Is It Possible to Automatically Port Kernel Modules?

Yanjie Zhen, Wei Zhang, **Zhenyang Dai**, Junjie Mao, Yu Chen

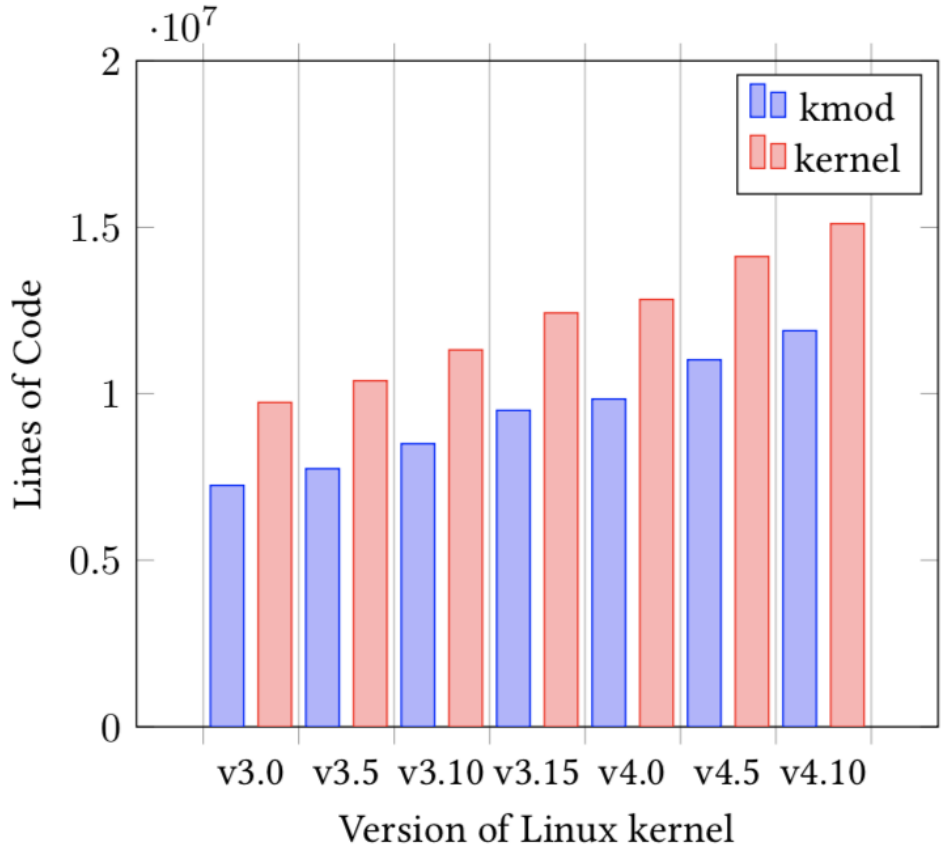
Tsinghua University

August 28, 2018

Outline

Kernel modules (kmods) are essential

- A flexible way to extend the functionality of Linux kernel.
- Over 70% of Linux source codes are kernel modules.



Porting kernel modules is necessary but hard

Porting kmods is necessary:

- Kernel interfaces (APIs) are not stable.
- Forward porting: to enjoy enhanced security, boosted performance and new functionality from the fast evolving kernel.
- Back porting: for stability/compatibility requirements.

But, it is also hard:

- The Linux kernel is large.
- It is complex.
- And it evolves fast.

Existing approaches for porting kernel modules

Manually:

- A trial-error-fix approach.
- Using "`git log`" or Google.

Existing tools:

- Based on automated program repair techniques: rely heavily on compilation error/warning messages
- Based on instance-based inference techniques: infer change patterns from source code modifications

Outline

Our work

An empirical study:

- Focus on the characteristics and representations of kernel API changes.
- Dataset: 200 patches related to 10 active kmods from Linux repository over the last 7 year.

Discussion:

- Feasibility and challenges of porting kmods automatically.
- Our insights in building effective automated tools.

