Finding the right level of abstraction: Program analysis and the Linux kernel

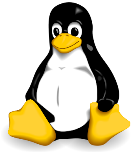
Julia Lawall (Inria/LIP6-Whisper)

March 8, 2016

Our focus

The Linux kernel:

- Critical software.
  - Used in embedded systems, desktops, servers, etc.
- Very large.
  - Over 22 000 .c files.
  - Over 13.6 million lines of C code.
  - Increase of 44% since July 2011 (Linux 3.0).
  - Hundreds of contributors.
- More and less experienced developers.
  - Maintainers, contributors, developers of proprietary drivers



Critical questions

Bugs seem inevitable:

- How can we find and fix bugs in the code?

Code must continually evolve:

- How can we improve security, performance, maintainability, etc?

Critical questions

Bugs seem inevitable:

- How can we find and fix bugs in the code?

Code must continually evolve:

- How can we improve security, performance, maintainability, etc?

Requires program analysis, program transformation techniques.

- At the core of programming languages research.

A little history...

Evolution of program analysis research

Starting point:

- Idealized imperative imperative or functional languages
- Emphasis on improving precision.
- Less attention to scalability.

Evolution of program analysis research

Starting point:

- Idealized imperative imperative or functional languages
- Emphasis on improving precision.
- Less attention to scalability.
- Messy language features not addressed.
- Relevance of precise analyses to real code not clear.

Program analysis and operating systems

Late 1990s

- OS community finding that one size does not fit all
  - Need for programmability, and thus safety.
- Language community saw a need for validation on real applications
  - Operating systems are complex (Linux 1.0: $>$120 KLOC)
  - OS correctness really matters

Strategy 1: Reimplementation in safer languages

SPIN: Extensible operating system developed in Modula 3
[Bershad et al.: SOSP 1995]

FoxNet: Network protocol stack in SML
[Biagioni, Harper, Lee: LFP 1994, HOSC 2001]

House: Operating system implemented in Haskell
[Hallgren, Jones, Leslie, and Tolmach: ICFP 2005]

Assessment

- Interesting issues explored.
- Little practical usage.
- Legacy incompatible.

Strategy 2: A safer C

CCured: make existing C programs type safe

- Validatable pointer operations run unchanged.
- Runtime metadata and checks for dangerous operations.
- Some valid code is rejected.
- [Necula, McPeak, Weimer: POPL 2002]

Cyclone: a safe dialect of C

- Like C, but annotations on dangerous code.
- [Jim, et al.:USENIX 2002]

Assessment

- Overheads remain.
- Library incompatibilities.

Strategy 3: Unsound, incomplete analysis

Observations:

- The Linux kernel is large, but many modules are self-contained.
  - E.g., flow-sensitive interprocedural pointer analysis perhaps unnecessary.
- The Linux kernel has to be understood by humans.
  - Open source development model.

Maybe we can do less and get more:

- Less precision.
- Find more real bugs in more large code bases

Metal [Engler et al., OSDI 2002]

Lightweight, programmable bug detection:

- State machine to describe bug patterns
- Patterns to match code fragments
- Applied on control-flow graph

Metal [Engler et al., OSDI 2002]

Lightweight, programmable bug detection:

- State machine to describe bug patterns
- Patterns to match code fragments
- Applied on control-flow graph

Example:

```
state decl any_pointer v;

start: { kfree(v) } ==> v.freed;

v.freed:
  { *v } ==> v.stop, { err("use after free"); }
| { kfree(v) } ==> v.stop, { err("double free"); }
;
```

Assessment

+ Lightweight scanning can process a huge code base
+ Hundreds of bugs found, in Linux and BSD
+ More precise version: SLAM (SDV) from Microsoft
+ Shifted attention to what kind of bugs to look for
  - Motivated protocol mining using machine learning techniques
- State machine notation unstructured
- Finds bugs but doesn't fix them

Our approach: Coccinelle

Coccinelle

Approach:

- Static analysis to find patterns in C code.
- Automatic transformation to fix bugs.
- User scriptable, based on patch notation (semantic patches).
- http://coccinelle.lip6.fr/

Coccinelle

Approach:

- Static analysis to find patterns in C code.
- Automatic transformation to fix bugs.
- User scriptable, based on patch notation (semantic patches).
- `http://coccinelle.lip6.fr/`

Goal: Be accessible to C code developers.

