Testing the Kernel against Verified Oracles

Based on Mete’s bachelor’s thesis: Testing the Red Black Trees of the Linux Kernel against a Formally Verified Variant

About us

Mete Polat <metepolat2000@gmail.com>

(very soon graduated) undergraduate at the Technical University of Munich (TUM)

Interested in the development of high-assurance software stacks using formal verification

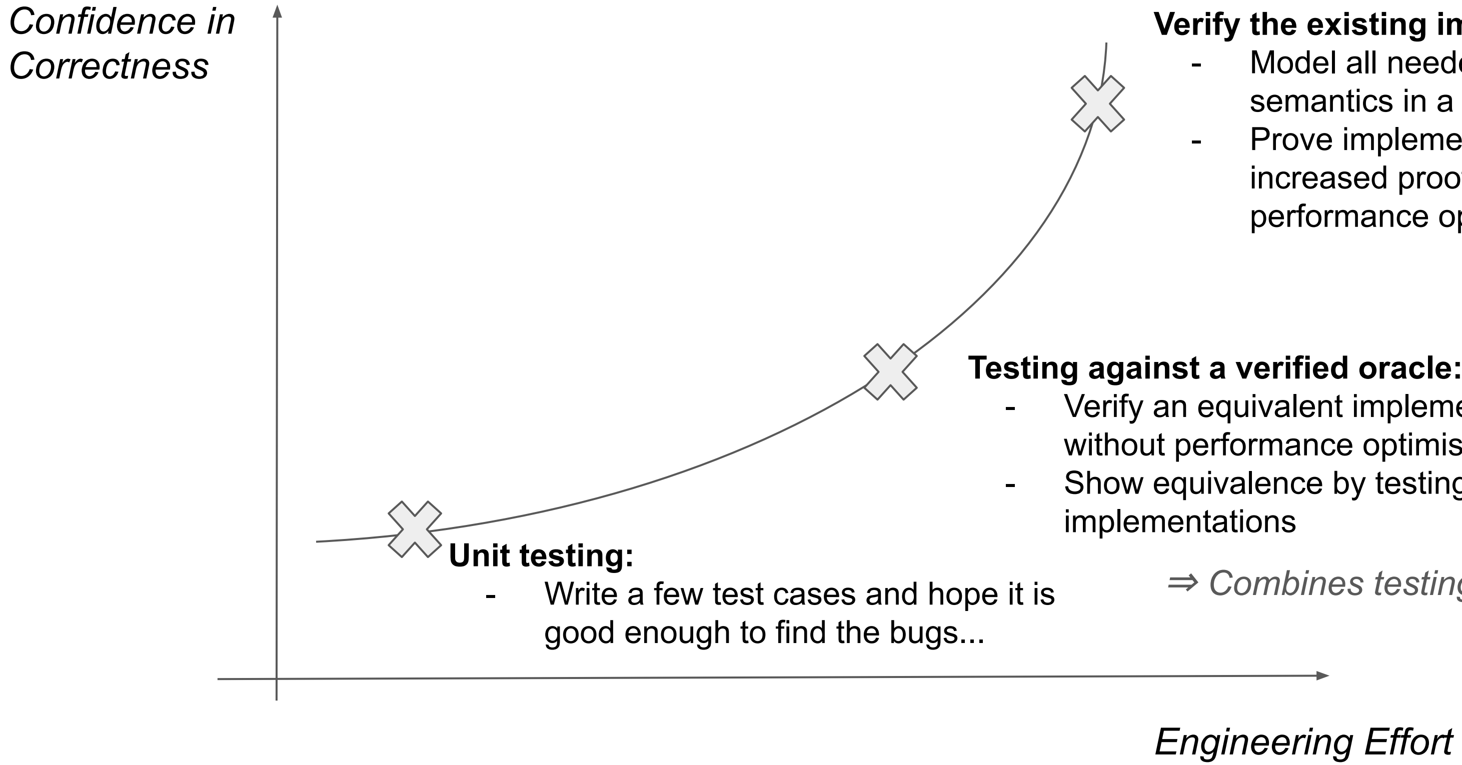
Lukas Bulwahn <lukas.bulwahn@gmail.com>

PhD in formal methods, Contributor to the theorem prover Isabelle during PhD

Chief Expert at Elektrobit:

- Interested in safety argumentations for Linux-based systems, active in ELISA community
- Kernel janitor

Motivation for using verified oracles



Verify the existing implementation:
- Model all needed details of the C semantics in a theorem prover
- Prove implementation correct with increased proof complexity due to performance optimisations.