Change patterns

Change patterns, or "patterns" for short, are a core concept of our work.

- Describe how the usage of an API should be updated.
- API changes $\xrightarrow{\text{cause}}$ "change instances" where the API is used $\xrightarrow{\text{infer}}$ change patterns.
- Provide reference for automatically porting.
- One patch may involve multiple patterns, and most patterns have multiple instances.

In our experiment, we infer change patterns manually. Semantic Patch Language (SmPL) is used as the representation of change patterns.

Change patterns: a concrete example

```
#define d_inode(d) ((d)->i)

-int f2(dentry *d) {
+int f2(dentry *d, inode *i){
-    if(d_inode(d)->flags & F2) { ... }
+    if(i->flags & F2) { ... }
-    return d->d_sb->mode;
+    return dsb(d)->mode;
}
struct ops ops1 = { .f1 = f1, .f2 = f2,

-int f3(dentry *x) {
+int f3(dentry *x, inode *i) {
     ...
-    return __f3(x->parent->d_sb);
+    return __f3(dsb(x->parent));
}
struct ops ops2 = { .f2 = f3, };

-int f4(dentry *d) {
+int f4(dentry *d, inode *i) {
     int ret;
-    if (d_inode(d)->flags & F3) { ... }
+    if (i->flags & F3) { ... }
     return ret;
}
struct ops ops3 = { .f2 = f4, };
```

- f is a function having an argument d of type `dentry*`. f acts as the `f2` callback field of some `ops struct`.
- For such f, add a new parameter i with type `inode*`, and replace `d_inode(`d`)->flags` with i`->flags`.
- Replace e`->d_sb` with `dsb(`e`)`, where e is any expression.

Change patterns: a concrete example

```
#define d_inode(d) ((d)->i)

-int f2(dentry *d) {
+int f2(dentry *d, inode *i){
-    if(d_inode(d)->flags & F2) { ... }
+    if(i->flags & F2) { ... }
-    return d->d_sb->mode;
+    return dsb(d)->mode;
}
struct ops ops1 = { .f1 = f1, .f2 = f2,

-int f3(dentry *x) {
+int f3(dentry *x, inode *i) {
     ...
-    return __f3(x->parent->d_sb);
+    return __f3(dsb(x->parent));
}
struct ops ops2 = { .f2 = f3, };

-int f4(dentry *d) {
+int f4(dentry *d, inode *i) {
     int ret;
-    if (d_inode(d)->flags & F3) { ... }
+    if (i->flags & F3) { ... }
     return ret;
}
struct ops ops3 = { .f2 = f4, };
```

```
@rule@
identifier s, f;
@@
struct ops s = { .f2 = f, };
@@
typedef dentry, inode;
identifier rule.f;
identifier _arg1, _arg2;
@@
- f (dentry *_arg1) {
+ f (dentry *_arg1, inode *_arg2) {
<...
-    d_inode(_arg1)->flags
+    _arg2->flags
...>
}

@@
expression E
@@
- E->d_sb
+ dsb(E)
}
```

Outline

Dataset

- 10 kernel modules from `fs/` and `drivers/`
- 1753 patches related to API changes
- 200 patches chosen randomly due to limited time
- 407 patterns inferred manually

Target modules	Commits
fs/btrfs	282
fs/ext2	141
fs/ext4	342
fs/jfs	104
fs/xfs	219
drivers/block/drbd	137
drivers/gpu/drm/radeon	291
drivers/net/.../ixgbe	68
drivers/usb/gadget	367
drivers/input/keyboard	130
Total	2081

