Practical Impacts of Variability in the Linux Kernel

Julia Lawall (Inria/LIP6)

March, 2017

Context

Linux is critical software.

- Used in embedded systems, desktops, servers, etc.

Linux is very large.

- Over 24 000 .c files
- Almost 15 million lines of C code in Linux 4.10.
- Increase of 56% since July 2011 (Linux 3.0).

Linux has both more and less experienced developers.

- Maintainers, contributors, developers of proprietary drivers

Context

Linux is critical software.

- Used in embedded systems, desktops, servers, etc.

Linux is very large.

- Over 24 000 .c files
- Almost 15 million lines of C code in Linux 4.10.
- Increase of 56% since July 2011 (Linux 3.0).

Linux has both more and less experienced developers.

- Maintainers, contributors, developers of proprietary drivers

Developers need reliable and precise information...

Goal: Automate bug finding and evolutions in C code

Find once, fix everywhere.

Approach: Coccinelle: `http://coccinelle.lip6.fr/`

- Static analysis to find patterns in C code.
- Automatic transformation to perform evolutions and fix bugs.
- User scriptable, based on patch notation (semantic patches).

Goal: Automate bug finding and evolutions in C code

Find once, fix everywhere.

Approach: Coccinelle: `http://coccinelle.lip6.fr/`

- Static analysis to find patterns in C code.
- Automatic transformation to perform evolutions and fix bugs.
- User scriptable, based on patch notation (semantic patches).

Goal: Be accessible to C code developers.

Example

Evolution: A new function: kzalloc (Linux 2.6.14)
⟹ Collateral evolution: Merge kmalloc and memset into kzalloc

```
fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (!fh) {
  dprintk(1,
    KERN_ERR
    "%s: zoran_open(): allocation of zoran_fh failed\n",
    ZR_DEVNAME(zr));
  return -ENOMEM;
}
memset(fh, 0, sizeof(struct zoran_fh));
```

Example

Evolution: A new function: kzalloc (Linux 2.6.14)
⟹ Collateral evolution: Merge kmalloc and memset into kzalloc

```
fh = kzalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (!fh) {
  dprintk(1,
    KERN_ERR
    "%s: zoran_open(): allocation of zoran_fh failed\n",
    ZR_DEVNAME(zr));
  return -ENOMEM;
}
```

A kmalloc → kzalloc semantic patch

```
@@
expression x, sz;
identifier f;
@@

x =
    kmalloc

        (sz, ...)
  ...

  memset(x, 0, sz);
```

A kmalloc → kzalloc semantic patch

```
@@
expression x, sz;
identifier f;
@@

x =
-   kmalloc

        (sz, ...)
  ...

- memset(x, 0, sz);
```

A kmalloc → kzalloc semantic patch

```
@@
expression x, sz;
identifier f;
@@

x =
-   kmalloc
+   kzalloc
        (sz, ...)
  ...

- memset(x, 0, sz);
```

A kmalloc → kzalloc semantic patch

```
@@
expression x, sz, E;
identifier f;
@@

x =
-   kmalloc
+   kzalloc
        (sz, ...)
  ...   when != (<+...x...+>) = E
        when != f(...,x,...)
- memset(x, 0, sz);
```

Results

- Correctly updates 14 occurrences
  - 5 false positives, could be eliminated by more "`when`" tests

Results

- Correctly updates 14 occurrences
  - 5 false positives, could be eliminated by more “`when`” tests
- Other opportunities:
  - `acpi_os_allocate` → `acpi_os_allocate_zeroed`
  - `dma_pool_alloc` → `dma_pool_zalloc`
  - `dma_alloc_coherent` → `dma_zalloc_coherent`
  - `kmem_cache_alloc` → `kmem_cache_zalloc`
  - `pci_alloc_consistent` → `pci_zalloc_consistent`
  - `vmalloc` → `vzalloc`
  - `vmalloc_node` → `vzalloc_node`

A more complex example: Constification

Motivation:

- The Linux kernel uses structures heavily
  - Many contain function pointers.
  - Analogous to OO classes.

A more complex example: Constification

Motivation:

- The Linux kernel uses structures heavily
  - Many contain function pointers.
  - Analogous to OO classes.
- Security risk:
  - Overwriting function pointers allows executing arbitrary code with kernel privileges.
  - Overwriting other values can lead to e.g. invalid device interactions, crashes, and DoS.

