Coccinelle: Empowering developers to make large-scale changes to their software

Julia Lawall (Inria-Paris, Whisper team, Julia.Lawall@inria.fr)

June 26, 2023

Software lifecycle

- As software matures, it often grows.

Software lifecycle

- As software matures, it often grows.
- Code growth often reveals the need for new design decisions.

Software lifecycle

- As software matures, it often grows.
- Code growth often reveals the need for new design decisions.
- Implementing new design decisions gets more and more costly as the code gets bigger.

Software lifecycle

- As software matures, it often grows.
- Code growth often reveals the need for new design decisions.
- Implementing new design decisions gets more and more costly as the code gets bigger.
- Our focus: The Linux kernel.
  - 122K LOC in 1994 (v1.0).
  - 24M LOC today (v6.3).

Context of this talk

The Kernel Self Protection Project (KSPP)

- Goal: eliminate classes of bugs and methods of exploitation.
- Introduce compiler features and adjust the code base pervasively to reduce the attack surface.

All examples in this talk are motivated by large-scale transformations done by Kees Cook and Gustavo Silva of the KSPP.

Some examples

Example: introducing `array_size`

Problem: Allocating an array typically involves a multiplication:

```
height * width
```

Example: introducing `array_size`

Problem: Allocating an array typically involves a multiplication:

```
height * width
```

Multiplications can overflow.

Example: introducing `array_size`

Problem: Allocating an array typically involves a multiplication:

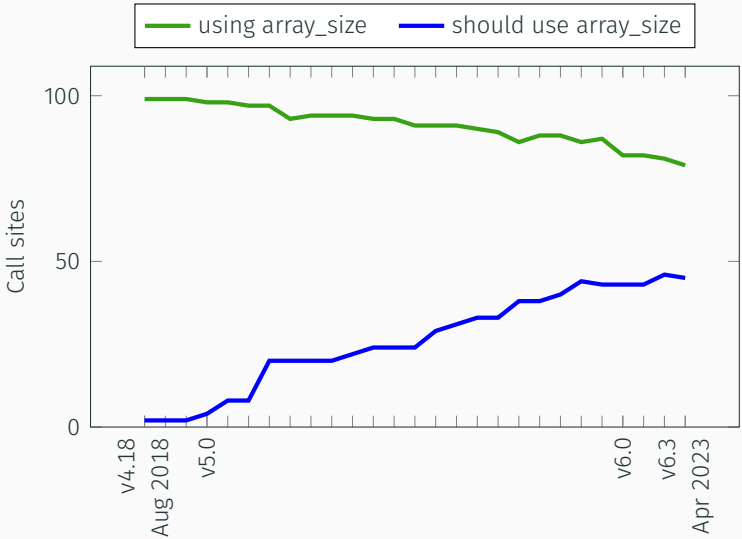
```
height * width
```

Multiplications can overflow.

Solution (since May 2018, v4.18): `array_size(h,w)`:

- Returns the product of the arguments, or `SIZE_MAX` on overflow.
- An allocation of size `SIZE_MAX` will fail, triggering error handling.
- Example: `vzalloc(array_size(num_pages, sizeof(*zram->table))`.

`array_size` uses in calls to `vzalloc` over time



Example: `init_timer` → `setup_timer`

Initializing a timer requires:

- The callback function to run when the timer expires
- The data that should be passed to that callback function

Example: `init_timer` → `setup_timer`

Initializing a timer requires:

- The callback function to run when the timer expires
- The data that should be passed to that callback function

Original initialization strategy (present in Linux v1.2.0):

```
init_timer(&ns_timer);
ns_timer.data = 0UL;
ns_timer.function = ns_poll;
```

