Example:

```
@r@
position p;
@@
ERR_PTR@p(...)
```

```
@script:python@
p << r.p;
@@
count = count + 1
```

How does it work?

Spatch matches the first rule against each top-level code element, then the second rule against each top-level code element, etc.

Each match creates an environment mapping metavariables to code fragments or positions.

- Some metavariables are only used in the current rule.
- Others are inherited by later rules.
- Each SmPL rule is invoked once for each pair of
  - A toplevel code element, and
  - An environment of inherited metavariables
- Each script rule is invoked once for each each environment that binds all of the inherited metavariables.

Example

```
@r@
position p;
@@
ERR_PTR@p(...)

@script:python@
p << r.p;
@@
count = count + 1
```

```
static struct jump_label_entry *add_jump_label_module_entry(...)
{ ...
  e = kmalloc(sizeof(struct jump_label_module_entry), GFP_KERNEL);
  if (!e)
    return ERR_PTR(-ENOMEM);
  ... }

static struct jump_label_entry *add_jump_label_entry(...)
{ ...
  e = get_jump_label_entry(key);
  if (e)
    return ERR_PTR(-EEXIST);

  e = kmalloc(sizeof(struct jump_label_entry), GFP_KERNEL);
  if (!e)
    return ERR_PTR(-ENOMEM);
  ... }
```

- Matching r against `add_jump_label_module_entry` gives one environment: $[p \mapsto \text{line } 5]$
- Matching r against `add_jump_label_entry` gives two environments: $[p \mapsto \text{line } 12]$, $[p \mapsto \text{line } 16]$
- Script rule invoked 3 times.

Print the final counter value

Finalize script

- Invoked once, when spatch terminates.
- Can access the state obtained from the treatment of all files.

Example:

```
@finalize:python@
print count
```

Summary

Complete semantic patch:

```
@initialize:python@
count = 0

@r@
position p;
@@
ERR_PTR@p(...)

@script:python@
p << r.p;
@@
count = count + 1

@finalize:python@
print count
```

Result for Linux-next, 01.21.2011: 3166

Finding missing IS_ERR tests

If a function returns ERR_PTR(...) its result must be tested using IS_ERR.

ERR_PTR(...) can be returned directly:

```
static struct fsnotify_event *get_one_event(struct fsnotify_group *group, size_t count) {
        if (fsnotify_notify_queue_is_empty(group))
                return NULL;
        if (FAN_EVENT_METADATA_LEN > count)
                return ERR_PTR(-EINVAL);
        return fsnotify_remove_notify_event(group);
}
```

Or returned via a variable:

```
struct ctl_table_header *sysctl_head_grab(struct ctl_table_header *head) {
        if (!head)
                BUG();
        spin_lock(&sysctl_lock);
        if (!use_table(head))
                head = ERR_PTR(-ENOENT);
        spin_unlock(&sysctl_lock);
        return head;
}
```

Collecting functions that return ERR_PTR(...)

```
@r exists@
identifier f,x;
expression E;
@@

f(...) {
  ... when any
(
  return ERR_PTR(...);
|
  x = ERR_PTR(...)
  ... when != x = E
  return x;
)
}
```

Finding missing IS_ERR tests

```
@e exists@
identifier r.f,fld;
expression x;
position p1,p2;

@@

 x@p1 = f(...)

 ...



 x@p2->fld
```

```
@script:python@
f << r.f;
p1 << e.p1;
p2 << e.p2;
@@
cocci.print_main (f,p1)
cocci.print_secs ("ref",p2)
```

Finding missing IS_ERR tests

```
@e exists@
identifier r.f,fld;
expression x;
position p1,p2;
statement S1, S2;
@@
(
 IS_ERR(x = f(...))
|
 x@p1 = f(...)
)
 ... when != IS_ERR(x)
(
 if (IS_ERR(x) ||...) S1 else S2
|
 x@p2->fld
)
```

```
@script:python@
f << r.f;
p1 << e.p1;
p2 << e.p2;
@@
cocci.print_main (f,p1)
cocci.print_secs ("ref",p2)
```

Iteration

Problem: Limited to functions and function calls in the same file.

Solution idea:

- Invoke collection rules on the entire code base.
- Then, reinvoke spatch on the bug finding rules for each collected function.

Issues:

- Two phases.
- In each phase, use only a subset of the semantic patch rules.
- Need a way to give arguments to a semantic patch.

Invoking a subset of the rules of a semantic patch

```
virtual after_start

@r depends on !after_start exists@
identifier f;
@@
f(...) { ... return ERR_PTR(...); }

@e depends on after_start exists@
...
@@
...
```

Command line options to invoke e:

```
spatch -sp_file rule.cocci -D after_start
```

Giving arguments to a semantic patch

```
@e depends on after_start exists@
identifier virtual.f, fld;
expression x;
position p1,p2;
statement S1, S2;
@@
(
 IS_ERR(x = f(...))
|
 x@p1 = f(...)
)
 ... when != IS_ERR(x)
(
 if (IS_ERR(x) ||...) S1 else S2
|
 x@p2->fld
)
```

Command line options to define `f`:

`spatch -sp_file rule.cocci -D f=alloc`

Constructing the iteration (OCaml only)

Goal: Use the identifiers collected by the following rule as arguments to the semantic patch:

```
@r depends on !after_start exists@
identifier f;
@@
f(...)  ... return ERR_PTR(...);
```

Relevant information:

- Files to consider (the current set or a smaller one?)
- The validity of virtual rules.
- The bindings of virtual identifiers

OCaml code

```
@script:ocaml@
f << r.f;
@@

let it = new iteration() in
(* it#set_files file_list *)
it#add_virtual_rule After_start;
it#add_virtual_identifier F f;
it#register()
```

Summary

```
virtual after_start

@r depends on !after_start exists@
identifier f;
@@
f(...)  ... return ERR_PTR(...);

@script:ocaml@
f << r.f;
@@
let it = new iteration() in
(* it#set_files file_list *)
it#add_virtual_rule After_start;
it#add_virtual_identifier F f;
it#register()
```

```
@e depends on after_start exists@
identifier virtual.f, fld;
expression x; position p1,p2;
statement S1, S2;
@@
(
 IS_ERR(x = f(...))
|
 x@p1 = f(...)
)
 ... when != IS_ERR(x)
(
 if (IS_ERR(x) ||...) S1 else S2
|
 x@p2->fld
)

@script:python@
p1 << e.p1; p2 << e.p2;
f << virtual.f;
@@
cocci.print_main (f,p1)
cocci.print_main ("ref",p2)
```

Results (Linux-next from 21.01.2011)

7 reports:

- 5 real bugs
- 2 false positives

Issues:

- Not interprocedural.
  - Can also iterate the function finding process.
- Not sensitive to function visibility (static).
  - For bugs, find them directly.
  - For function collection, limit reinvocation to the same file.

Conclusion

Main issues:

- Initialize and finalize: scripts for initializing and accessing global state.
- Environments for managing inherited metavariables.
- Virtual rules.
- Virtual identifiers.
- Iteration.