Structure usage: Example

```
static struct ethtool_ops hip04_ethtool_ops = {
        .get_coalesce               = hip04_get_coalesce,
        ... };
static struct net_device_ops hip04_netdev_ops = {
        .ndo_open                   = hip04_mac_open,
        ... };

static int hip04_mac_probe(struct platform_device *pdev) {
  struct net_device *ndev;
  ...
  ndev->netdev_ops = &hip04_netdev_ops;
  ndev->ethtool_ops = &hip04_ethtool_ops;
  ...
  ret = register_netdev(ndev);
  ...
}

static struct platform_driver hip04_mac_driver = {
        .probe  = hip04_mac_probe,
        ... };
module_platform_driver(hip04_mac_driver);
```

Structure usage: Example

```
static struct ethtool_ops hip04_ethtool_ops = {
        .get_coalesce              = hip04_get_coalesce,
        ... };
static struct net_device_ops hip04_netdev_ops = {
        .ndo_open                  = hip04_mac_open,
        ... };

static int hip04_mac_probe(struct platform_device *pdev) {
  struct net_device *ndev;
  ...
  ndev->netdev_ops = &hip04_netdev_ops;
  ndev->ethtool_ops = &hip04_ethtool_ops;
  ...
  ret = register_netdev(ndev);
  ...
}

static struct platform_driver hip04_mac_driver = {
        .probe  = hip04_mac_probe,
        ... };
module_platform_driver(hip04_mac_driver);
```

Structure usage: Example

```
static struct ethtool_ops hip04_ethtool_ops = {
        .get_coalesce           = hip04_get_coalesce,
        ... };
static struct net_device_ops hip04_netdev_ops = {
        .ndo_open               = hip04_mac_open,
        ... };

static int hip04_mac_probe(struct platform_device *pdev) {
  struct net_device *ndev;
  ...
  ndev->netdev_ops = &hip04_netdev_ops;
  ndev->ethtool_ops = &hip04_ethtool_ops;
  ...
  ret = register_netdev(ndev);
  ...
}

static struct platform_driver hip04_mac_driver = {
        .probe  = hip04_mac_probe,
        ... };
module_platform_driver(hip04_mac_driver);
```

Structure usage: Example

```
static struct ethtool_ops hip04_ethtool_ops = {
        .get_coalesce           = hip04_get_coalesce,
        ... };
static struct net_device_ops hip04_netdev_ops = {
        .ndo_open               = hip04_mac_open,
        ... };

static int hip04_mac_probe(struct platform_device *pdev) {
  struct net_device *ndev;
  ...
  ndev->netdev_ops = &hip04_netdev_ops;
  ndev->ethtool_ops = &hip04_ethtool_ops;
  ...
  ret = register_netdev(ndev);
  ...
}

static struct platform_driver hip04_mac_driver = {
        .probe  = hip04_mac_probe,
        ... };
module_platform_driver(hip04_mac_driver);
```

Constification: Generic approach

- Search for contexts where a structure is used
- Check for const types

Constification: Generic approach

- Search for contexts where a structure is used
- Check for const types
- Problem: Can be expensive
  - `net_device` is defined in include/linux/netdevice.h
  - Not the current file or an immediately included one.
  - Recursive includes are expensive.

Constification: Tailored approach

- Search for context where const structures of the same type are used.
- Check that the target structure is only used in these contexts.
- No need for header files.

Constification

```
@r disable optional_qualifier@
identifier i;
@@
static struct net_device_ops i = { ... };

@ok@
identifier r.i; struct net_device e; position p;
@@
e.netdev_ops = &i@p;

@bad@
position p != ok.p; identifier r.i;
@@
i@p

@depends on !bad disable optional_qualifier@
identifier r.i;
@@
static
+const
 struct net_device_ops i = { ... };
```

Constification

```
@r disable optional_qualifier@
identifier i;
@@
static struct net_device_ops i = { ... };

@ok@
identifier r.i; struct net_device e; position p;
@@
e.netdev_ops = &i@p;

@bad@
position p != ok.p; identifier r.i;
@@
i@p

@depends on !bad disable optional_qualifier@
identifier r.i;
@@
static
+const
 struct net_device_ops i = { ... };
```

Updated 8 drivers in Linux 4.5 (patches subsequently integrated)

What about variability?

Coccinelle applies a semantic patch to a complete code base

- Unaware of makefile constraints
- If #ifdefs are well structured, they are converted to if-like control flow structures.
- Undisciplined #ifdefs disappear or choose first branch.

