SPINFER: Inferring Semantic Patches for the Linux Kernel

Lucas SERRANO, Van-Anh NGUYEN, Ferdian THUNG
Lingxiao JIANG, David LO, Julia LAWALL, Gilles MULLER

Maintenance of the Linux kernel

Maintenance tasks are very common in all software projects.

Maintenance of the Linux kernel

Maintenance tasks are very common in all software projects.

These tasks can consist of:

- Refactoring portions of code
- Cleaning dead code
- Migrating APIs to new version

Maintenance of the Linux kernel

Maintenance tasks are very common in all software projects.

These tasks can consist of:

- Refactoring portions of code
- Cleaning dead code
- Migrating APIs to new version

But maintaining the Linux kernel is particularly hard:

- 18M lines of C code
- 13M lines of driver code
- The same kernel API can be used by thousands of files

Even simple API migrations can be difficult to do

Motivating Example

Example of API migration

Example of low-resolution timer structure initialization:

- Originally with the `init_timer` function
- Since 2006 with `setup_timer`

Example of API migration

Example of low-resolution timer structure initialization:

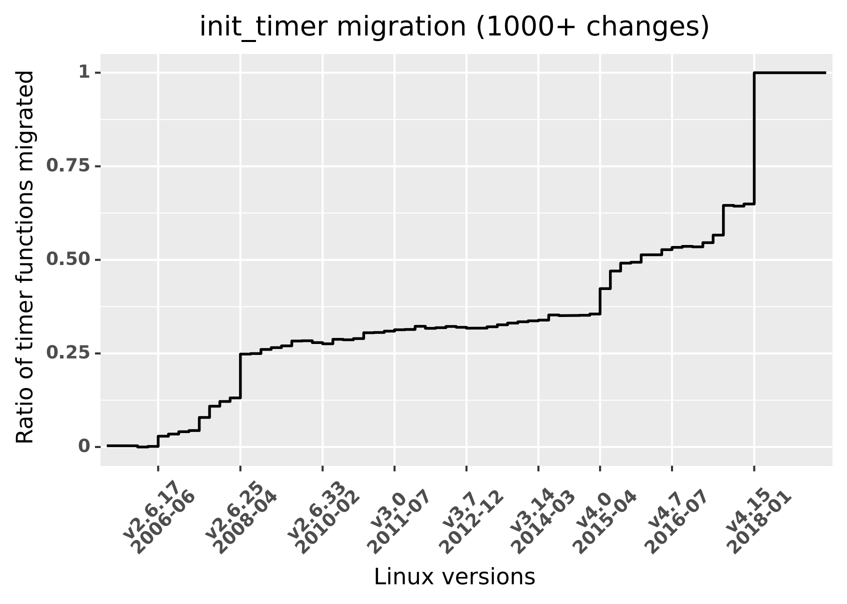
- Originally with the `init_timer` function
- Since 2006 with `setup_timer`

Old function was not removed, the migration was not mandatory.

init_timer migration

```
drivers/atm/nicstar.c
@@ -284,10 +284,8 @@ static int __init nicstar_init(void)
-        init_timer(&ns_timer);
+        setup_timer(&ns_timer, ns_poll, 0UL);
         ns_timer.expires = jiffies + NS_POLL_PERIOD;
-        ns_timer.data = 0UL;
-        ns_timer.function = ns_poll;
```

```
drivers/gpu/drm/omapdrm/dss/dsi.c
@@ -5449,9 +5449,7 @@ static int dsi_bind(struct device *dev,
-    init_timer(&dsi->te_timer);
-    dsi->te_timer.function = dsi_te_timeout;
-    dsi->te_timer.data = 0;
+    setup_timer(&dsi->te_timer, dsi_te_timeout, 0);
```



init_timer migration (1000+ changes)
Ratio of timer functions migrated
1
0.75
0.50
0.25
0
v2.6.17 2006-06
v2.6.25 2008-04
v2.6.33 2010-02
v3.0 2011-07
v3.7 2012-12
v3.14 2014-03
v4.0 2015-04
v4.7 2016-07
v4.15 2018-01
Linux versions

Automation

In 2018 these interfaces were considered insecure and were both replaced.

But at this time API usage was in inconsistent state:

- 60% using the new `setup_timer`
- 40% using the old `init_timer`

Automation

In 2018 these interfaces were considered insecure and were both replaced.

