

Towards Verification of Linux Kernel Code

Julia Lawall, Keisuke Nishimura, Jean-Pierre Lozi

Inria

June 30, 2024

- Seems reliable...

The Linux kernel

- Seems reliable... But is actually full of bugs.

- Seems reliable... *But is actually full of bugs.*
- Some fit well-known patterns:
 - missing free, use after free, dereference of NULL, missing unlock, misplaced memory barrier, etc.
 - Existing tools handle these issues more or less well (Smatch, Coccinelle, Coverity, etc.)

The Linux kernel

- Seems reliable... *But is actually full of bugs.*
- Some fit well-known patterns:
 - missing free, use after free, dereference of NULL, missing unlock, misplaced memory barrier, etc.
 - Existing tools handle these issues more or less well (Smatch, Coccinelle, Coverity, etc.)
- Some depend on an algorithm, and are completely context specific:

The Linux kernel

- Seems reliable... *But is actually full of bugs.*
- Some fit well-known patterns:
 - missing free, use after free, dereference of NULL, missing unlock, misplaced memory barrier, etc.
 - Existing tools handle these issues more or less well (Smatch, Coccinelle, Coverity, etc.)
- Some depend on an algorithm, and are completely context specific:
 - *Maybe the Linux kernel is a candidate for verification?*

A challenge

Verification is really hard...

- Some efforts have succeeded: SeL4, CertiKOS, etc.
- Tools are getting better?

A challenge

Verification is really hard...

- Some efforts have succeeded: SeL4, CertiKOS, etc.
- Tools are getting better?

A further wrinkle: The Linux kernel is updated regularly...

- Sometimes with new algorithms.
- Sometimes to optimize existing code.
- Sometimes to fix bugs.

A challenge

Verification is really hard...

- Some efforts have succeeded: SeL4, CertiKOS, etc.
- Tools are getting better?

A further wrinkle: The Linux kernel is updated regularly...

- Sometimes with new algorithms.
- Sometimes to optimize existing code.
- Sometimes to fix bugs.
- Sometimes introducing bugs. :(

- For optimizations, the overall input-output behavior should not change.

- For optimizations, the overall input-output behavior should not change.
- Maybe we could define pre and post conditions for one version and reuse them on new versions?

- Prove the correctness of Linux kernel code, focusing on core functionalities.

Goals

- Prove the correctness of Linux kernel code, focusing on core functionalities.
- Port specifications and proofs from one Linux kernel release to the next.

Goals

- Prove the correctness of Linux kernel code, focusing on core functionalities.
- Port specifications and proofs from one Linux kernel release to the next.
- Achieve uptake from the Linux kernel community?

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.
 - Generate the necessary verification conditions (Hoare logic).
 - The use of SMT solvers should provide some resistance to code changes.

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.
 - Generate the necessary verification conditions (Hoare logic).
 - The use of SMT solvers should provide some resistance to code changes.
- Focus on the algorithm.

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.
 - Generate the necessary verification conditions (Hoare logic).
 - The use of SMT solvers should provide some resistance to code changes.
- Focus on the algorithm.

What we don't do:

- No consideration of concurrency.
- No consideration of **hidden** memory issues (aliasing, null pointers, use after free, etc.).

What we do:

- Use the original C source code for the function of interest.
- Write dummy definitions in C for external functions, as needed.
- Use Frama-C to manage the proving process.
 - Generate the necessary verification conditions (Hoare logic).
 - The use of SMT solvers should provide some resistance to code changes.
- Focus on the algorithm.

What we don't do:

- No consideration of concurrency.
- No consideration of `hidden` memory issues (aliasing, null pointers, use after free, etc.).
- These are hard issues, but developers can make mistakes without them.

A case study: should_we_balance

Goal:

- Should a core should try to steal tasks during load balancing?

Starting point:

- Patch first proposed in August 2013.
- Extracted from scattered existing code.
- First patch was buggy.
- First released in Linux v3.12.

Subsequent history:

- 10 variants over time (+1 proposed by Keisuke).
- Several recent optimizations.

