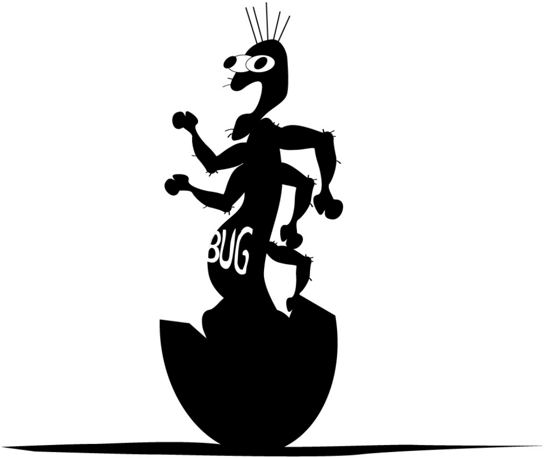
# WYSIWIB: A Declarative Approach to Finding API Protocols and Bugs in Linux Code

Julia Lawall (University of Copenhagen)

Julien Brunel, Nicolas Palix, René Rydhof Hansen,
Henrik Stuart, Gilles Muller

# Bugs: They're everywhere!



# Our focus

## Bugs in Linux

- Linux is critical software.
  - Used in embedded systems, desktops, servers, etc.
- Linux is very large.
  - Over 12 000 .c files
  - Over 7 million lines of code
  - Increase of almost 50% since 2006.
- Linux has both more and less experienced developers.
  - Maintainers, contributers, developers of proprietary drivers

# A bug finding story

## `netif_rx` might free its argument

```
commit d30f53aeb31d453a5230f526bea592af07944564
Author: Wang Chen <wangchen@cn.fujitsu.com>
Date:   Tue Dec 4 10:01:37 2007 +0800

    SMC911X: Fix using of dereferenced skb after netif_rx

diff --git a/drivers/net/smc911x.c b/drivers/net/smc911x.c
--- a/drivers/net/smc911x.c
+++ b/drivers/net/smc911x.c
@@ -1304,3 +1304,3 @@ smc911x_rx_dma_irq(int dma, void *data)
-       netif_rx(skb);
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += skb->len;
+       netif_rx(skb);
```

A potential kernel OOPS.

# Are there other netif_rx bugs?

```
diff --git a/drivers/net/smc911x.c b/drivers/net/smc911x.c
--- a/drivers/net/smc911x.c
+++ b/drivers/net/smc911x.c
@@ -1304,3 +1304,3 @@ smc911x_rx_dma_irq(int dma, void *data)
-       netif_rx(skb);
        dev->stats.rx_packets++;
        dev->stats.rx_bytes += skb->len;
+       netif_rx(skb);
```

These bugs are hard to find with grep:

- Bug pattern crosses multiple lines
- Old and new code looks identical (on different lines)
- 314 calls to `netif_rx`

# Are there other netif_rx bugs?

```
diff --git a/drivers/net/smc911x.c b/drivers/net/smc911x.c
--- a/drivers/net/smc911x.c
+++ b/drivers/net/smc911x.c
@@ -1304,3 +1304,3 @@ smc911x_rx_dma_irq(int dma, void *data)
-        netif_rx(skb);
         dev->stats.rx_packets++;
         dev->stats.rx_bytes += skb->len;
+        netif_rx(skb);
```

These bugs are hard to find with grep:

- Bug pattern crosses multiple lines
- Old and new code looks identical (on different lines)
- 314 calls to `netif_rx`

Need a tool that takes multiple lines of code into account

# Coccinelle

A program matching and transformation engine

- Matching of complex patterns (semantic matches)
- Elements connected by control-flow paths
- Elements abstracted via metavariables

```
@@
expression X;
@@
* f(X)
  ...
* g(X)
```

# Use Coccinelle to find other potentially dangerous netif_rx calls

```
@@
expression skb;
identifier fld;
@@
* netif_rx(skb)
  ...
* skb->fld
```

- Two more bugs found.
- The function `netif_rx_ni` has the same behavior.

# Are there netif_rx_ni bugs?

In the semantic match, replace `netif_rx` by `netif_rx_ni`.

```
@@
expression skb;
identifier fld;
@@
* netif_rx_ni(skb)
  ...
* skb->fld
```

- Three more bugs found.