But at this time API usage was in inconsistent state:

- 60% using the new `setup_timer`
- 40% using the old `init_timer`

Could the transformation have been done automatically?

First contribution:
Taxonomy of transformation challenges

Related work

There are a lot of tools to perform API migration by learning from examples: REFAZER, LASE, AppEvolve, Meditor, . . .

But it was hard to know what kind of transformation they could handle.

Our first contribution is to classify transformation challenges.

Transformation challenges taxonomy

Challenges can be organized in 5 main categories:

Transformation challenges taxonomy

Challenges can be organized in 5 main categories:

1. Control-flow dependencies

Transformation challenges taxonomy

Challenges can be organized in 5 main categories:

1. Control-flow dependencies
2. Data-flow dependencies

Transformation challenges taxonomy

Challenges can be organized in 5 main categories:

1. Control-flow dependencies
2. Data-flow dependencies
3. Number of variants

Transformation challenges taxonomy

Challenges can be organized in 5 main categories:

1. Control-flow dependencies
2. Data-flow dependencies
3. Number of variants
4. Number of instances

Transformation challenges taxonomy

Challenges can be organized in 5 main categories:

1. Control-flow dependencies
2. Data-flow dependencies
3. Number of variants
4. Number of instances
5. Presence of unrelated changes

Need for a new tool

We found that all tools cannot handle transformation that:

- Require control-flow dependencies
- Have multiple variants

Need for a new tool

We found that all tools cannot handle transformation that:

- Require control-flow dependencies
- Have multiple variants

Both of these constraints are common in Linux kernel transformations.

And they were necessary for our timer example.

Need for a new tool

We found that all tools cannot handle transformation that:

- Require control-flow dependencies
- Have multiple variants

Both of these constraints are common in Linux kernel transformations.

And they were necessary for our timer example.

Moreover transformation rules used by these tools are not exposed
Meaning that developers cannot check if the transformation will be correct.

Second contribution: Spinfer

A tool suitable for the Linux kernel

To perform API migration in the Linux kernel we want a tool that:

- Learns transformation from examples
- Handles both control-flow dependencies and transformation variants
- Exposes transformation rules to developers

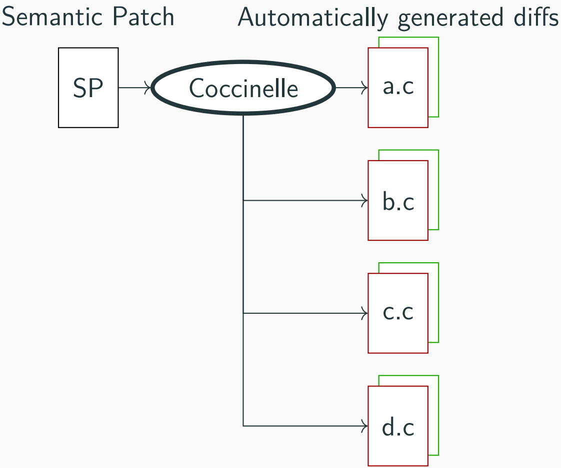
Transformation rules

Fortunately, a transformation rules language is already used in the Linux kernel.

Since 2008 Coccinelle rules are used to perform some transformations.

Even used in our motivating example.

Coccinelle


Semantic Patch
Automatically generated diffs
SP
Coccinelle
a.c
b.c
c.c
d.c

Semantic patch

```
@@
expression E0, E1, E2;
@@
- init_timer(E0);
+ setup_timer(E0, E1, E2);
...
- E0.data = E2;
- E0.function = E1;
```

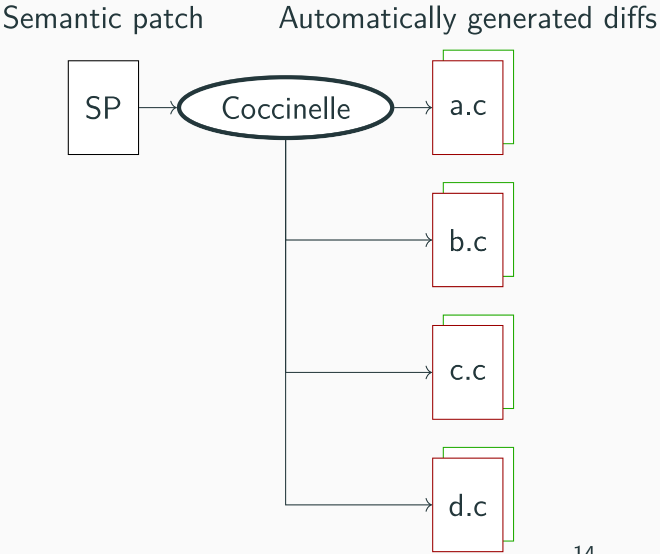
Semantic patch

```
@@
expression E0, E1, E2;
@@
- init_timer(E0);
+ setup_timer(E0, E1, E2);
...
- E0.data = E2;
- E0.function = E1;
```