Example: `init_timer` → `setup_timer`

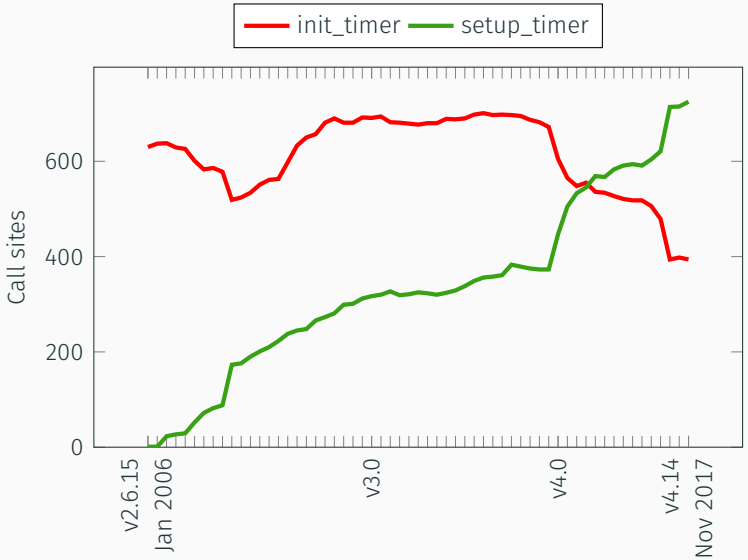
Replacement initialization strategy (introduced in Linux v2.6.15, Jan. 2006):

```
setup_timer(&ns_timer, ns_poll, 0UL);
```

Advantages:

- More concise
- More uniform

Example: `init_timer` → `setup_timer`



Example: flexible arrays

```
struct bcm6345_l1_cpu {
        void __iomem            *map_base;
        unsigned int            parent_irq;
        u32                     enable_cache[];
};
```

- The array `enable_cache` can have any size (flexible).
- The size is chosen at allocation time.
- Originally, not supported in C.

Simulating flexible arrays

Array of size 1:

```
struct bcm6345_l1_cpu {
        ...
        u32 enable_cache[1];
};
```

- Leads to off-by-one errors for `sizeof`.
- Problematic for `memcpy` bounds checking.

Simulating flexible arrays

Array of size 0:

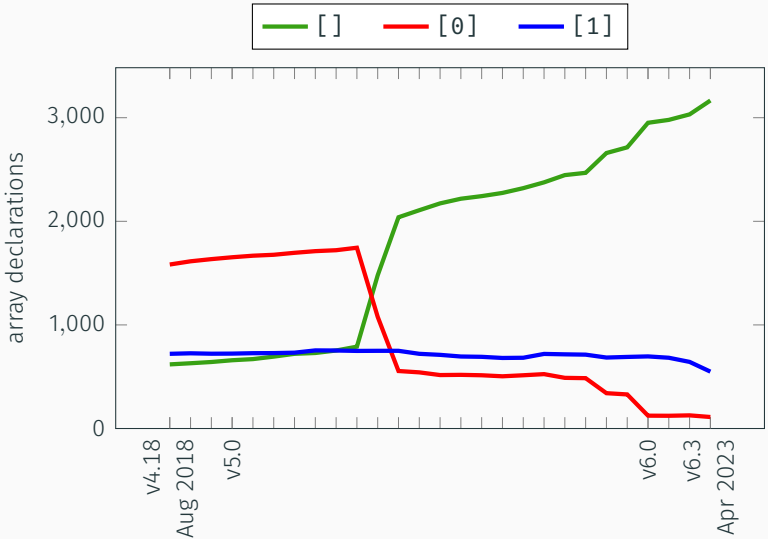
```
struct bcm6345_l1_cpu {
        ...
        u32 enable_cache[0];
};
```

- `sizeof` gives the array size 0.
- Also problematic for `memcpy` bounds checking.

Flexible array goals

- Convert arrays at the end of structure types declared as `[0]` or `[1]` to `[]`.
- Requires checking uses of these types, especially with `sizeof` and `memcpy`.
  - Not fully systematic.