# Are there netif_rx_ni bugs?

In the semantic match, replace `netif_rx` by `netif_rx_ni`.

```
@@
expression skb;
identifier fld;
@@
* netif_rx_ni(skb)
  ...
* skb->fld
```

- Three more bugs found.

How can we generalize this approach?

# Generalizing the approach

- Are there other functions like `netif_rx` and `netif_rx_ni`?
- How can we find them?
- How can we find bugs in their usage?

A general problem:
API usage constraints are often not well understood.

# Existing tools

SLAM/SDV, Splint, Flawfinder, etc.

- Find potential bugs in the use of a fixed set of APIs
- Not easily portable to other software

Metal/Coverity, PR-Miner etc.

- Search for API usage protocols
- Report potential violations of these protocols
- To avoid false positives, rely on data mining
- May overlook infrequent patterns

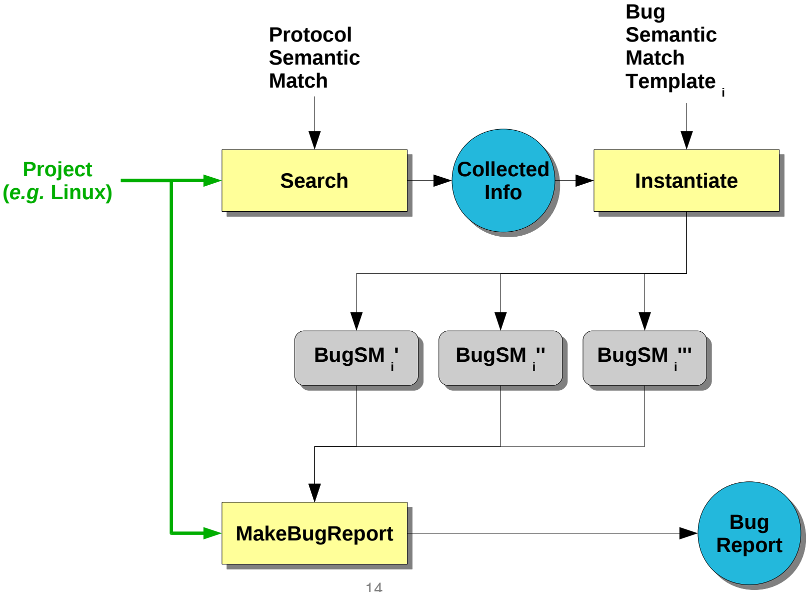
Both miss software-specific, infrequently used protocols.

The software developer's knowledge must be taken into account.

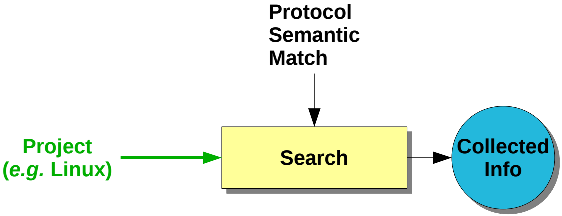
# Our hypotheses

- API functions share common protocol types and associated bug types.
- These protocol and bug types can be expressed as code patterns.
- Developers are aware of these protocol and bug types.
- Developers are not aware of some relevent API functions.