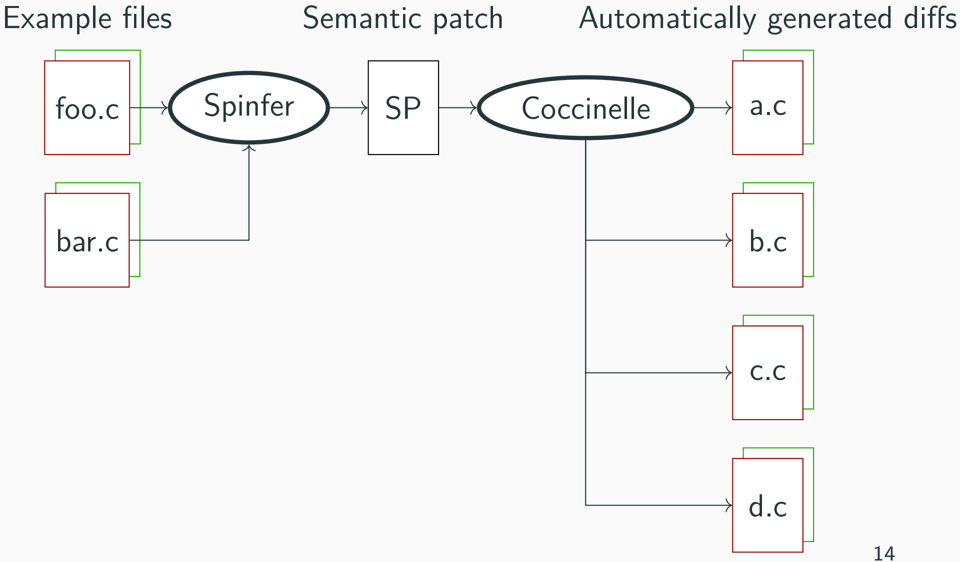
Generates diffs like this:

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_P_P;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;
```

Our approach: Spinfer



Our approach: Spinfer



Infering semantic patches

How to convert transformation instances...

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_P_P;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;
```

... to a semantic patch.

```
@@
expression E0, E1, E2;
@@
- init_timer(E0);
+ setup_timer(E0, E1, E2);
...
- E0.data = E2;
- E0.function = E1;
```

1: Extracting modified statements

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_POLL_PERIOD;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;

- init_timer(&dsi->te_timer);
- dsi->te_timer.function = dsi_te_timeout;
- dsi->te_timer.data = 0;
+ setup_timer(&dsi->te_timer, dsi_te_timeout, 0);
```

1: Extracting modified statements

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_POLL_PERIOD;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;

- init_timer(&dsi->te_timer);
- dsi->te_timer.function = dsi_te_timeout;
- dsi->te_timer.data = 0;
+ setup_timer(&dsi->te_timer, dsi_te_timeout, 0);
```

2: Clustering similar statements

```
- init_timer(&ns_timer);
- init_timer(&dsi->te_timer);

+ setup_timer(&ns_timer, ns_poll, 0UL);
+ setup_timer(&dsi->te_timer, dsi_te_timeout, 0);

- ns_timer.data = 0UL;
- dsi->te_timer.data = 0;

- ns_timer.function = ns_poll;
- dsi->te_timer.function = dsi_te_timeout;
```

3: Abstracting clusters

\- init_timer(*Expr*);

```
- init_timer(&ns_timer);
- init_timer(&dsi->te_timer);
```

\+ setup_timer(*Expr*, *Expr*, *Expr*);

```
+ setup_timer(&ns_timer, ns_poll, 0UL);
+ setup_timer(&dsi->te_timer, dsi_te_timeout, 0);
```

\- *Expr*.data = *Expr*;

```
- ns_timer.data = 0UL;
- dsi->te_timer.data = 0;
```

