

# How do we make KUnit work for us?

And should we ever... work for KUnit?

Tiffany Yang <[ynaffit@google.com](mailto:ynaffit@google.com)>

# On unit testing

## test\_linear\_ranges.c

- [Link to patch](#)

Matti Vaittinen

```
/* First things first. I deeply dislike unit-tests. I have seen all the hell
* breaking loose when people who think the unit tests are "the silver bullet"
* to kill bugs get to decide how a company should implement testing strategy...
*
* Believe me, it may get _really_ ridiculous. It is tempting to think that
* walking through all the possible execution branches will nail down 100% of
* bugs. This may lead to ideas about demands to get certain % of "test
* coverage" - measured as line coverage. And that is one of the worst things
* you can do.
*
* Ask people to provide line coverage and they do. I've seen clever tools
* which generate test cases to test the existing functions - and by default
* these tools expect code to be correct and just generate checks which are
* passing when ran against current code-base. Run this generator and you'll get
* tests that do not test code is correct but just verify nothing changes.
* Problem is that testing working code is pointless. And if it is not
* working, your test must not assume it is working. You won't catch any bugs
* by such tests. What you can do is to generate a huge amount of tests.
* Especially if you were asked to provide 100% line-coverage x_x. So what
* does these tests - which are not finding any bugs now - do?
*
* They add inertia to every future development. I think it was Terry Pratchett
* who wrote someone having same impact as thick syrup has to chronometre.
* Excessive amount of unit-tests have this effect to development. If you do
* actually find _any_ bug from code in such environment and try fixing it...
* ...chances are you also need to fix the test cases. In sunny day you fix one
* test. But I've done refactoring which resulted 500+ broken tests (which had
* really zero value other than proving to managers that we do do "quality")...
*
* After this being said - there are situations where UTs can be handy. If you
* have algorithms which take some input and should produce output - then you
* can implement few, carefully selected simple UT-cases which test this. I've
* previously used this for example for netlink and device-tree data parsing
* functions. Feed some data examples to functions and verify the output is as
* expected. I am not covering all the cases but I will see the logic should be
* working.
*
* Here we also do some minor testing. I don't want to go through all branches
* or test more or less obvious things - but I want to see the main logic is
* working. And I definitely don't want to add 500+ test cases that break when
* some simple fix is done x_x. So - let's only add few, well selected tests
* which ensure as much logic is good as possible.
*/
```



**Excessive amount of unit-tests... add inertia to every future development.**  
**If you do actually find `_any_` bug from code in such environment and try fixing it...chances are you also need to fix the test cases.**

Matti Vaittinen

“ There are situations where UTs can be handy. I don't want to... test more or less obvious things - but I want to see the main logic is working. So - let's only add few, well selected tests which ensure as much logic is good as possible.

Matti Vaittinen

# Unit Testing

## Strengths

How unit tests can shorten our development cycle:

- **Reproducibility:** Tests are written to be deterministic.
- **Isolation:** KUnit can check the output of kernel functions directly.
- **Maintainability:** Unit tests serve as a form of documentation. They clearly lay out the expected output of a function and how it's used. This allows more confident refactoring.
- **Ease of use:** Once the test has a structure, adding new cases is fairly painless (even more so with GenAI!)

All together, these qualities can allow us to quickly debug some specific error cases instead of having to contrive userland behavior to trip the necessary failure preconditions deep in the kernel.

## Weaknesses

Limitations of unit tests compared to our other forms of testing:

- To write a unit test case, we have to already know the failure state.
- Fuzzing/stress tests are better for actually discovering new failure states.
- Unit tests are difficult to write when state is difficult to represent.

# More KUnit Benefits

- The `binder_alloc` test, which took over 5 minutes just to run on cuttlefish (excluding boot + build time) takes **~25 seconds** to compile and run on UML (and ~35 seconds to run on qemu).
- Supports testing on a variety of architectures! Especially useful for cases like [Kees' recent ffs regression test](#).
- Additional features: ubsan integration, `kunit_bus` for devices, test-managed resources, user address space attachment, obtaining running test in non-test code, parameterized test cases, exporting functions to test-only, dependency injection, etc.



# Cool Recent Work

Kernel developers are innovating with KUnit!