Coccinelle

Approach:

- Static analysis to find patterns in C code.
- Automatic transformation to fix bugs.
- User scriptable, based on patch notation (semantic patches).
- `http://coccinelle.lip6.fr/`

Goal: Be accessible to C code developers.

Find once, fix everywhere.

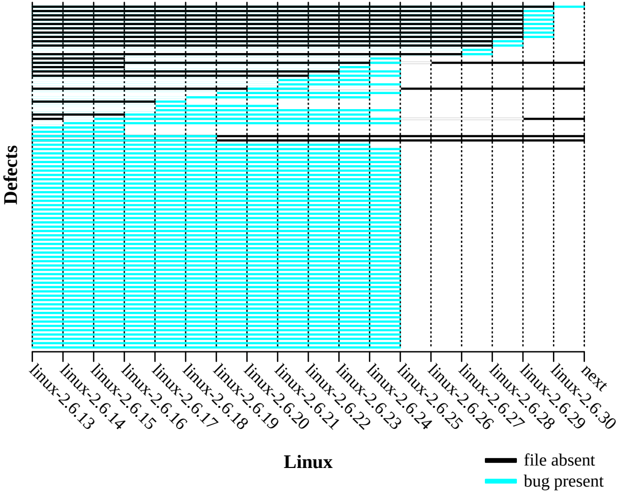
Bug: !x&y

```
Author: Al Viro <viro@ZenIV.linux.org.uk>

    wmi: (!x & y) strikes again

diff --git a/drivers/acpi/wmi.c b/drivers/acpi/wmi.c
@@ -247,7 +247,7 @@
  block = &wblock->gblock;
  handle = wblock->handle;

- if (!block->flags & ACPI_WMI_METHOD)
+ if (!(block->flags & ACPI_WMI_METHOD))
    return AE_BAD_DATA;

  if (block->instance_count < instance)
```

Issue

Isolated problems, but these bug types can occur many times

!x&y case:



Bug: !x&y

```
Author: Al Viro <viro@ZenIV.linux.org.uk>

    wmi: (!x & y) strikes again

diff --git a/drivers/acpi/wmi.c b/drivers/acpi/wmi.c
@@ -247,7 +247,7 @@
  block = &wblock->gblock;
  handle = wblock->handle;

- if (!block->flags & ACPI_WMI_METHOD)
+ if (!(block->flags & ACPI_WMI_METHOD))
    return AE_BAD_DATA;

  if (block->instance_count < instance)
```

How to automate this change?

- For any `!E & C`
  - where `E` is any expression, and
  - where `C` is any constant,
- Add parentheses around `E & C`

Finding and fixing !x&y bugs using Coccinelle

```
@@
expression E;
constant C;
@@

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

- `E` is an arbitrary expression.
- `C` is an arbitrary constant.

Example

Original code:

```
if (!state->card->
    ac97_status & CENTER_LFE_ON)
        val &= ~DSP_BIND_CENTER_LFE;
```

Semantic patch:

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

Generated code:

```
if (!(state->card->ac97_status & CENTER_LFE_ON))
        val &= ~DSP_BIND_CENTER_LFE;
```

Example

Original code:

```
  if (!state->card->
     ac97_status & CENTER_LFE_ON)
          val &= ~DSP_BIND_CENTER_LFE;
```

Semantic patch:

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

Generated code:

```
  if (!(state->card->ac97_status & CENTER_LFE_ON))
          val &= ~DSP_BIND_CENTER_LFE;
```

96 instances in Linux from 2.6.13 (August 2005) to v2.6.28 (December 2008)

Some more Coccinelle features

Dots:

```
a();
...
b(x);
```

Nests:

```
if (<+... x == NULL ...+>) S
```

Positions:

```
foo@p(x,y)
```

Some more Coccinelle features

Dots:

```
a();
... when != x
b(x);
```

Nests:

```
if (<+... x == NULL ...+>) S
```

Positions:

```
foo@p(x,y)
```

A more complex example

Linux commit 364d5716a:
rtnetlink: ifla_vf_policy: fix misuses of NLA_BINARY

ifla_vf_policy[] is wrong in advertising its individual member types as NLA_BINARY since .type = NLA_BINARY in combination with .len declares the len member as *max* attribute length [0, len].

Our starting point

Linux commit 364d5716a:
rtnetlink: ifla_vf_policy: fix misuses of NLA_BINARY

ifla_vf_policy[] is wrong in advertising its individual member types as NLA_BINARY since .type = NLA_BINARY in combination with .len declares the len member as *max* attribute length [0, len].

```
- [IFLA_VF_MAC]      = { .type = NLA_BINARY,
-                        .len = sizeof(struct ifla_vf_mac) },
- [IFLA_VF_VLAN]     = { .type = NLA_BINARY,
-                        .len = sizeof(struct ifla_vf_vlan) },
- ...
+ [IFLA_VF_MAC]      = { .len = sizeof(struct ifla_vf_mac) },
+ [IFLA_VF_VLAN]     = { .len = sizeof(struct ifla_vf_vlan) },
+ ...
```

Understanding the problem

Some uses of IFLA_VF_MAC, IFLA_VF_VLAN:

```
struct ifla_vf_mac *ivm = nla_data(tb[IFLA_VF_MAC]);
struct ifla_vf_vlan *ivv = nla_data(tb[IFLA_VF_VLAN]);
```

Understanding the problem

Some uses of IFLA_VF_MAC, IFLA_VF_VLAN:

```
struct ifla_vf_mac *ivm = nla_data(tb[IFLA_VF_MAC]);
struct ifla_vf_vlan *ivv = nla_data(tb[IFLA_VF_VLAN]);
```

In a little more detail:

```
struct ifla_vf_mac *ivm = nla_data(tb[IFLA_VF_MAC]);

err = -EOPNOTSUPP;
if (ops->ndo_set_vf_mac)
  err = ops->ndo_set_vf_mac(dev, ivm->vf, ivm->mac);
```

Understanding the problem

Some uses of `IFLA_VF_MAC`, `IFLA_VF_VLAN`:

```
struct ifla_vf_mac *ivm = nla_data(tb[IFLA_VF_MAC]);
struct ifla_vf_vlan *ivv = nla_data(tb[IFLA_VF_VLAN]);
```

In a little more detail:

```
struct ifla_vf_mac *ivm = nla_data(tb[IFLA_VF_MAC]);

err = -EOPNOTSUPP;
if (ops->ndo_set_vf_mac)
  err = ops->ndo_set_vf_mac(dev, ivm->vf, ivm->mac);
```

Too small ivm may not have `vf` and `mac` fields.

Exploring a little more

What are some other usage contexts of `nla_data`?

```
@@
constant c : script:ocaml() { is_nla_binary(c) };
expression e;
@@
* nla_data(e[c])
```

Exploring a little more

What are some other usage contexts of `nla_data`?