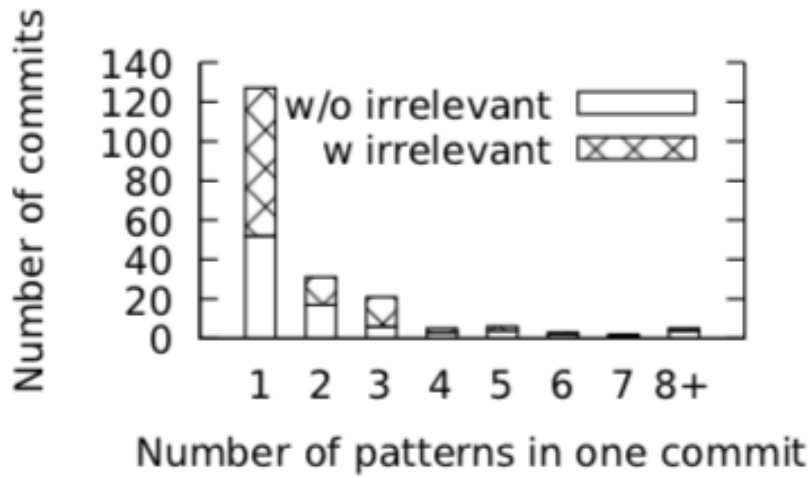
Questions to answer

Validity of the assumptions on inputs i.e. patches

- How many change patterns are there in a single commit?
- How many instances does each pattern have?
- Does the compiler always complain about out-of-date API usages?

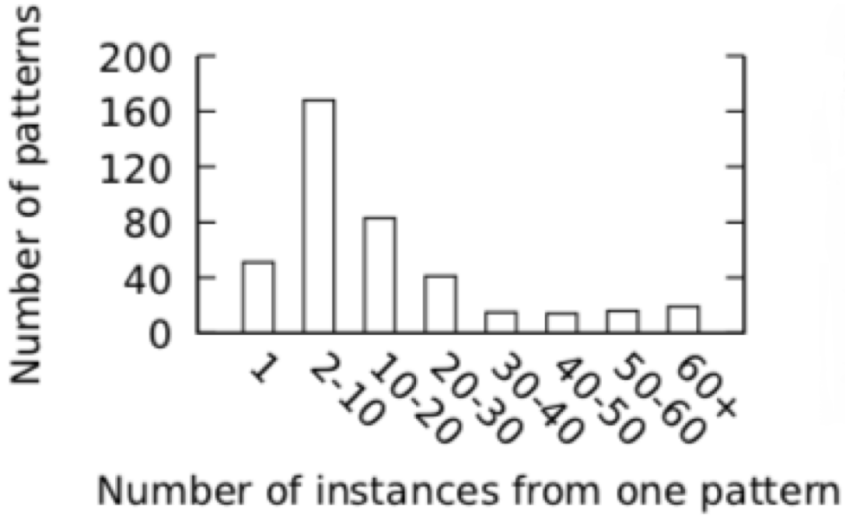
Q1-1: How many change patterns are there in a single commit?

- 36.5% (73 out of 200) of the commits involve multiple patterns
- 111 commits (55.5%) contain changes irrelevant to API usage updates $\rightarrow$ *w irrelevant*



Q1-2: How many instances does each pattern have?

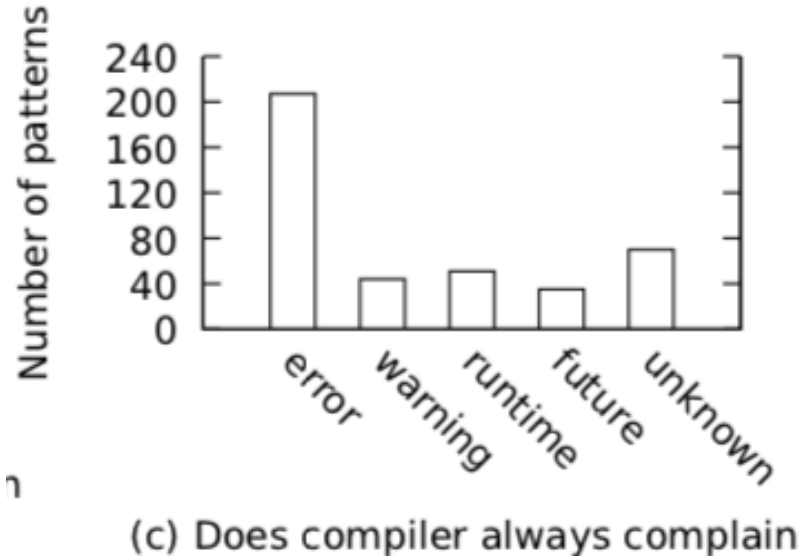
- 87.5% of patterns (356 out of 407) have multiple instances.