The original definition

```
static int should_we_balance(struct lb_env *env) {
    struct sched_group *sg = env->sd->groups;
    struct cpumask *sg_cpus, *sg_mask;
    int cpu, balance_cpu = -1;

    if (env->idle == CPU_NEWLY_IDLE)
        return 1;
    sg_cpus = sched_group_cpus(sg);
    sg_mask = sched_group_mask(sg);
    for_each_cpu_and(cpu, sg_cpus, env->cpus) {
        if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
            continue;
        balance_cpu = cpu;
        break;
    }
    if (balance_cpu == -1)
        balance_cpu = group_balance_cpu(sg);
    return balance_cpu != env->dst_cpu; // != should be ==
}
```

Key properties

For a given environment *env*,

- **Uniqueness** If two non-newly idle cores call *should_we_balance*, then at most one of them should get a positive result.
- **Existence** *should_we_balance* should return *true* for some core on the machine.

Key properties

For a given environment *env*,

- **Uniqueness** If two non-newly idle cores call *should_we_balance*, then at most one of them should get a positive result.
- **Existence** *should_we_balance* should return *true* for some core on the machine.

What to prove?

- Frama-C proves postconditions from preconditions.
 - Describes function semantics in terms of the input-output behavior.
- Our key properties are somewhat different, but start with that.

Initial version (verification expert): pre and post conditions

```
/*@  
... // data validity, no side effects  
  
behavior newly_idle:  
  assumes env->idle == CPU_NEWLY_IDLE;  
  ensures \result;  
  
behavior not_newly_idle1:  
  assumes env->idle != CPU_NEWLY_IDLE;  
  assumes \exists integer i; relevant(i, env) && idle_cpu(i);  
  ensures \forall integer i;  
    relevant(i, env) ==> idle_cpu(i) ==>  
      (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>  
      (\result <==> env->dst_cpu != i);  
  
behavior not_newly_idle2:  
  assumes env->idle != CPU_NEWLY_IDLE;  
  assumes \forall integer i; relevant(i, env) ==> !idle_cpu(i);  
  ensures \result <==> group_balance_cpu(env->sd->groups) != env->dst_cpu;  
  
complete behaviors;  
disjoint behaviors;  
*/
```


Initial version (verification expert): loop invariants

```
static int should_we_balance(struct lb_env *env)
{
    ...
    sg_cpus = sched_group_cpus(sg);
    sg_mask = sched_group_mask(sg);
    /*@
     loop invariant 0 <= cpu <= small_cpumask_bits;
     loop invariant \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
     loop assigns cpu;
     loop variant small_cpumask_bits - cpu;
    */
    for_each_cpu_and(cpu, sg_cpus, env->cpus) {
        if (!cpumask_test_cpu(cpu, sg_mask) || !idle_cpu(cpu))
            continue;

        balance_cpu = cpu;
        break;
    }
    ...
}
```

Change types

#	Commit id	Date	Release	Impact
0	23f0d2093c78	Aug. 2013	–	create the function
1	b0cff9d88ce2	Sep. 2013	v3.12	replace != by ==
2	af218122b103	May 2017	–	eliminate a redundant function call
3	e5c14b1fb892	May 2017	v4.13	rename a function
4	024c9d2faebd	Oct. 2017	v4.14	check validity of the stealing CPU
5	97fb7a0a8944	Mar. 2018	v4.17	improve comments
6	64297f2b03cc	Apr. 2020	v5.8	return early on finding an idle core
7	792b9f65a568	Jun. 2022	v6.0	abort if tasks are detected on a newly idle CPU
8	b1bfeab9b002	Jul. 2023	–	prefer fully idle cores
9	f8858d96061f	Sep. 2023	v6.6	remove non-idle hyperthreads from the CPU mask
10	6d7e4782bcf5	Oct. 2023	v6.8	change a condition of the selection algorithm

Red versions contain bugs.

Question:

As the code changes,
can developers update the specifications accordingly?

Change types and proof impact: No impact

Changes in comments clearly have no impact on the proof.

Change types and proof impact: No impact

Changes in comments clearly have no impact on the proof.

