**Kernel Testing Tools:**

**1. Coccinelle** :

**What is Coccinelle ?**

Tool to automatically analyze and write C code. Developed by Julia Lawall, a researcher at INRIA Research Labs in Paris. It was designed to primarily target device drivers.

**What was the primary reason that Coccinelle began to be developed ?**

Device drivers call into driver support libraries. If the APIs in these libraries change, then the call site of this API in all device drivers have to be changed. The changes to APIs may happen very often and its impossible to transform every call site to match the changed API. Coccinelle helps do a mass transformation of such call sites.

**Examples of use cases**

a. A classic example is that of the Linux USB code. This code had to be written and rewritten to support faster devices. Each time there were changes made in the Linux USB Code, all kernel drivers had to be updated. Manually converting them can introduce bugs. Hence coccinelle helps here.

b. Coccinelle is not just helpful for the use case of device drivers but sometimes for the core kernel code too. A new clockevent mode called TICK_STOPPED had to be introduced to help NOHZ_FULL dynamic ticks code. The clock device drivers currently did not return any value. All clock device drivers had to return a value instead of void and their pr_err(), BUG_ON and WARN_Ons had to be replaced by returning a unique error code. Again coccinelle came to the rescue.

**Example of Coccinelle scripts:**

```
Assume the C code that you want to change is:

Example 1:

int proc_info(int x) {                          ======>     int
proc_info(int x, scsi *y) {
     scsi *y;                           ...
     ...                                }
     y = scsi_get();
     if (!y) {...return -1;}
     ...
     scsi_put(y);
}
```

So in this example we want to **move a local variable to a function parameter**, **delete calls to library functions** and **delete error check.**

So one can write a coccinelle script like below to start with:

```
@@
     function proc_info;
     identifier x,y;
@@
     int proc_info(int x,
+               scsi *y)
-    scsi *y
     ...
-    y = scsi_get();
-    if (!y) {...return -1; }
-    scsi_put(y);
```

Notice :
1. The use of meta-variables x and y
2. The use of ... operator to indicate 'any code'
3. The use of modifiers @, + and – just like the format of a patch.

This language is called **SmPL or the Semantic Patch Language** and can be executed by using the below command.

*"spatch *.c < proc_info.spatch"*

**Advantages of SmPL**

1. You can apply it to a group of files
2. It is agnostic to spacing, indentation and comments.
3. It abstracts away the specific details and variations at each code site.
4. You can also start a semantic patch by copy pasting from a regular patch and then generalizing it.

**Simpler examples:**

```
1. a = (int) kmalloc (...)      =>  a = kmalloc(...)
2. a & (a-1)                =>  is_power_of_2(a)
3. memset(X,0, PAGE_SIZE) =>  clear_page(X)
```

**1. Using Coccinelle:**

```
     @@
*(kmalloc") *\(/= \1\(/"
          expression E, type T;
     @@
          E =
-              (T)
               kmalloc(...)
```

**1. Using Regular expressions**

```
perl -pi -e "s/ ?= ?\([^\)]*\)
```

**2. Using Coccinelle**

```
     @@
     expression N;
     @@
-    N & (N-1)
+    is_power_of_2(N)
```

**2. Using Regular expression**

```
grep -e "([^\(\)+] ?\& ?\(\1 ?- ?1\)"
```

**3. Using Coccinelle**

```
  @@
     expression X;
   @@
-    memset(X,0,PAGE_SIZE)
+    clear_page(X)
```

**3. Regular expression**

```
grep -e "memset ?\([^,]+, ?, ?0, ?PAGE_SIZE\)"
```

*The portion between @@ and @@ is the meta-variable declaration and that which comes after it is the code. Both combined form a coccinelle rule.*

**Let us consider Example 1 C Code above again:**

The full semantic patch would be like below:

**@ rule1 @**

```
struct SHT fops;
identifier proc_info;
@@
fops.proc_info = proc_info;
```

End of rule 1: Note how the *identifiers are in italics*. This means they can be named anything in the drivers. But they should all have a member called proc_info ( *Notice how it is not italicized* )

**@ rule2 @**

```
identifier rule1.proc_info;
identifier buffer, start, inout, hostno;
identifier hostptr;
@@
     proc_info (
+          struct Scsi_Host *hostptr,
           char *buffer, char **start,
-          int hostno,
           int inout) {
     ...
-               struct Scsi_Host *hostptr;
                ...
-               hostptr = scsi_host_hn_get(hostno);
                ...
```

```
?-              if (!hostptr) {...return...;}
                ...
?-              scsi_host_put(hostptr);
     }
```

**@ rule3 @**

```
identifier rule1.proc_info;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
     proc_info(...) {
     <...
-    hostno
+    hostptr -> host_no
     ...>
}
```

**@ rule4 @**

```
identifier rule1.proc.info;
identifier func;
expression buffer, start, inout, hostno;
identifer hostptr;
@@
     func(..., struct Scsi_Host *hostptr, ...) {
<...
     proc_info(
+         hostptr,
          buffer, start,
-         hostno,
     inout)
...>
}
```

**Note:**

1. All occurrences of the rules are applied. It is a global replacement. "..." means for all subsequent paths. One '-' can erase multiple lines.