1

Thomas Weißschuh's new [printk ringbuffer test](#) – it's kind of a stress test!

2

Coiby Xu's use of an inductive proof for the [crash\\_exclude\\_mem\\_range](#) test!

3

Brian Norris & Sinan Nalkaya's addition of [sparse interrupt support to UML](#) for running KUnit tests!

4

Nam Cao's test that use simulated instructions to [detect errors in riscv kprobes](#)!

# Takeaways



Bugs make good, meaningful  
KUnit test cases



We can [add functionality](#) to KUnit  
to support our use cases






**The world is our  
oyster!**

Or is it?

# Challenges



**Dependency Injection  
requires Invasive Code  
Changes**



**Tests Must Run on  
Production Builds**



**Shared Global State**

# Invasive Code changes

## Dependency injection requires annotating source code directly

KUnit offers a [static\\_stub](#) API to redirect function calls to mock functions, but using it requires modifying the function being stubbed out, **not** the function under test.

- Calls that must be stubbed often belong to another subsystem **that we do not own**.
- Stub macros pollute code.

Suppose we want to test the following function:

```
static void my_function_under_test(struct my_struct *my_state)
{
    call_to_other_subsystem(my_state->other_subsys_struct);
    ...
}
```

To isolate our code under test, we may wish to replace `call_to_other_subsystem` with a test-only mock implementation.

This would require us to go into the source for `other_subsystem` and annotate the first line of the function:

```
static void call_to_other_subsystem(struct other_subsys_struct
*other_state)
{
    KUNIT_STATIC_STUB_REDIRECT(call_to_another_subsystem, other_state);

    /* actual implementation */
}
```



# No Kernel Test Builds

## Tests run on Production Kernel Builds

Android does not make any test-only kernel builds. Our KUnit tests are built and loaded as modules.

- Any instrumentation to implement test code will live in production.
- Changes to allow testing must not impact performance.
- This restricts us from being able to run KUnit tests for boot.

# Shared Global State

## No simple way to mock state for test

At best, live global state makes test runs non-deterministic. At worst, test runs can have unintended side effects on a running kernel.

- Core kernel subsystems are often coupled to others through shared global state.
- This state is difficult to track; it may be buried deep within a call chain.
- Downstream calls may be updated to touch global state at any time.
- Even when the function under test reads global state without altering it, changes in that state may result in unreproducible output.

Suppose we wanted to test `sched_balance_newidle`, even if we managed to call it with an isolated test run queue, the function immediately gets the global `jiffies`, and then uses it in a downstream call.

```
/*
 * sched_balance_newidle is called by schedule() if this_cpu
 * is about to become
 * idle. Attempts to pull tasks from other CPUs.
 *
 * Returns:
 *   < 0 - we released the lock and there are !fair tasks
 *   present
 *   0 - failed, no new tasks
 *   > 0 - success, new (fair) tasks present
 */
static int sched_balance_newidle(struct rq *this_rq, struct
rq_flags *rf)
{
    unsigned long next_balance = jiffies + HZ;
    int this_cpu = this_rq->cpu;
    ...
    if (!get_rd_overloaded(this_rq->rd) ||
        (sd && this_rq->avg_idle <
sd->max_newidle_lb_cost)) {

        if (sd)
            update_next_balance(sd, &next_balance);
        rcu_read_unlock();
    }
}
```

# How can we address these challenges for KUnit?



Accessing global state through local pointers



Refactoring code to make it more modular



A much better solution from someone in the audience!!

# Is KUnit even the right solution?

**What if we built some library from kernel source in tree?**

## Related Work

- [clknetstim](#): Clock and network simulator built from source
- KFuzzTest ([v2](#)): In-tree library for creating kernel fuzz targets

**What if we refactor for easier testing out of tree?**

## Related Work

- Last year's core VMA refactoring ([v3](#) patch series)

**What qualities would a better solution have?**

The image features four green curved shapes, resembling quarter-circles or stylized corners, positioned in the top-left, top-right, bottom-left, and bottom-right corners of the slide. The central text is a question about adopting KUnit in Android development.

**What's the biggest barrier  
to adopting KUnit in your  
test dev?**



Get in touch

[ynaffit@google.com](mailto:ynaffit@google.com)



# Thank you