Code changes may also have no impact on the proof.

```
static int should_we_balance(struct lb_env *env)
{
    struct sched_group *sg = env->sd->groups;
-   int cpu, balance_cpu = -1;
+   int cpu;

    ...
    for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
        if (!idle_cpu(cpu))
            continue;
-       balance_cpu = cpu;
-       break;
+       return cpu == env->dst_cpu;
    }
-   if (balance_cpu == -1)
-       balance_cpu = group_balance_cpu(sg);
-   return balance_cpu == env->dst_cpu;
+   return group_balance_cpu(sg) == env->dst_cpu;
}
```

Change types and proof impact: more invasive changes

```
for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
    if (!idle_cpu(cpu))
        continue;
+   if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
+       if (idle_smt == -1)
+           idle_smt = cpu;
+       continue;
+   }
    return cpu == env->dst_cpu;
}
```

- Sensitive to hyperthreads.
- Avoid a core whose hyperthread is occupied, but keep it as a fallback.

Change types and proof impact: more invasive changes

```
for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
    if (!idle_cpu(cpu))
        continue;
+   if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
+       if (idle_smt == -1)
+           idle_smt = cpu;
+       continue;
+   }
    return cpu == env->dst_cpu;
}
```

Specification change:

```
/*@
loop invariant 0 <= cpu <= small_cpumask_bits;
- loop invariant \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
- loop assigns cpu;
+ loop invariant env->sd->flags & SD_SHARE_CPUCAPACITY ==> idle_smt == -1;
+ loop invariant idle_smt == -1 ==> \forall integer j; 0 <= j < cpu ==> relevant(j, env) ==> !idle_cpu(j);
+ loop invariant idle_smt != -1 ==> 0 <= idle_smt < cpu && relevant(idle_smt, env) && idle_cpu(idle_smt);
+ loop invariant idle_smt != -1 ==> \forall integer j; 0 <= j < idle_smt ==> relevant(j, env) ==> !idle_cpu(j);
+ loop invariant idle_smt != -1 ==> \forall integer j; idle_smt <= j < cpu ==> relevant(j, env) ==> !idle_core(j);
+ loop assigns cpu, idle_smt;
loop variant small_cpumask_bits - cpu;
*/
```

Change types and proof impact: invasive changes

```
+   cpumask_copy(swb_cpus, group_balance_mask(sg));
-   for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+   for_each_cpu_and(cpu, swb_cpus, env->cpus) {
        if (!idle_cpu(cpu))
            continue;
        if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
            if (idle_smt == -1)
                idle_smt = cpu;
#ifdef CONFIG_SCHED_SMT
+           cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
#endif
            continue;
        }
        return cpu == env->dst_cpu;
    }
```


Change types and proof impact: invasive changes

```
+   cpumask_copy(swb_cpus, group_balance_mask(sg));
-   for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+   for_each_cpu_and(cpu, swb_cpus, env->cpus) {
        if (!idle_cpu(cpu))
            continue;
        if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
            if (idle_smt == -1)
                idle_smt = cpu;
#ifdef CONFIG_SCHED_SMT
+           cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
#endif
            continue;
        }
        return cpu == env->dst_cpu;
    }
```

- `cpumask_andnot` writes into its first argument.
 - Such side effects impact the loop invariants.

Change types and proof impact: invasive changes

```
+   cpumask_copy(swb_cpus, group_balance_mask(sg));
-   for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+   for_each_cpu_and(cpu, swb_cpus, env->cpus) {
        if (!idle_cpu(cpu))
            continue;
        if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
            if (idle_smt == -1)
                idle_smt = cpu;
#ifdef CONFIG_SCHED_SMT
+           cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
#endif
            continue;
        }
        return cpu == env->dst_cpu;
    }
```

- *cpumask_andnot* writes into its first argument.
 - Such side effects impact the loop invariants.
- The first two arguments to *cpumask_andnot* are aliases.