2. **Isomorphisms**. Consider rule1. It can do the patching for both the below expressions if you specify isomorphic rules in addition to rule1.

a. myops->proc_info = scsiglue_info;
b. struct SHT wd7000 = {
.proc_info = wd7000_proc_info,
.open = wd7000_proc_info,
}

```
@@
type T;
T E, *E1;
identifier f1d;
@@
E.f1d <=> E1->f1d
```

This rule will patch the first example (a) also with the semantic patching

and

```
@@
type T;TE;
identifier v, f1d;
expression E1;
@@
E.f1d = e1; => T v = {.f1d = E1};
```

This rule will take care of the second example b.

In Summary, the vision behind coccinelle is to help **collateral evolution.**

**Coccinelle can also be used for bug fixes**.

The goal is to enable programmers who become aware of a potential pattern of bugs and use this pattern to search for other instances of the bug.

```
--- a/drivers/net/smc911x.c
+++ b/drivers/net/smc911x.c
@@ -1299,9 +1299,9 @@
PRINT_PKT(skb->data, skb->len);
dev->last_rx = jiffies;
skb->protocol = eth_type_trans(skb, dev);
- netif_rx(skb);
dev->stats.rx_packets++;
dev->stats.rx_bytes += skb->len;
+ netif_rx(skb);
spin_lock_irqsave(&lp->lock, flags);
pkts = (SMC_GET_RX_FIFO_INF()&RX_FIFO_INF_RXSUSED_)>>16
```

In the above example, its possible that skb can be freed after call to netif_rx. Hence it is important to move the call to netif_rx to the position specified above.

**A semantic patch which detects for similar call sites can be written.**

```
@r exists@
expression skb,e,e1;
position p;
identifier fld;
@@

(
netif_rx(skb)
|
netif_rx_ni(skb)
)
...when != skb = e
skb@p->fld

@ script python @
skb << r.skb; p << r.p
@@
print "matching expr: %s, position: %d" % (skb, p)
```

The above is the extended semantic patch which matches more functions having semantics similar to netif_rx() like netif_rx_ni(skb)

**For more information on bug finding tool using coccinelle:**
http://lig-membres.imag.fr/palix/papers/dsn09-lawall.pdf

**Other test projects running in the community:**

**1. kselftest: tools/testing/selftests in the kernel source tree**

The purpose is to provide developers and end users with a quick way of running tests against the kernel. These tests are not full scale stress tests but are basic sanity tests. The goal is to be able to run all tests well within 15-20 minutes.

`git://git.kernel.org/pub/scm/shuah/linux-kselftest`

**Build**
`make -C tools/testing/selftest/s run_tests`

**Run**
`make -C tools/testing/selftests run_tests`

**Run a specific test**
`make -C tools/testing/selftests TARGETS=cpu-hotplug run_tests`

**Other testing efforts:**

1. lkp https://git.kernel.org/cgit/linux/kernel/git/wfg/lkp-tests.git

2. Dave Jones **Trinity Fuzz tester** which tests system call APIs:
Git tree: https://github.com/kernelslacker/trinity

a. There were efforts in developing fuzz testing earlier but they were not sophisticated enough.
b. The kernel system call interfaces expect certain kinds of input. We cannot pass arbitrary inputs. It is bound to fail. Nor do they help us find unexpected errors. Hence Trinity is an effort at sophisticated fuzz testing.

**Trinity** has aided not just in system call testing, but also in **bug finding** in different parts of the kernel.

`./configure.sh and make`

`./trinity -c madvise`

`./trinity -V /bin -c execve` to test user space programs.

This command was demoed to have executed random programs in /bin and the system log revealed a lot of segmentation faults with a large number of programs crashing.

**Trinity supports** Alpha, Aarch64, ARM, i386, IA-64, MIPS, PowerPC-32, **PowerPC-64**, S390, S390x, SPARC-64, x86-64.