Flexible array uses over time



Assessment

- Treewide changes are needed, touching up to 1000 or more code sites.
  - Issues recur, hence a need for recording expertise and automation.
- Some changes are systematic, but tedious and error prone.
  - `array_size`, `init_timer`
- Some require study of each case, but can benefit from systematically collected information.
  - flexible arrays

Coccinelle to the rescue!

What is Coccinelle?

- Pattern-based tool for matching and transforming C code
- Under development since 2005. Open source since 2008.
- Allows code changes to be expressed using patch-like code patterns (semantic patches).
- Goal: fit with the existing habits of the Linux programmer.

Semantic patches

Code fragments, annotated with $-$ and $+$.

- Like patches, but independent of irrelevant details
  (line numbers, spacing, variable names, etc.)
- Configuration independent (does not use the C preprocessor).

Semantic patches are written using the Semantic Patch Language (SmPL).

A SimPLe SamPLe of a semantic patch

Patch:

```
@@ -832,7 +832,7 @@
  */
- f(0,12);
+ f_zero(12);
  bdx_restore_mac(ndev);
```

Semantic patch:

```
@@
expression e;
@@
- f(0,e);
+ f_zero(e);
```

A SimPLe SamPLe of a semantic patch

Patch:

```
@@ -832,7 +832,7 @@
  */
- f(0,12);
+ f_zero(12);
  bdx_restore_mac(ndev);
```

Semantic patch:

```
@@
expression e;
@@
- f(0,e);
+ f_zero(e);
```

- `e` matches any expression.
- The semantic-patch rule replaces any call to `f` by a call to `f_zero`, when `f` has `0` as the first argument.

Developing a semantic patch: Working forwards from examples

Developing an `array_size` semantic patch

A patch:

```
diff --git a/drivers/block/zram/zram_drv.c b/drivers/block/zram/zram_drv.c
--- a/drivers/block/zram/zram_drv.c
+++ b/drivers/block/zram/zram_drv.c
@@ -898,7 +898,7 @@ static bool zram_meta_alloc(struct zram *zram, u64 disksize)
        size_t num_pages;

        num_pages = disksize >> PAGE_SHIFT;
-       zram->table = vzalloc(num_pages * sizeof(*zram->table));
+       zram->table = vzalloc(array_size(num_pages, sizeof(*zram->table)));
        if (!zram->table)
                return false;
```

Step 1: Drop irrelevant statements

```
-       zram->table = vzalloc(num_pages * sizeof(*zram->table));
+       zram->table = vzalloc(array_size(num_pages, sizeof(*zram->table)));
```

Step 2: Drop irrelevant context

```
-                       vzalloc(num_pages * sizeof(*zram->table))
+                       vzalloc(array_size(num_pages, sizeof(*zram->table)))
```

- Previously, the `vzalloc` call had to appear in an assignment statement.
- After dropping the context, it can appear as any subexpression.

Step 3: Introduce metavariables

```
@@
expression E1, E2;
@@

-                       vzalloc(E1 * E2)
+                       vzalloc(array_size(E1, E2))
```

Linux v4.17: Updates 102 calls across 67 files.

Developing an `init_timer` → `setup_timer` semantic patch

A patch: derived from drivers/atm/nicstar.c

```
-               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;
```

Step 1: Remove irrelevant code

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

Step 2: Abstract over subterms

```
@@
expression timer, fn_arg, data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...
-               timer.data = data_arg;
-               timer.function = fn_arg;
```

Step 3: Generalize a little more

```
@@
expression timer, fn_arg, data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...
-               timer.data = data_arg;
                ...
-               timer.function = fn_arg;
```

Results

Dataset: 598 Linux kernel `init_timer` files from different versions.

- 828 calls.
- Our semantic patch updates 308 of them.

Results

Dataset: 598 Linux kernel `init_timer` files from different versions.

- 828 calls.
- Our semantic patch updates 308 of them.