Change types and proof impact: invasive changes

```
+   cpumask_copy(swb_cpus, group_balance_mask(sg));
-   for_each_cpu_and(cpu, group_balance_mask(sg), env->cpus) {
+   for_each_cpu_and(cpu, swb_cpus, env->cpus) {
        if (!idle_cpu(cpu))
            continue;
        if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
            if (idle_smt == -1)
                idle_smt = cpu;
#ifdef CONFIG_SCHED_SMT
+           cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
#endif
            continue;
        }
        return cpu == env->dst_cpu;
    }
```

- `cpumask_andnot` writes into its first argument.
 - Such side effects impact the loop invariants.
- The first two arguments to `cpumask_andnot` are aliases.

Months of work... One assert needed.

Bugs found and optimization opportunities

An older behavior:

```
behavior not_newly_idle1:
  assumes env->idle != CPU_NEWLY_IDLE;
  assumes env->cpus->bits[env->dst_cpu];
  assumes \exists integer i; relevant(i, env) && idle_cpu(i);
  ensures \forall integer i; relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (\result <==> env->dst_cpu == i);
```

A newer behavior: (bug introduced)

```
behavior not_newly_idle1b:
  assumes env->idle != CPU_NEWLY_IDLE;
  assumes env->cpus->bits[env->dst_cpu];
  assumes !(env->sd->flags & SD_SHARE_CPUCAPACITY);
  assumes \forall integer i; relevant(i, env) ==> !idle_core(i);
  assumes \exists integer i; relevant(i, env) && idle_cpu(i);
  ensures \forall integer i; relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (\result ==> (env->dst_cpu == i || env->dst_cpu == group_balance_cpu(env->sd->groups)));
  ensures \forall integer i; relevant(i, env) ==> idle_cpu(i) ==>
    (\forall integer j; 0 <= j < i ==> relevant(j, env) ==> !idle_cpu(j)) ==>
    (env->dst_cpu == i ==> \result);
```

Bugs found and optimization opportunities

Optimization opportunity: (ifdefs elided)

```
cpumask_copy(swb_cpus, group_balance_mask(sg));
for_each_cpu_and(cpu, swb_cpus, env->cpus) {
    if (!idle_cpu(cpu)) {
+       if (idle_smt != -1)
+           cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
        continue;
    }
    if (!(env->sd->flags & SD_SHARE_CPUCAPACITY) && !is_core_idle(cpu)) {
        if (idle_smt == -1)
            idle_smt = cpu;
        cpumask_andnot(swb_cpus, swb_cpus, cpu_smt_mask(cpu));
        continue;
    }
    return cpu == env->dst_cpu;
}
```

No changes needed to the specifications!

Work estimate:

- Maybe 1.5 months for versions 0 - 8. ✓
- 3.5 months for version 9 (*cpumask_andnot*) ✗
 - Resolved some misunderstandings about Frama-C. ✓
- No work for proving correct the fix of the bug in v8 and v9. ✓

Future work

- Prove the Uniqueness and Existence properties.

Future work

- Prove the Uniqueness and Existence properties.
- Prove the correctness of more code in the Linux kernel.

Future work

- Prove the Uniqueness and Existence properties.
- Prove the correctness of more code in the Linux kernel.
 - How to recognize bugs in the code based on these proofs?
 - How to recognize optimization opportunities based on these proofs?

Future work

- Prove the Uniqueness and Existence properties.
- Prove the correctness of more code in the Linux kernel.
 - How to recognize bugs in the code based on these proofs?
 - How to recognize optimization opportunities based on these proofs?
- Port specifications and proofs from one Linux kernel release to the next.

Future work

- Prove the Uniqueness and Existence properties.
- Prove the correctness of more code in the Linux kernel.
 - How to recognize bugs in the code based on these proofs?
 - How to recognize optimization opportunities based on these proofs?
- Port specifications and proofs from one Linux kernel release to the next.
 - What changes are needed in the specifications?
 - How to automate them?
 - How to recognize cases that can't be automated (i.e., new algorithms)?

Future work

- Prove the Uniqueness and Existence properties.
- Prove the correctness of more code in the Linux kernel.
 - How to recognize bugs in the code based on these proofs?
 - How to recognize optimization opportunities based on these proofs?
- Port specifications and proofs from one Linux kernel release to the next.
 - What changes are needed in the specifications?
 - How to automate them?
 - How to recognize cases that can't be automated (i.e., new algorithms)?
- Uptake from the Linux kernel community?