Q1-3: Does the compiler always complain about out-of-date API usages?

When some pattern is missed,

- 51 patterns (12.5%) change runtime behavior without triggering any compiler message $\rightarrow$ *runtime*
- 105 patterns (25.8%) neither cause compiler error/warning nor alter program runtime behavior $\rightarrow$ *future & unknown*



(c) Does compiler always complain

Questions to answer

Validity of the assumptions on inputs i.e. patches

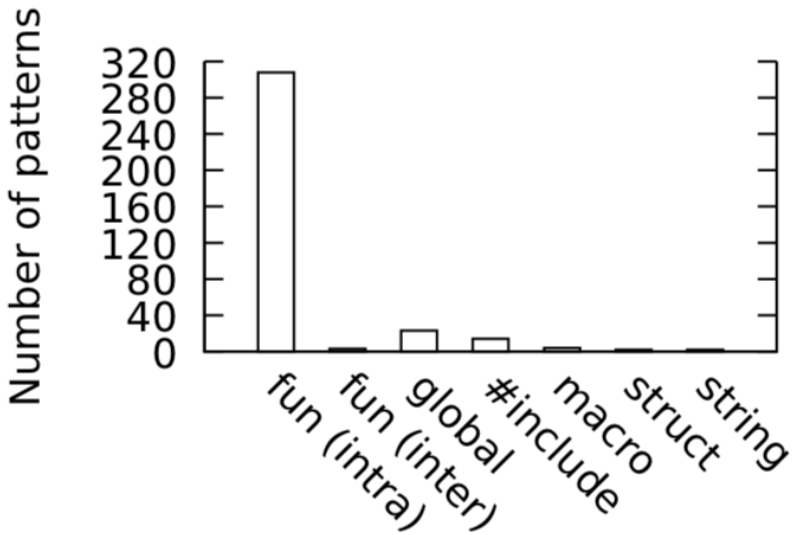
- How many change patterns are there in a single commit?
- How many instances does each pattern have?
- Does the compiler always complain about out-of-date API usages?

Representation of changes (within 356 patterns with multiple instances)

- What kinds of code are changed by the patterns?
- How often do the patterns involve macro invocations?
- What kinds of contexts do the patterns require?

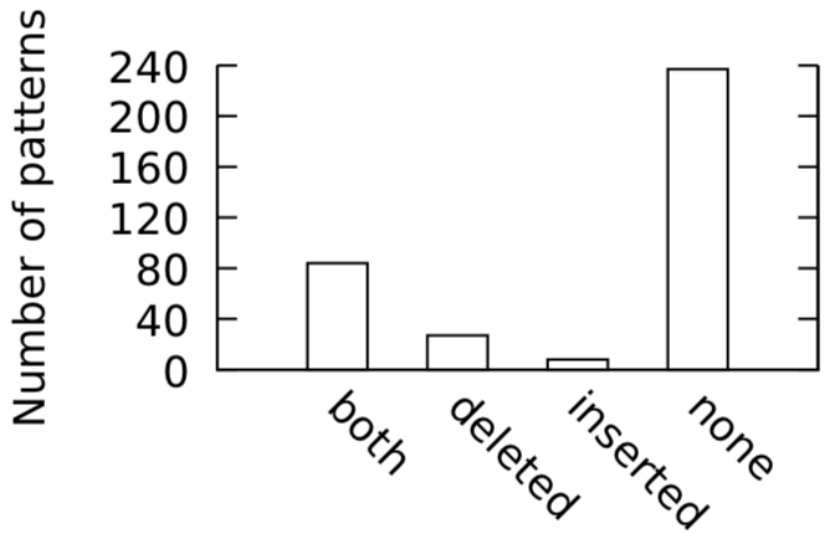
Q2-1: What kinds of code are changed by the patterns?