Untreated example: drivers/tty/n_gsm.c:

```
init_timer(&dlci->t1);
dlci->t1.function = gsm_dlci_t1;
dlci->t1.data = (unsigned long)dlci;
```

Extending the `init_timer` semantic patch

```
@@
expression timer, fn_arg, data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...

-               timer.data = data_arg;
                ...
-               timer.function = fn_arg;
```

Extending the `init_timer` semantic patch

```
@@
expression timer, fn_arg, data_arg;
@@
-               init_timer(&timer);
+               setup_timer(&timer, fn_arg, data_arg);
                ...
(
-               timer.data = data_arg;
                ...
-               timer.function = fn_arg;
|
-               timer.function = fn_arg;
                ...
-               timer.data = data_arg;
)
```

Covers 656/828 calls.

Results

Remaining issues

- Some code initializes `function` and `data` before calling `init_timer`.
- Some timers have no data initialization, default to 0.
- Coccinelle sometimes times out.

Complete semantic patch

- 6 rules, 68 lines of code.
- Covers 808/828 calls.
- TODO: Some timers have no local `function` or `data` initialization.

A semantic patch for flexible arrays: 0 case

```
@@
identifier S, member, array;
type T1, T2;
@@

struct S {
   ...
   T1 member;
   T2 array[
-    0
   ];
};
```

`sizeof` now causes a compiler error $\Longrightarrow$ Further effort required.

A semantic patch for flexible arrays: 1 case

```
@@
identifier S, member, array;
type T1, T2;
@@

struct S {
   ...
   T1 member;
*  T2 array[1];
};
```

* highlights a line, for further study.

Assessment

Understandable:

- Small, readable rules transform a large amount of code.

Assessment

Understandable:

- Small, readable rules transform a large amount of code.

Features:

- Metavariables
- Control-flow (…)
- Disjunctions

Assessment

Understandable:

- Small, readable rules transform a large amount of code.

Features:

- Metavariables
- Control-flow (...)
- Disjunctions

Caveat:

- Do our semantic patches do the right thing?

Refining a semantic patch: Working backwards from results

array_size: First attempt

```
@@
expression E1, E2;
@@

  vzalloc(
-   E1 * E2
+   array_size(E1, E2)
  )
```

Updates 102 calls across 67 files.

Problem: Extra parentheses

```
@@ -45,7 +45,7 @@

        memset(&pmlmepriv->assoc_ssid, 0, sizeof(struct ndis_802_11_ssid));

-       pbuf = vzalloc(MAX_BSS_CNT * (sizeof(struct wlan_network)));
+       pbuf = vzalloc(array_size(MAX_BSS_CNT, (sizeof(struct wlan_network))));

        if (pbuf == NULL) {
                res = _FAIL;
```

Problem: Extra parentheses

```
@@ -45,7 +45,7 @@

        memset(&pmlmepriv->assoc_ssid, 0, sizeof(struct ndis_802_11_ssid));

-       pbuf = vzalloc(MAX_BSS_CNT * (sizeof(struct wlan_network)));
+       pbuf = vzalloc(array_size(MAX_BSS_CNT, (sizeof(struct wlan_network))));
                                                ↑                              ↑
        if (pbuf == NULL) {
                res = _FAIL;
```

Function arguments don’t need parentheses.

Dropping any extra parentheses

```
@@
expression E1, E2;
@@

  vzalloc(
-   (E1) * (E2)
+   array_size(E1, E2)
  )
```

Dropping any extra parentheses

```
@@
expression E1, E2;
@@

  vzalloc(
-   (E1) * (E2)
+   array_size(E1, E2)
  )
```

How does `(E1) * (E2)` match `MAX_BSS_CNT * (sizeof(struct wlan_network))`?

Isomorphisms

Tedious to write all options, with and without parentheses.

- Parenthesis isomorphism:

```
Expression
@ paren @
expression E;
@@
 (E) => E
```

Isomorphisms

Tedious to write all options, with and without parentheses.