```
@@
constant c : script:ocaml() { is_nla_binary(c) };
expression e;
@@
* nla_data(e[c])
```

Sample result (drivers/net/macvlan.c):

```
if (nla_len(tb[IFLA_ADDRESS]) != ETH_ALEN)
  return -EINVAL;
if (!is_valid_ether_addr(nla_data(tb[IFLA_ADDRESS])))
  return -EADDRNOTAVAIL;
```

Assessment

Some observations:

- `nla_len` obtains the actual data size.
- Bad sized data should be rejected before accesses.

Assessment

Some observations:

- `nla_len` obtains the actual data size.
- Bad sized data should be rejected before accesses.

Semantic patch idea:

- Find the positions of safe accesses.
- Report on accesses at other positions.

First attempt

```
@checked1@
constant c;
expression e;
position p;
@@
if (<+... nla_len(e[c]) ...+>) { ... return ...; }
...
nla_data@p(e[c])

@@
constant c : script:ocaml() { is_nla_binary(c) };
expression e;
position p != checked1.p;
@@
* nla_data@p(e[c])
```

Planning ahead

```
@checked2@
constant c;
expression l,e,e1;
position p;
@@

l = nla_len(e[c])
... when != l = e1
if (<+... l ...+>) { ... return ...; }
...
nla_data@p(e[c])
```

More possibilities (simplified)

`nla_data` before test, but test before access:

```
@checked3@
constant c;
expression e,l,b;
position p;
@@

l = nla_len(e[c])
b = nla_data@p(e[c])
if (<+... l ...+>)
  { ... return ...; }
```

```
@checked4@
constant c;
expression e,l,b;
position p;
@@

b = nla_data@p(e[c])
l = nla_len(e[c])
if (<+... l ...+>)
  { ... return ...; }
```

Let's try it out!

net/tipc/udp_media.c:

```
 if (opts[TIPC_NLA_UDP_LOCAL] && opts[TIPC_NLA_UDP_REMOTE]) {
   sa_local = nla_data(opts[TIPC_NLA_UDP_LOCAL]);
   sa_remote = nla_data(opts[TIPC_NLA_UDP_REMOTE]);
 } else {
err:
   pr_err("Invalid UDP bearer configuration");
   return -EINVAL;
 }
 if ((sa_local->ss_family & sa_remote->ss_family) == AF_INET) {
   ...
 }
```

Let's try it out!

net/tipc/udp_media.c:

```
 if (opts[TIPC_NLA_UDP_LOCAL] && opts[TIPC_NLA_UDP_REMOTE]) {
   sa_local = nla_data(opts[TIPC_NLA_UDP_LOCAL]);
   sa_remote = nla_data(opts[TIPC_NLA_UDP_REMOTE]);
 } else {
err:
   pr_err("Invalid UDP bearer configuration");
   return -EINVAL;
 }
 if ((sa_local->ss_family & sa_remote->ss_family) == AF_INET) {
   ...
 }
```

Let's try it out!

net/tipc/udp_media.c:

```
 if (opts[TIPC_NLA_UDP_LOCAL] && opts[TIPC_NLA_UDP_REMOTE]) {
   sa_local = nla_data(opts[TIPC_NLA_UDP_LOCAL]);
   sa_remote = nla_data(opts[TIPC_NLA_UDP_REMOTE]);
 } else {
err:
   pr_err("Invalid UDP bearer configuration");
   return -EINVAL;
 }
 if ((sa_local->ss_family & sa_remote->ss_family) == AF_INET) {
   ...
 }
```

Both bugs confirmed.

Some more found code

net/wireless/nl80211.c:

```
if (tb[NL80211_KEY_DATA]) {
  k->p.key = nla_data(tb[NL80211_KEY_DATA]);
  k->p.key_len = nla_len(tb[NL80211_KEY_DATA]);
}
```

Some more found code

net/wireless/nl80211.c:

```
if (tb[NL80211_KEY_DATA]) {
  k->p.key = nla_data(tb[NL80211_KEY_DATA]);
  k->p.key_len = nla_len(tb[NL80211_KEY_DATA]);
}
```

net/netlabel/netlabel_unlabeled.c:

```
ret_val = security_secctx_to_secid(
              nla_data(info->attrs[NLBL_UNLABEL_A_SECCTX]),
              nla_len(info->attrs[NLBL_UNLABEL_A_SECCTX]),
              &secid);
```

Issues

Some uses of `nla_len` and `nla_data` values:

- Stored in structure fields.
- Passed to a function.

Issues

Some uses of `nla_len` and `nla_data` values:

- Stored in structure fields.
- Passed to a function.

Two options:

- Collect structure types and field names, and search for uses.
  - Likewise for function names and arguments.

Issues

Some uses of `nla_len` and `nla_data` values:

- Stored in structure fields.
- Passed to a function.

Two options:

- Collect structure types and field names, and search for uses.
  - Likewise for function names and arguments.
- Assume that if the developer thought of taking the length, he also thought of doing the right thing with it.
  - Less precise, but pragmatic.

Extending the semantic patch: structure fields

```
@semichecked1@
constant c; position p;
expression e,b,bp;
identifier f1,f2; type T;
@@

(
b.f1 = nla_len(e[c]);
...
b.f2 = (T)nla_data@p(e[c]);
|
b.f1 = (T)nla_data@p(e[c]);
...
b.f2 = nla_len(e[c]);
)
```

```
@semichecked2@
constant c; position p;
expression e,b,bp;
identifier f1,f2; type T;
@@

(
b->f1 = nla_len(e[c]);
...
b->f2 = (T)nla_data@p(e[c]);
|
b->f1 = (T)nla_data@p(e[c]);
...
b->f2 = nla_len(e[c]);
)
```

Extending the semantic patch: function arguments

```
@semichecked3@
constant c; position p;
expression e,e1,e2;
identifier f; type T;
@@

e1 = (T)nla_data@p(e[c])
...
e2 = nla_len(e[c])
...
(
f(...,e1,...,e2,...)
|
f(...,e2,...,e1,...)
)
```

```
@semichecked4@
constant c; position p;
expression e,e1,e2;
identifier f; type T;
@@

e1 = nla_len(e[c])
...
e2 = (T)nla_data@p(e[c])
...
(
f(...,e1,...,e2,...)
|
f(...,e2,...,e1,...)
)
```

Also a rule for direct function arguments.

Final bug reporting rule

```
@checked1@
constant c; expression e; position p;
@@
if (<+... nla_len(e[c]) ...+>) { ... return ...; }
...
nla_data@p(e[c])

[...]