- 86.5% of the patterns (308 out of 356) touch function definitions. $\rightarrow$ *fun (intra) & fun (inter)*
- 6.4% of the patterns (23 out of 356) update definitions of global variables. $\rightarrow$ *global*



Q2-2: How often do the patterns involve macro invocations?

- 33.4% (119 out of 356) of the patterns involve macro invocations. $\rightarrow$ *both & deleted & inserted*



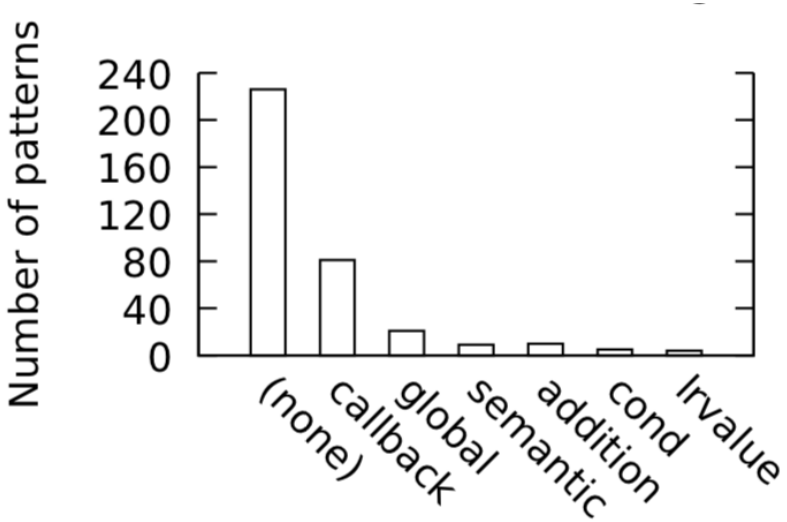
Change patterns: a concrete example

```
#define d_inode(d) ((d)->i)

-int f2(dentry *d) {
+int f2(dentry *d, inode *i){
-    if(d_inode(d)->flags & F2) { ... }
+    if(i->flags & F2) { ... }
-    return d->d_sb->mode;
+    return dsb(d)->mode;
}
struct ops ops1 = { .f1 = f1, .f2 = f2,

-int f3(dentry *x) {
+int f3(dentry *x, inode *i) {
     ...
-    return __f3(x->parent->d_sb);
+    return __f3(dsb(x->parent));
}
struct ops ops2 = { .f2 = f3, };

-int f4(dentry *d) {
+int f4(dentry *d, inode *i) {
     int ret;
-    if (d_inode(d)->flags & F3) { ... }
+    if (i->flags & F3) { ... }
     return ret;
}
struct ops ops3 = { .f2 = f4, };
```

```
@rule@
identifier s, f;
@@
struct ops s = { .f2 = f, };
@@
typedef dentry, inode;
identifier rule.f;
identifier _arg1, _arg2;
@@
- f (dentry *_arg1) {
+ f (dentry *_arg1, inode *_arg2) {
<...
-    d_inode(_arg1)->flags
+    _arg2->flags
...>
}

@@
expression E
@@
- E->d_sb
+ dsb(E)
}
```

Q2-3: What kinds of contexts do the patterns require?

- 36.5% (130 out of 356) of changes are context dependent. $\rightarrow$ *from the second to the last column*
- Callbacks (81 out of 356) and global variables (21 out of 356) account for the majority (78.5%). $\rightarrow$ *callback & global*



Outline

Discussion

Feasibility:

- Based on compilation messages: severely limited
- Based on pattern inference: promising

Challenges:

- Noises: multiple patterns in a single commit; modification of irrelevant code
- Macros: infer pattern with macros
- Contexts: especially callbacks and global variables

Thank you!
More in the paper!
Any questions?