- Parenthesis isomorphism:

```
Expression
@ paren @
expression E;
@@
 (E) => E
```

- Applied during semantic patch preprocessing.

Isomorphisms

Tedious to write all options, with and without parentheses.

- Parenthesis isomorphism:

```
Expression
@ paren @
expression E;
@@
 (E) => E
```

- Applied during semantic patch preprocessing.
- Other isomorphisms:
  - `{ S } => S`
  - `X == NULL => !X`
  - etc.

Problem: Constants

```
@@ -844,10 +844,10 @@
        tpg_init(&dev->tpg, 640, 360);
        if (tpg_alloc(&dev->tpg, MAX_ZOOM * MAX_WIDTH))
                goto free_dev;
-       dev->scaled_line = vzalloc(MAX_ZOOM * MAX_WIDTH);
+       dev->scaled_line = vzalloc(array_size(MAX_ZOOM, MAX_WIDTH));
        if (!dev->scaled_line)
                goto free_dev;
-       dev->blended_line = vzalloc(MAX_ZOOM * MAX_WIDTH);
+       dev->blended_line = vzalloc(array_size(MAX_ZOOM, MAX_WIDTH));
        if (!dev->blended_line)
                goto free_dev;
```

Problem: Constants

```
@@ -844,10 +844,10 @@
        tpg_init(&dev->tpg, 640, 360);
        if (tpg_alloc(&dev->tpg, MAX_ZOOM * MAX_WIDTH))
                goto free_dev;
-       dev->scaled_line = vzalloc(MAX_ZOOM * MAX_WIDTH);
+       dev->scaled_line = vzalloc(array_size(MAX_ZOOM, MAX_WIDTH));
        if (!dev->scaled_line)
                goto free_dev;
-       dev->blended_line = vzalloc(MAX_ZOOM * MAX_WIDTH);
+       dev->blended_line = vzalloc(array_size(MAX_ZOOM, MAX_WIDTH));
        if (!dev->blended_line)
                goto free_dev;
```

Multiplication can be resolved by the compiler.

Disjunctions

Multiplication of constants $\Rightarrow$ don't use `array_size`.

Multiplication of a non constant $\Rightarrow$ do use `array_size`.

Disjunctions

Multiplication of constants ⇒ don't use `array_size`.

Multiplication of a non constant ⇒ do use `array_size`.

```
@@
expression E1, E2;
constant C1, C2;
@@

(
  vzalloc(C1 * C2)
|
  vzalloc(
-   (E1) * (E2)
+   array_size(E1, E2)
  )
)
```

Problem: Three dimensions

```
@@ -369,11 +369,11 @@
        struct page **pages;
        dma_addr_t *addrs;

-       pages = vzalloc(dd->cfgctxts * dd->rcvtidcnt * sizeof(struct page *));
+       pages = vzalloc(array_size(dd->cfgctxts * dd->rcvtidcnt, sizeof(struct page *)));
        if (!pages)                                  ↑
                goto bail;

-       addrs = vzalloc(dd->cfgctxts * dd->rcvtidcnt * sizeof(dma_addr_t));
+       addrs = vzalloc(array_size(dd->cfgctxts * dd->rcvtidcnt, sizeof(dma_addr_t)));
        if (!addrs)                                  ↑
                goto bail_free;
```

Problem: Three dimensions

```
@@ -369,11 +369,11 @@
        struct page **pages;
        dma_addr_t *addrs;

-       pages = vzalloc(dd->cfgctxts * dd->rcvtidcnt * sizeof(struct page *));
+       pages = vzalloc(array_size(dd->cfgctxts * dd->rcvtidcnt, sizeof(struct page *)));
        if (!pages)                                  ↑
                goto bail;

-       addrs = vzalloc(dd->cfgctxts * dd->rcvtidcnt * sizeof(dma_addr_t));
+       addrs = vzalloc(array_size(dd->cfgctxts * dd->rcvtidcnt, sizeof(dma_addr_t)));
        if (!addrs)                                  ↑
                goto bail_free;
```

`array3_size` checks both multiplications.