Testing against a verified oracle:
- Verify an equivalent implementation without performance optimisations
- Show equivalence by testing both implementations

⇒ Combines testing and verification world

Unit testing:
- Write a few test cases and hope it is good enough to find the bugs...

Our concrete example

We use Isabelle [1].

> Isabelle = proof assistant (for proving mathematical theorems and software) for formal verification (a human creates machine-checked correctness proof)

Isabelle developers verified a Red-Black Tree implementation in Isabelle. (Our verified test oracle)

Kernel developers made a Red-Black Tree implementation in the kernel. (Our implementation under test)

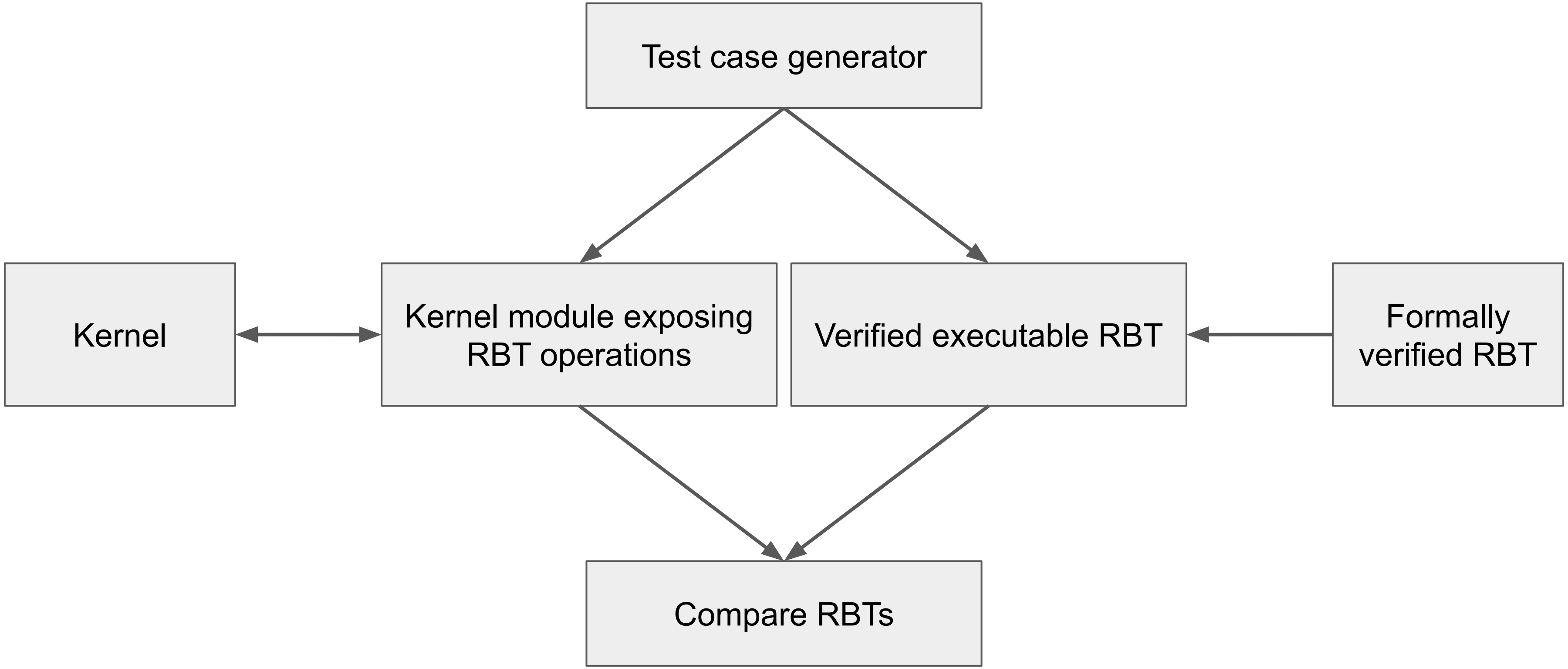
We test the equivalence of those two implementations *extensively*.