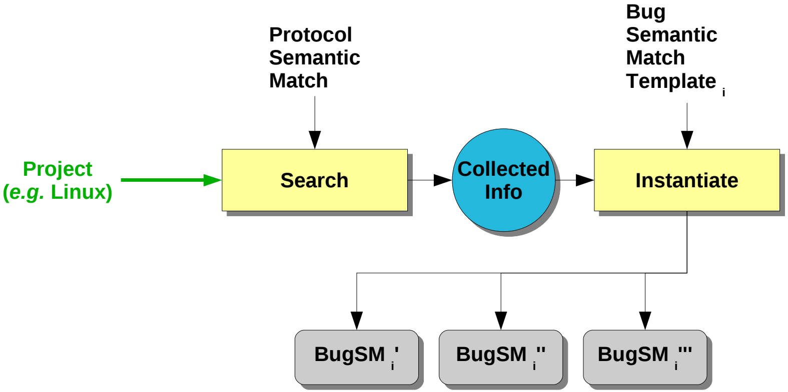
# Our approach



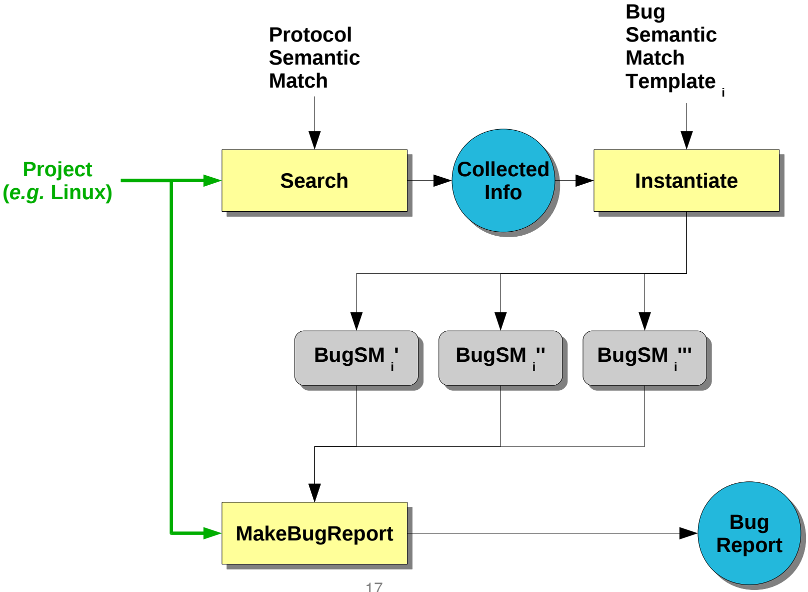
# Our approach



# Our approach



# Our approach



# Case studies

Use after free:

- Protocol finding: identify functions that may free an argument.
- Bug finding: reference to the argument after calling the function.

Memory leaks:

- Protocol finding: identify allocators and deallocators.
- Bug finding: allocator calls with no deallocator calls in error handling code.

Inconsistent error checks:

- Protocol finding: identify functions that return NULL, and those that retun ERR_PTR.
- Bug finding: Inappropriate tests, insufficient tests.

# Bug finding results

## Use after free

| | reported sites | bugs | false positives |
|---|---|---|---|
| guaranteed free | 10 | 5 | 5 |
| possible free | 22 | 9 | 13 |

## Memory leaks

| | reported sites | bugs | false positives |
|---|---|---|---|
| alloc/dealloc | 261 | 141 | 120 |

# Inconsistent error checks

## Protocol finding results:

| | classified | false positives |
|---|---|---|
| NULL only | 1640 | 9 |
| ERR_PTR only | 478 | 1 |
| NULL or ERR_PTR | 112 | 9 |
| Pointer only | 623 | 5 |
| unknown | 7123 | N/A |

## Bug finding: inappropriate tests

| | reported sites | bugs | false positives |
|---|---|---|---|
| NULL only | 2 | 2 | 0 |
| ERR_PTR only | 26 | 19 | 7 |
| Pointer only | 44 | 23 | 21 |

## Bug finding: insufficient tests

| | reported sites | bugs | false positives |
|---|---|---|---|
| NULL only | 201 | 139 | 62 |
| ERR_PTR only | 21 | 17 | 4 |
| NULL or ERR_PTR | 11 | 5 | 6 |

# Conclusions and future work

A framework for finding protocols and bugs in Linux code

- Exploits the expertise of Linux developers about their software
- Lightweight, C-code based specification language

Finds bugs in Linux code

- Many have been fixed by ourselves or others.

Future work: Combining our framework with data mining

- Reduces the rate of false positives.
- Allows even simpler specifications.

# Kill bugs before they hatch!!!


COCCINELLE

# Use after free

## Protocol finding:

```
@kfree exists@
identifier f,x; position p; type T;
@@
void f(...,T x@p,...) { ... kfree(x); ... }

@other_path exists@
identifier kfree.f, kfree.x; position kfree.p; type kfree.T;
@@
void f(...,T x@p,...) { ... when != kfree(x); }


@ script:python depends on other_path @
f << kfree.f; t << kfree.T;
@@
print "possible_kfree: FN:%s TY:%s" % (f,t)

@ script:python depends on !other_path @
f << kfree.f; t << kfree.T;
@@
print "guaranteed_kfree: FN:%s TY:%s" % (f,t)
```

