Introduction to Coccinelle

Julia Lawall
University of Copenhagen

January 26, 2011
Overview

- The structure of a semantic patch.
- Isomorphisms.
- Depends on.
- Dots.
- Nests.
- Positions.
- Python.
The structure of a semantic patch

Goals:
- Specify patterns of code to be found and transformed.
- Specify which terms should be abstracted over.
- C-like, patch-like notation.
The problem: Combining a boolean (0/1) with a constant using 
& is usually meaningless:

```c
if(!erq->flags & IW_ENCODE_MODE)
{
    return -EINVAL;
}
```

The solution: Add parentheses.

Our goal: Do this automatically for any expression E and 
constant C.
A semantic patch for the !& problem

```plaintext
expression E;
constant C;
```

- !E & C
+ !(E & C)

Two parts per rule:
- Metavariable declaration
- Transformation specification

A semantic patch can contain multiple rules
Issues

Metavariable types

- expression, statement, type, constant, local id
- A type from the source program
- iterator, declarer, iterator name, declarer name, typedef

Transformation specification

- in the leftmost column for something to remove
- + in the leftmost column for something to add
- * in the leftmost column for something of interest
  - Cannot be used with + and −.
- Spaces, newlines irrelevant.
Exercise 1

Write rules to introduce calls to the following functions:

```c
static inline void *
ide_get_hwifdata (ide_hwif_t * hwif)
{
    return hwif->hwif_data;
}

static inline void
ide_set_hwifdata (ide_hwif_t * hwif, void *data)
{
    hwif->hwif_data = data;
}
```

Hints:

- To only consider `ide_hwif_t`-typed expressions, declare a “metavariable” `typedef ide_hwif_t;`.
- Consider both structures and pointers to structures.
- Consider the ordering of the rules.
Practical issues

To check that your semantic patch is valid:

```
spatch -parse_cocci mysp.cocci
```

To run your semantic patch:

```
spatch -sp_file mysp.cocci -dir linux-x.y.z
```

To understand why your semantic patch didn’t work:

```
spatch -sp_file mysp.cocci -dir linux-x.y.z -debug
```

If you don’t need to include header files:

```
spatch -sp_file mysp.cocci -dir linux-x.y.z
    -no_includes -include_headers
```
Solution 1

typedef ide_hwif_t;
ide_hwif_t *dev;
expression data;

- dev->hwif_data = data
+ ide_set_hwifdata(dev, data)

- dev->hwif_data
+ ide_get_hwifdata(dev)
Solution 2 (more concise)

```c
ide_hwif_t *dev;
expression data;

(  
  - dev->hwif_data = data
  + ide_set_hwifdata(dev, data)
  |  
  - dev->hwif_data
  + ide_get_hwifdata(dev)
)
Solution 3 (more complete)

@@ ide_hwif_t *dev; expression data; @@
(
- dev->hwif_data = data
+ ide_set_hwifdata(dev,data)
|
- dev->hwif_data
+ ide_get_hwifdata(dev)
)

@@ ide_hwif_t dev; expression data; @@
(
- dev.hwif_data = data
+ ide_set_hwifdata(&dev,data)
|
- dev.hwif_data
+ ide_get_hwifdata(&dev)
)
Isomorphisms

Goals:

- Transparently treat similar code patterns in a similar way.
The following code is fairly hard to understand:

```c
return (time_ns * 1000 + tick_ps - 1) / tick_ps;
```

kernel.h provides the following macro:

```c
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
```

This is used, but not everywhere it could be.

We can write a semantic patch to introduce new uses.
DIV_ROUND_UP semantic patch

One option:

@@ expression n,d; @@

- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP(n,d)

Another option:

@@ expression n,d; @@

- (n + d - 1) / d
+ DIV_ROUND_UP(n,d)

Problem: How many parentheses to put, to capture all occurrences?
Isomorphisms

An isomorphism relates code patterns that are considered to be similar:

Expression
@ drop_cast @ expression E; pure type T; @@

\((T)E \Rightarrow E\)

Expression
@ paren @ expression E; @@

\((E) \Rightarrow E\)

Expression
@ is_null @ expression X; @@

\(X == \text{NULL} \Leftrightarrow \text{NULL} == X \Rightarrow \neg X\)
Isomorphisms, contd.

Isomorphisms are handled by rewriting.

\[
(((n) + (d) - 1) / (d))
\]

becomes:

\[
((n) + (d) - 1) / (d) \\
((n) + d - 1) / (d) \\
((n) + d - 1) / d \\
(n + (d) - 1) / (d) \\
(n + (d) - 1) / d \\
(n + d - 1) / (d) \\
(n + d - 1) / d \\
\text{etc.}
\]
Practical issues

Default isomorphisms are defined in standard.iso

To use a different set of default isomorphisms:

    spatch -sp_file mysp.cocci -dir linux-x.y.z -iso_file empty.iso

To drop specific isomorphisms:

@disable paren@ expression n,d; @@
- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP (n, d)

To add rule-specific isomorphisms:

@using "myparen.iso" disable paren@
expression n,d;
@@
- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP (n, d)
Goals:

- Define multiple matching and transformation rules.
- Express that the applicability of one rule depends on the success or failure of another.
DIV_ROUND_UP is defined in kernel.h

- The transformation might not be correct if kernel.h is not included.
- **Problem:** #include <linux/kernel.h> is far from the call to DIV_ROUND_UP

```
#include <linux/kernel.h>

expression n,d;

- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP(n,d)
```
Goals:

- Specify patterns consisting of fragments of code separated by arbitrary execution paths.
- Specify constraints on the contents of those execution paths.
Nested spin_lock_irqsave

spin_lock_irqsave(lock,flags):
  ▶ Takes a lock.
  ▶ Saves current interrupt status in flags.
  ▶ Disables interrupts.

Invalid nested usage:

spin_lock_irqsave(&port->lock, flags);
if (sx_crtsts(port->port.tty))
  if (set & TIOCM_RTS) port->MSVR |= MSVR_DTR;
  else if (set & TIOCM_DTR) port->MSVR |= MSVR_DTR;
spin_lock_irqsave(&bp->lock, flags);
sx_out(bp, CD186x_CAR, port_No(port));
sx_out(bp, CD186x_MSVR, port->MSVR);
spin_unlock_irqrestore(&bp->lock, flags);
spin_unlock_irqrestore(&port->lock, flags);
Detecting nested spin_lock_irqsave

Observations:

- **Calls to** `spin_lock_irqsave` **share their second argument.**
  - **Solution:** repeated metavariables.

- **Calls to** `spin_lock_irqsave` **may be separated by arbitrary code.**
  - **Solution:** ...

- **There should be no calls to** `spin_lock_irqrestore` **between the calls to** `spin_lock_irqsave`.  
  - **Solution:** when
A semantic match for detecting nested spin_lock_irqsave

@@
expression lock1, lock2;
expression flags;
@@

*spin_lock_irqsave(lock1, flags)
... when != flags
*spin_lock_irqsave(lock2, flags)
Nests

Goals:

- Describe terms that can occur any number of times within an execution path.
- 0 or more times, or 1 or more times.
Detecting memory leaks

A simple case of a memory leak:

- An allocation.
- Storage in a local variable.
- No use.
- Return of an error code (negative constant).

```c
local idexpression x;
statement S;
constant C;
```

```c
*x = \(kmalloc\|kzalloc\|kzalloc\)(...);
...
if (x == NULL) S
... when != x
*return -C;
```
3 bugs detected, for example:

tmp_store = kmalloc(sizeof(*tmp_store), GFP_KERNEL);
if (!tmp_store) {
    ti->error = "Exception store allocation failed";
    return -ENOMEM;
}

persistent = toupper(*argv[1]);
if (persistent != 'P' && persistent != 'N') {
    ti->error = "Persistent flag is not P or N";
    return -EINVAL;
}
Towards a more general semantic match

    if (chip == NULL) {
        chip = kzalloc(sizeof(struct chip_data), GFP_KERNEL);
        if (!chip)
            return -ENOMEM;

        chip->enable_dma = 0;
        chip_info = spi->controller_data;
    }

    if (chip_info) {
        if (chip_info->ctl_reg&(SPE|MSTR|CPOL|CPHA|LSBF)) {
            dev_err(&spi->dev, "do not set bits in ctl_reg "
                    "that the SPI framework manages");
            return -EINVAL;
        }
    } ...

Accessing a field of chip doesn’t eliminate the need to free it.
A more general semantic match

@@
local idexpression x;
statement S;
constant C;
@@

*x = \(\text{kmalloc}\|\text{kzalloc}\|\text{kzalloc}\)(...);
...
if (x == NULL) S
<... when != x
x->fld = E
...>
*return -C;

Finds 2 more bugs, but 1 false positive as well.
Other uses of nests

\(<\ldots\ P \ldots\ >:\)

- Change all occurrences within a region of code.
- **Example:** a parameter is replaced by a call to an access function.

\(<+\ldots\ P \ldots\ +\>:\)

- Change or match at least one occurrence in a region of code.
- Change or match at least one occurrence within an expression.
- **Example:** `kfree(<+\ldots\ x \ldots\ +\>)`;
Positions and Python

Goals:
- Positions: remember exactly what fragment of code was matched.
- Python: do arbitrary computation, especially printing.
if (mode & V4L2_TUNER_MODE_MONO)
    s1 |= TDA8425_S1_STEREO_MONO;

- V4L2_TUNER_MODE_MONO is 0.
- The test is always false.
Detecting & with 0

One strategy:

- Search for constants that are defined to 0.
- Check that there is not another nonzero definition.
- Find a corresponding use of &.

Another strategy:

- Find a use of &.
- Check that the constant is 0.
- Check that there is not another nonzero definition.
- Report on the bug site.

The better strategy depends on how many matches there are at each step.

We take the second strategy, for illustration.
Find a use of &

@r expression@
ideminter C;
expression E;
position p;
@@

E & C@p

- The rule has a name: r.
- p is a position metavariable, so we can find the same & expression later.
Check that \( C \) is 0

```c
_identifier r.C;
#define C 0
```

```c
_identifier r.C;
_expression E != 0;
#define C E
```

- Both rules inherit \( C \).
- Each rule is applied once for each value of \( C \).
- The second rule puts a constraint on \( E \).
  - Constraints on constants, expressions, identifiers, positions
  - Regular expressions allowed for constants and identifiers.
Printing the result

@script:python depends on s && !t@
p << r.p;
C << r.C;
@@
cocci.print_main("and with 0", p)

- Python rules only inherit metavariables, using << notation.
- Depends on clause is evaluated for each inherited set of metavariable bindings.
- print_main is part of a library for printing output in Emacs org mode.
The complete semantic patch

@r expression@
identifier C;
expression E;
position p;
@@
E & C@p

@s@ identifier r.C; @@
#define C 0

@t@ identifier r.C; expression E != 0; @@
#define C E

@script:python depends on s && !t@
p << r.p;
C << r.C;
@@
cocci.print_main("and with 0", p)