[1] https://isabelle.in.tum.de

Another example, see: https://www21.in.tum.de/students/verified_testing/index.html

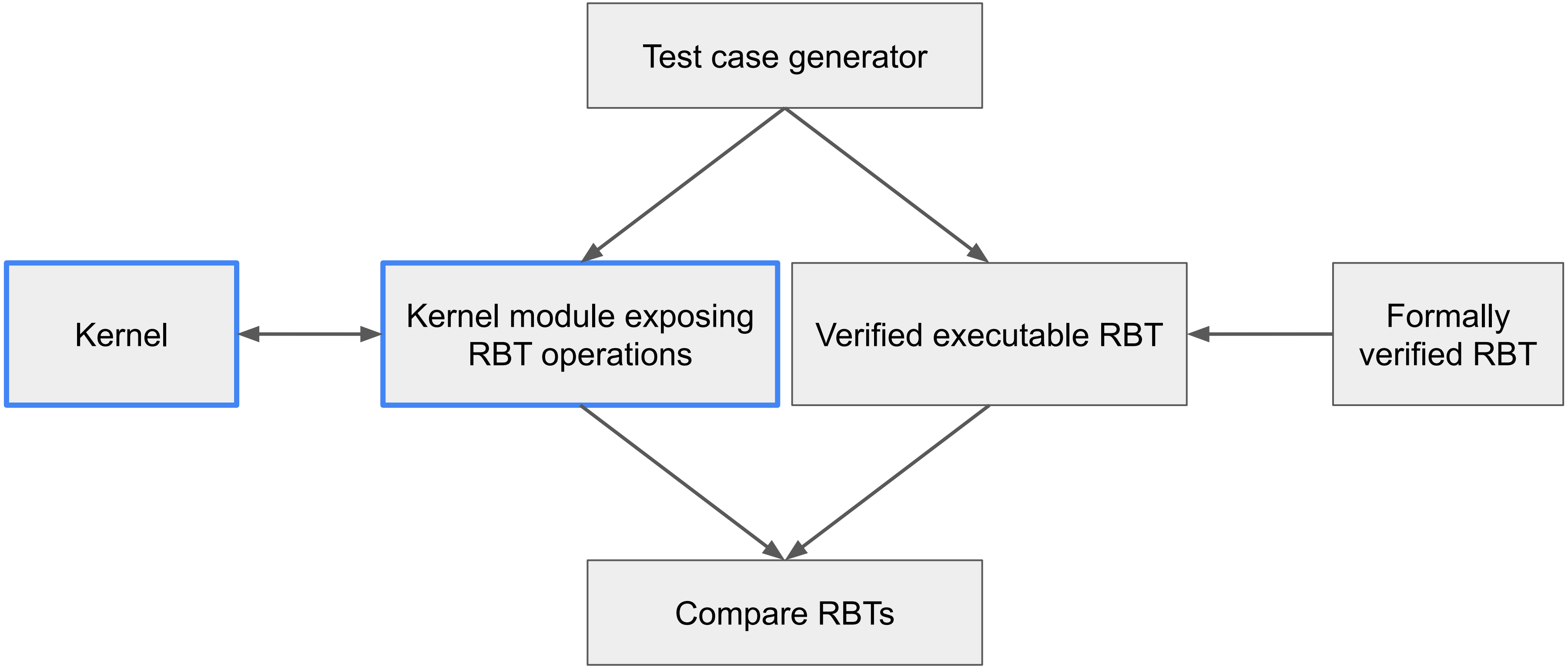
source code: https://github.com/metp/kernel-vs-verified-rbt