# Results

```
...
guaranteed_kfree: FN:put_tty_driver TY:struct tty_driver *
possible_kfree: FN:n_hdlc_release TY:struct n_hdlc *
guaranteed_kfree: FN:i2c_tiny_usb_free TY:struct i2c_tiny_usb
guaranteed_kfree: FN:framebuffer_release TY:struct fb_info *
guaranteed_kfree: FN:icom_free_adapter TY:struct icom_adapter
possible_kfree: FN:cxgb_free_mem TY:void *
possible_kfree: FN:c101_destroy_card TY:card_t *
...
```

# Bug finding semantic match template (simplified)

```
@x@
TY E;
identifier f;
expression E1;
position p1,p2;
@@

FN@p1(...,E,...);
...
(
  E = E1
|
  E@p2
)

@ script:python @
p1 << x.p1;
p2 << x.p2;
@@

cocci.print_main("call",p1)
cocci.print_secs("ref",p2)
```

# Instantiated bug finding semantic match template (simplified)

```
@x@
struct tty_driver * E;
identifier f;
expression E1;
position p1,p2;
@@
put_tty_driver@p1(...,E,...);
...
(
  E = E1
|
  E@p2
)

@ script:python @
p1 << x.p1;
p2 << x.p2;
@@
cocci.print_main("call",p1)
cocci.print_secs("ref",p2)
```

```
@x@
struct n_hdlc * E;
identifier f;
expression E1;
position p1,p2;
@@
n_hdlc_release@p1(...,E,...);
...
(
  E = E1
|
  E@p2
)

@ script:python @
p1 << x.p1;
p2 << x.p2;
@@
cocci.print_main("call",p1)
cocci.print_secs("ref",p2)
```

# Case study: Inconsistent error checks

Error values in Linux:

- Some functions return NULL
- Some functions return ERR_PTR(...)
- Some functions may return both

Protocol finding:

- Find functions containing `return NULL;`, `return ERR_PTR(...);`, etc.

Bug finding:

- Find function calls followed by incorrect or insufficient tests.

# Protocol finding semantic match

```
@rn exists@
identifier f,fld;
expression E,E1;
@@

f(...)  ...
(
return NULL;
|
E = NULL
... when != E=E1
    when != E->fld
return E;
)


@ script:python @
f << rn.f;
@@

print "null: FN:%s" % f
```

# Results

```
...
null: FN:aac_fib_alloc
null: FN:aac_init_adapter
null: FN:aarp_alloc
null: FN:abituguru3_update_device
null: FN:abituguru_update_device
null: FN:ac97_alloc_codec
null: FN:acpi_pci_irq_find_prt_entry
...
```

# Bug finding semantic match template

```
@unprotected exists@
identifier fld, fld1;
position p,p1;
expression x,E,E1;
@@
 x@p1 = FN(...);
  ... when != x = E
      when != &x
      when != x->fld1
(
  x == NULL
|
  x@p->fld
)

@ script:python @
p1 << unprotected.p1;    // position of call
p << unprotected.p;      // position of ref
fld << unprotected.fld; // identifier
@@
cocci.print_main("call",p1)
cocci.print_secs("field ref",p)
```

# Instantiated bug finding semantic match template

```
@unprotected exists@
identifier fld, fld1;
position p,p1;
expression x,E,E1;
@@
 x@p1 = aac_fib_alloc(...);
  ... when != x = E
      when != &x
      when != x->fld1
(
  x == NULL
|
  x@p->fld
)

@ script:python @
p1 << unprotected.p1;
p << unprotected.p;
fld << unprotected.fld;
@@
cocci.print_main("call",p1)
cocci.print_secs("field ref",p)
```

```
@unprotected exists@
identifier fld, fld1;
position p,p1;
expression x,E,E1;
@@
 x@p1 = aac_init_adapter(...);
  ... when != x = E
      when != &x
      when != x->fld1
(
  x == NULL
|
  x@p->fld
)

@ script:python @
p1 << unprotected.p1;
p << unprotected.p;
fld << unprotected.fld;
@@
cocci.print_main("call",p1)
cocci.print_secs("field ref",p)
```

# A bug that was found

```
  aaci = aaci_init_card(dev);
  if (IS_ERR(aaci)) {
    ret = PTR_ERR(aaci);
    goto out;
  }
  [. . .]
out:
  if (aaci)
    snd_card_free(aaci->card);
```