Problem: Multiplication of 2 arguments matches multiplication of 3 arguments.

Rule ordering

A semantic patch can contain multiple rules.

- The first rule is applied to each top-level element (function, declaration, ...).
- The next rule is applied to each top-level element after application of the first rule.
- etc.

Rule ordering

```
@@
expression E1, E2, E3;
constant C1, C2, C3;
@@
(
  vzalloc(C1 * C2 * C3)
|
  vzalloc(
-   (E1) * (E2) * (E3)
+   array3_size(E1, E2, E3)
  )
)

@@
expression E1, E2;
constant C1, C2;
@@
(
  vzalloc(C1 * C2)
|
  vzalloc(
-   (E1) * (E2)
+   array_size(E1, E2)
  )
)
```

Some other issues

Argument ordering:

- It can be preferable to put `sizeof` as the last argument.
  `array_size(num_elements, element_size)`

Multiplication by 1:

- Some types have size 1, so the multiplication can just be dropped.

The complete semantic patch

```
@@ expression COUNT; typedef u8, __u8;
   type t = {u8,__u8,char,unsigned char}; @@
-  vzalloc((sizeof(t)) * (COUNT))
+  vzalloc(COUNT)

@@ expression COUNT; size_t e1, e2, e3; @@
(
-  vzalloc((e1) * (e2) * (e3))
+  vzalloc(array3_size(e1, e2, e3))
|
-  vzalloc((e1) * (e2) * (COUNT))
+  vzalloc(array3_size(COUNT, e1, e2))
)

@@ expression STRIDE, COUNT; size_t e; @@
-   vzalloc((e) * (COUNT) * (STRIDE))
+   vzalloc(array3_size(COUNT, STRIDE, e))

@@ expression E1, E2, E3; constant C1, C2, C3; @@
(
   vzalloc(C1 * C2 * C3)
|
-  vzalloc((E1) * (E2) * (E3))
+  vzalloc(array3_size(E1, E2, E3))
)
```

```
@@ size_t e1,e2; expression COUNT; @@
(
-  vzalloc((e1) * (e2))
+  vzalloc(array_size(e1, e2))
|
-  vzalloc((e1) * (COUNT))
+  vzalloc(array_size(COUNT, e1))
)

@@ expression E1, E2; constant C1, C2; @@
(
   vzalloc(C1 * C2)
|
-  vzalloc((E1) * (E2))
+  vzalloc(array_size(E1, E2))
)
```

Some other features

- Metavariable inheritance between rules.
- Interface to Python and OCaml scripting.
- Matching using regular expressions.
- Iteration.

Assessment

Understandable:

- Small, readable rules transform a large amount of code.

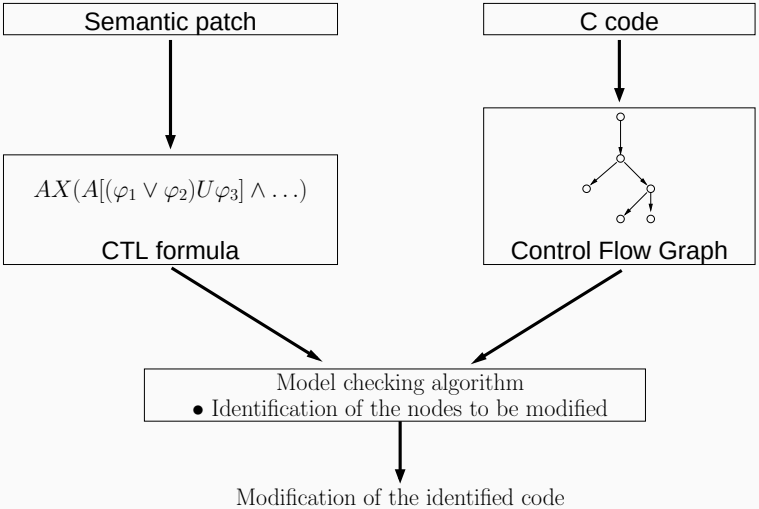
Reliable:

- No risk of confusion, with `bfad_init_timer`, `nes_nic_init_timer`, etc.
- Checks all files, for all architectures.