@@
constant c : script:ocaml() { is_nla_binary(c) };
expression e;
position p != { checked1.p, checked2.p, checked3.p, checked4.p,
                semichecked1.p, semichecked2.p,
                semichecked3.p, semichecked3.p, semichecked5.p }
@@
* nla_data@p(e[c])
```

Results

- 15 reports
- 10 probable real bugs
  - False positives mostly due to separate validation functions
- Fixes in progress
- Also possible bugs on `NLA_STRING` and `NLA_NUL_STRING` types

Impact in practice

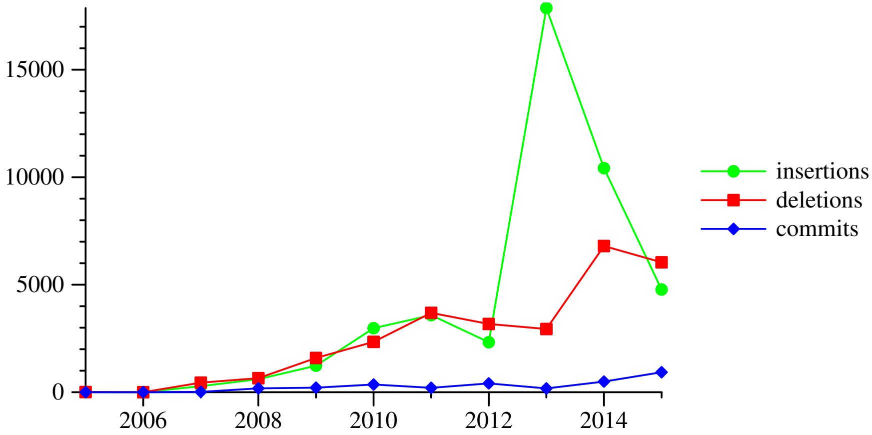
Methodology

`git log --grep ... linux`

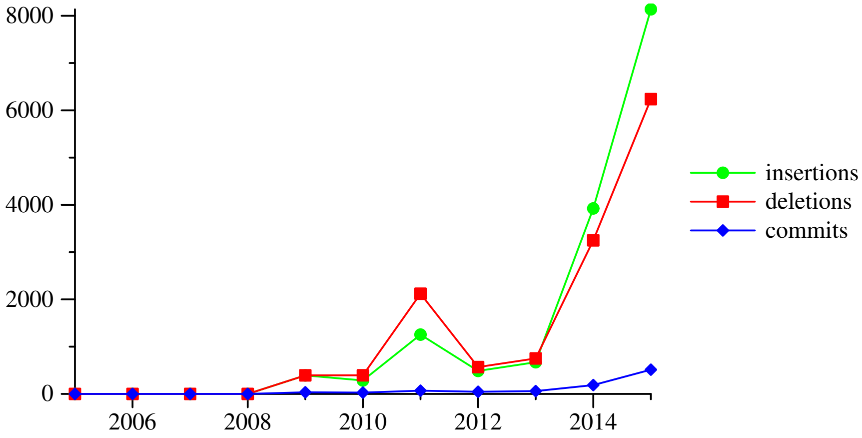
- occinelle
- semantic patch
- semantic match
- SmPL, SMPL, smpl

Use by year

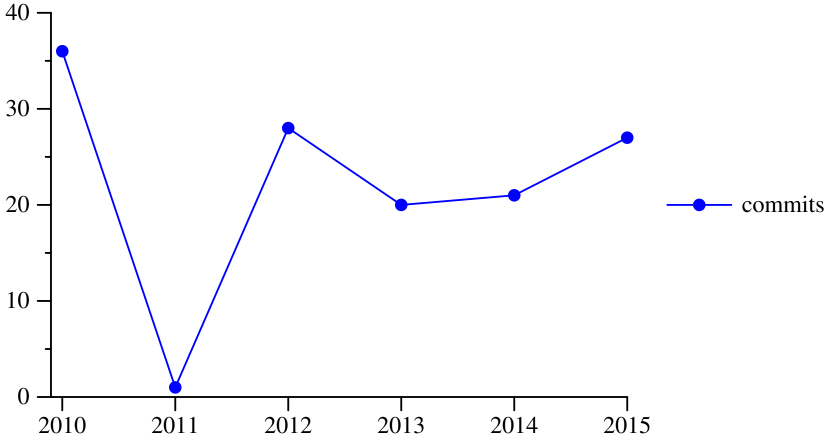
Whole kernel, excluding drivers/staging and scripts:



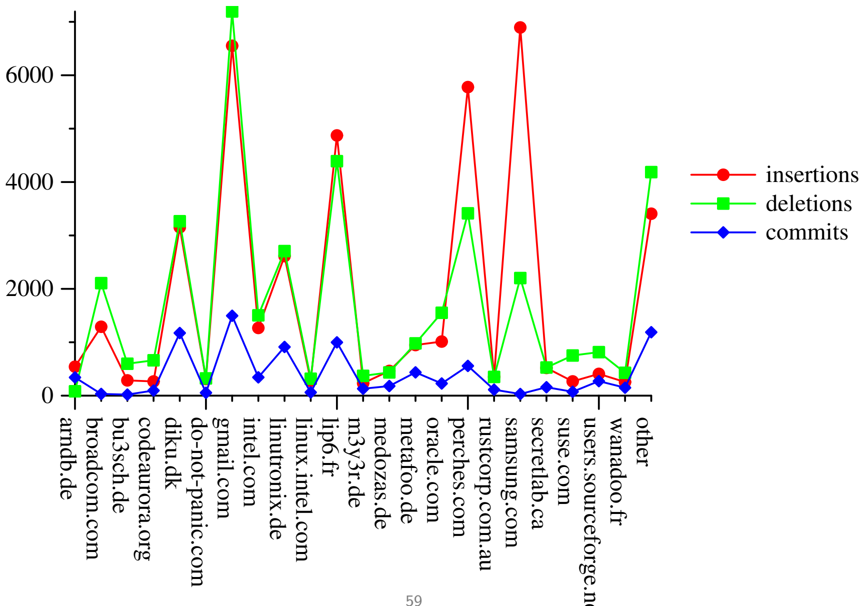
Use by year: drivers/staging



Use by year: scripts



Who is doing all of this work (up to Sep. 2015)?



Conclusion

Coccinelle: Pragmatic tool for scanning and transforming C code.

- Improves the reliability of maintenance tasks.
- Supports both code understanding and bug fixing.
- Also usable for software metrics.

Conclusion

Coccinelle: Pragmatic tool for scanning and transforming C code.

- Improves the reliability of maintenance tasks.
- Supports both code understanding and bug fixing.
- Also usable for software metrics.

Impact:

- Around 4000 Coccinelle-based patches accepted into Linux.