\- *Expr*.function = *Expr*;

```
- ns_timer.function = ns_poll;
- dsi->te_timer.function = dsi_te_timeout;
```

4: Assembling abstractions

```
- init_timer(Expr);
- Expr.function = Expr;
- Expr.data = Expr;
+ setup_timer(Expr, Expr, Expr);
```

4: Assembling abstractions

```
- init_timer(Expr);
- Expr.function = Expr;
- Expr.data = Expr;
+ setup_timer(Expr, Expr, Expr);
```

Spinfer takes a first abstraction

- init_timer(*Expr*);

4: Assembling abstractions

```
- init_timer(Expr);
- Expr.function = Expr;
```

```
- Expr.data = Expr;
+ setup_timer(Expr, Expr, Expr);
```

It extends rules using control-flow dependencies

```
- init_timer(Expr);
...
- Expr.function = Expr;
```

5: Rule splitting

When there are inconsistencies in control-flow, rules are split:

```
- init_timer(Expr);
...
- Expr.data = Expr;
- Expr.function = Expr;
```

```
- init_timer(Expr);
...
- Expr.function = Expr;
- Expr.data = Expr;
```

This allows Spinfer to discover transformation variants.

6: Iterating

This process goes on until all abstractions are exhausted.

```
- init_timer(Expr);
+ setup_timer(Expr, Expr, Expr);
...
- Expr.data = Expr;
- Expr.function = Expr;
```

```
- init_timer(Expr);
+ setup_timer(Expr, Expr, Expr);
...
- Expr.function = Expr;
- Expr.data = Expr;
```

7: Metavariable discovery

To obtain a valid rule Spinfer transforms abstractions into metavariables:

A unique name is chosen for each set of terms found in the examples.

```
- init_timer(Expr);
+ setup_timer(Expr, Expr, Expr);
...
- Expr.data = Expr;
- Expr.function = Expr;
```

```
@@
expression E0, E1, E2;
@@
- init_timer(E0);
+ setup_timer(E0, E1, E2);
...
- E0.data = E2;
- E0.function = E1;
```

Obtained semantic patch

Spinfer obtained these two rules:

```
@@
expression E0, E1, E2;
@@
- init_timer(E0);
+ setup_timer(E0, E1, E2);
...
- E0.data = E2;
- E0.function = E1;
```

```
@@
expression E0, E1, E2;
@@
- init_timer(E0);
+ setup_timer(E0, E1, E2);
...
- E0.function = E1;
- E0.data = E2;
```

Evaluation

Evaluation

We evaluated Spinfer by learning real Linux kernel transformations.

We extracted two datasets of 40 groups of transformation each:

- One selected to be challenging
- Another randomly sampled from changes in 2018

We compared the results produced by Spinfer generated semantic patches to the results produced by a human written semantic patch.

Results on the randomly sampled dataset

Spinfer was learning on one part of the changes and evaluated on the other part.

Learning set was 10 files or half the dataset.

Results on the randomly sampled dataset

Spinfer was learning on one part of the changes and evaluated on the other part.

Learning set was 10 files or half the dataset.

Two metrics:

- Precision: fraction of changes produced that were correct
- Recall: fraction of needed changes that were produced

Results on the randomly sampled dataset

Spinfer was learning on one part of the changes and evaluated on the other part.

Learning set was 10 files or half the dataset.

Two metrics:

- Precision: fraction of changes produced that were correct
- Recall: fraction of needed changes that were produced

Spinfer obtained 87% precision and 62% recall in average.

In 8 cases Spinfer obtained a perfect semantic patch.

More experiments on the paper

Conclusion

Spinfer learns semantic patches from examples.

It can learn transformations variants with many constraints such as:

- Control-flow dependencies
- Data-flow dependencies
- Transformation variants

Conclusion

Spinfer learns semantic patches from examples.

It can learn transformations variants with many constraints such as:

- Control-flow dependencies
- Data-flow dependencies
- Transformation variants

It uses code clustering to find similar pieces of code and abstract them.

Abstractions are assembled using control-flow information.

Conclusion

Spinfer learns semantic patches from examples.

It can learn transformations variants with many constraints such as:

- Control-flow dependencies
- Data-flow dependencies
- Transformation variants

It uses code clustering to find similar pieces of code and abstract them.

Abstractions are assembled using control-flow information.

Produced semantic patches can be checked and fixed by developers.

Closing

Thank you

If you have more questions:

Lucas.Serrano@lip6.fr