Iterative development:

- Write rule.
- Run Coccinelle.
- Check results.
- Improve rule for omissions and incorrect transformations.

How does it work?



Processing of C code

Goal: Support processing real Linux source code.

Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

No preprocessing.

- Code manipulated in terms of what the developer sees in the code base.
- Avoids the need for most header files.

Processing of C code

Goal: Support processing real Linux source code.

Dedicated C parser, keeping space and comment information.

No preprocessing.

- Code manipulated in terms of what the developer sees in the code base.
- Avoids the need for most header files.

Intraprocedural CFG.

Processing of Semantic patches

Goal: Allow specifying changes at all code levels.

- Concise and readable.

Processing of Semantic patches

Goal: Allow specifying changes at all code levels.

- Concise and readable.

Support most of C, with few meta-level extensions

- ..., disjunctions, etc.

Processing of Semantic patches

Goal: Allow specifying changes at all code levels.

- Concise and readable.

Support most of C, with few meta-level extensions

- ..., disjunctions, etc.

Isomorphisms, to reduce semantic patch size

- `(E) => E`

Processing of Semantic patches

Goal: Allow specifying changes at all code levels.

- Concise and readable.

Support most of C, with few meta-level extensions

- ..., disjunctions, etc.

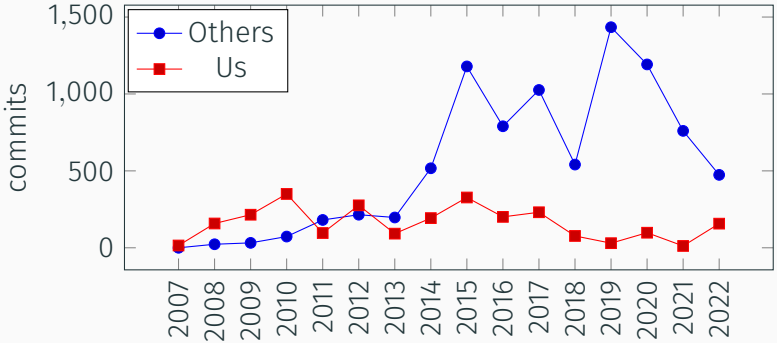
Isomorphisms, to reduce semantic patch size

- `(E) => E`

Implementation via translation to CTL

- Allows $\forall$ and $\exists$ quantification over paths.
- Model checking instrumented with witnesses to collect information about where and how to transform.

Impact: Patches in the Linux kernel



Over 9000 reported uses in all.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 9000 commits in the Linux kernel based on Coccinelle.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 9000 commits in the Linux kernel based on Coccinelle.
- Related work:
  - Inference of semantic patches from examples (spinfer).
  - Semantic patches for searching in commit histories (prequel).
  - Coccinelle for C++ and Rust.

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 9000 commits in the Linux kernel based on Coccinelle.
- Related work:
  - Inference of semantic patches from examples (spinfer).
  - Semantic patches for searching in commit histories (prequel).
  - Coccinelle for C++ and Rust.
- Probably, everyone attending this talk uses some Coccinelle modified code!

Conclusion

- Coccinelle: brings automatic matching and transformation to the systems software developer.
  - Enables needed evolution, independent of the amount of affected code.
- Success: Over 9000 commits in the Linux kernel based on Coccinelle.
- Related work:
  - Inference of semantic patches from examples (spinfer).
  - Semantic patches for searching in commit histories (prequel).
  - Coccinelle for C++ and Rust.
- Probably, everyone attending this talk uses some Coccinelle modified code!

https://coccinelle.gitlabpages.inria.fr/website