- Over 50 Coccinelle semantic patches available in the Linux source code.
- Applicable to other C software (wine, qemu, systemd, etc).

Conclusion

Coccinelle: Pragmatic tool for scanning and transforming C code.

- Improves the reliability of maintenance tasks.
- Supports both code understanding and bug fixing.
- Also usable for software metrics.

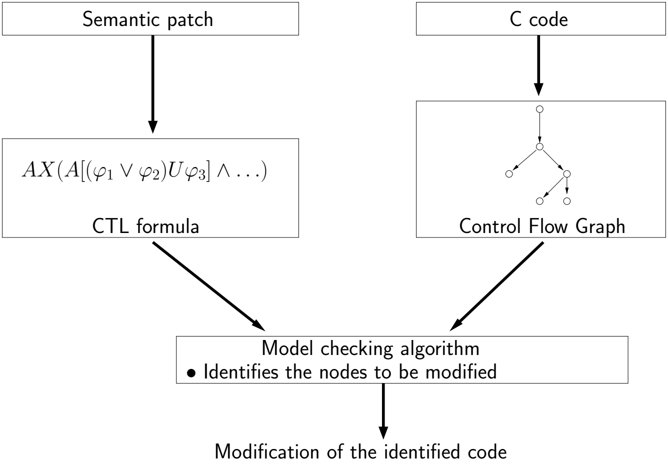
Impact:

- Around 4000 Coccinelle-based patches accepted into Linux.
- Over 50 Coccinelle semantic patches available in the Linux source code.
- Applicable to other C software (wine, qemu, systemd, etc).

http://coccinelle.lip6.fr/,
http://btrlinux.inria.fr/

How does it work?

Implementation overview



Matching via CTL [POPL'09]

Semantic patch rule $\Longrightarrow$ CTL formula

Source code function $\Longrightarrow$ control-flow graph

Provides reasoning about control-flow paths:

- `a ... b` transparently skips over gotos, around loops, etc.
- Forall (**A**) and exists (**E**) matching available.

CTL for Coccinelle

Extensions:

- Existentially quantified variables.
- Witnesses.

CTL for Coccinelle

Extensions:

- Existentially quantified variables.
- Witnesses.

Example:

```
@@ expression x; @@
a();
...
b(x);
```

$$\mathtt{a();} \wedge (\mathbf{AX}(\mathbf{A}[!(\mathtt{a();} \vee (\exists x, \mathtt{b(}x\mathtt{)}))\,\mathbf{U}\,(\exists x, \mathtt{b(}x\mathtt{)})]))$$