Possible impacts

False positives, false negatives

- In a top-level declaration.
- In a function.
- Across functions.

Example:

```
const struct raid6_recov_calls raid6_recov_avx512 = {
        ...,
#ifdef CONFIG_X86_64
        .name = "avx512x2",
#else
        .name = "avx512x1",
#endif
        .priority = 3,
};
```

Variability issues within functions?

	Total fns	Fns w/ #if[n][def]	Fns w/ #else
.c files	403,801	12,998 (3%)	2,445 (0%)
.h files	37,955	1,084 (2%)	512 (1%)

Coccinelle parsing of #ifdefs in Linux 4.10 functions:

- 16,410 treated in a structured way
- 201 (0.012%) ignored or use the first branch

Variability issues within functions?

Common categories of functions containing ifdefs (Linux v4.10):
(reachable by unfolding 0, 1, or 2 calls)

Category	number
EXPORT_SYMBOL, arg 1	886 (3%)
EXPORT_SYMBOL_GPL, arg 1	490 (2%)
module_init, arg 1	484 (9%)
pci_driver.probe	367 (7%)
request_irq, arg 2	337 (7%)
platform_driver.probe	241 (3%)
module_exit, arg 1	207 (5%)
net_device_ops.ndo_open	148 (7%)
INIT_WORK, arg 2	125 (2%)
file_operations.unlocked_ioctl	105 (5%)

Variability issues across functions?

Function names with multiple non-static definitions:

	All in one file	All in one dir	One dir + arch
`arch`	90	1230	–
`drivers`	106	433	118
`kernel`	59	106	116
`mm`	28	121	34
`fs`	16	58	7

0.5% of Linux kernel function names have multiple non-static definitions

Variability issues across functions?

Inconsistent properties
(.c and .h files of Linux 4.10):

- 4 function names have definitions with inconsistent locking assumptions
  - All false positives
- 1 function name has all parameters const in all but one case.
- 1 function name has one instance that returns only ERR_PTR; the others can also return NULL
- 21 functions names have definitions that make inconsistent assumptions about whether an argument is NULL

Variability issues across functions?

Inconsistent properties
(.c and .h files of Linux 4.10):

- 4 function names have definitions with inconsistent locking assumptions
  - All false positives
- 1 function name has all parameters const in all but one case.
- 1 function name has one instance that returns only ERR_PTR; the others can also return NULL
- 21 functions names have definitions that make inconsistent assumptions about whether an argument is NULL

1% of function names with multiple definitions

Variability and the Coccinelle user

- Coccinelle makes it easy to make changes that may be hard to test.
- Compilation testing is often the only alternative.
- Variability means that not all changed lines may be subjected to compilation.

Our proposal: JMake [DSN 2017]

Automates:

- Choice of architecture
  - The Linux kernel configuration space is mostly determined by this choice.
- Mutation of changed lines, to verify that they are subjected to the compiler.
  - Ensure `.i` files contains the mutation
  - Ensure the unmutated file produces a `.o` file
  - Minimal mutations, to reduce validation effort

Results for kmalloc+memset → kzalloc

- 52 patches, introducing 133 kzallocs (Linux v3.0 - Linux v4.4)
- For 2 files (2 patches) unable to choose an architecture
- For 1 file (1 patch) under a configuration variable that is never defined in the kernel.
  - ifdef is far from the change site and easy to miss
- For 7 files (5 patches) cause unknown (no apparent ifdef).
  - Likely compilation issues
  - In 1 of these files, the change is under `#if NOT_YET`

For 85% of patches, all changed lines subjected to compilation.

Results for my constification patches

- 194 patches
- For 5 files (3 patches) there is no Makefile in the directory with the changed file.
- For 2 files (2 patches) unable to choose an architecture
- For 3 files (3 patches) a function has two possible headers, only one subjected to compilation (if/else problem)
- For 1 file (1 patch) 2 function headers for x86 and 2 for arm64

For 95% of patches, all changed lines subjected to compilation.

Conclusion

- Pattern based language for matching and transforming C code
- Coccinelle mentioned in over 4800 Linux kernel patches
  - Also used by wine, systemd, qemu, etc.
  - Some support for C++
- Configuration-independent
  - Only rarely a problem for practical usage cases.
- Current work: Automatic inference of transformation rules to automate driver backporting and forwardporting
  - PhD and postdoc positions available!

http://coccinelle.lip6.fr/