RBT testing pipeline



source code: https://github.com/metp/kernel-vs-verified-rbt

RBT testing pipeline

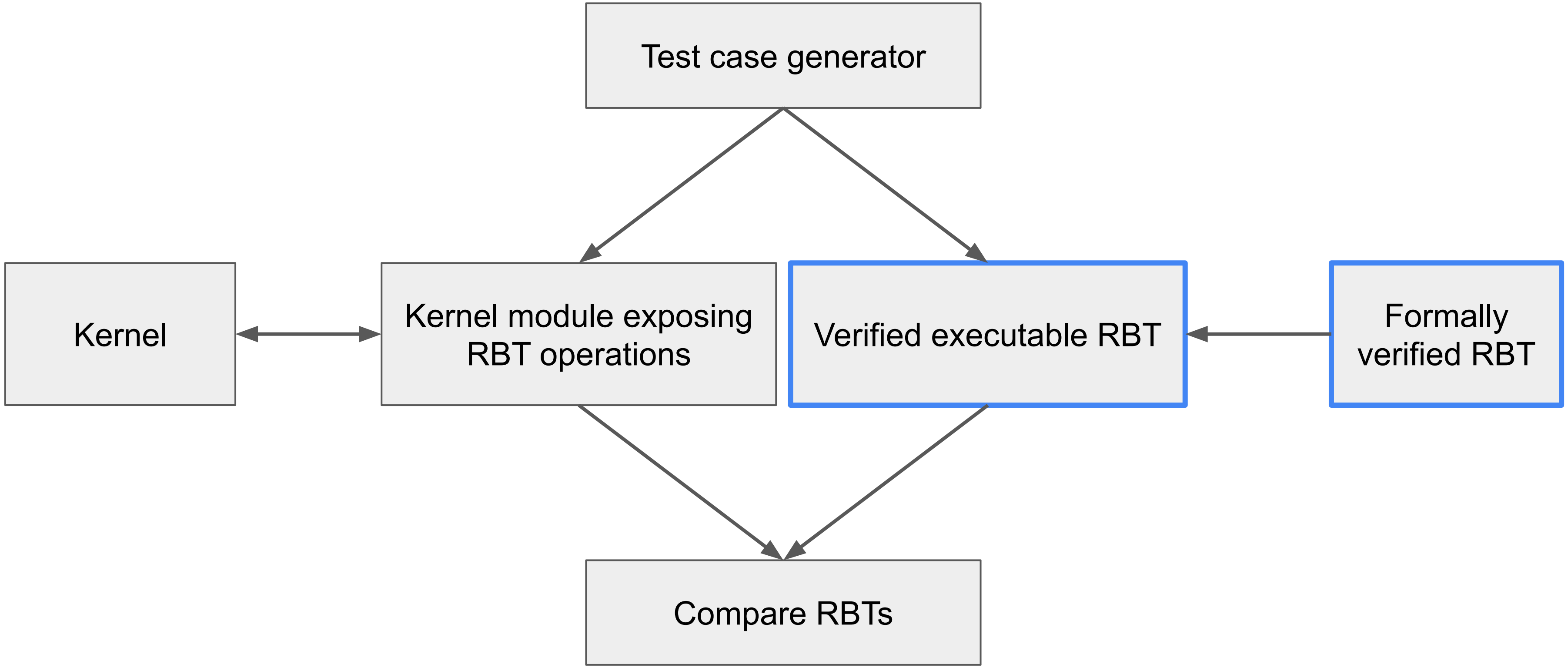


/sys/kernel/debug/rbt_if/

- cmd
  - Reading prints tree
  - Write 0 resets tree
  - Write 1 inserts key
  - Write 2 deletes key
- key

source code: https://github.com/metp/kernel-vs-verified-rbt

RBT testing pipeline



How to formally verify (functional) data structures?

Basic idea: Use the same methodologies as mathematicians use

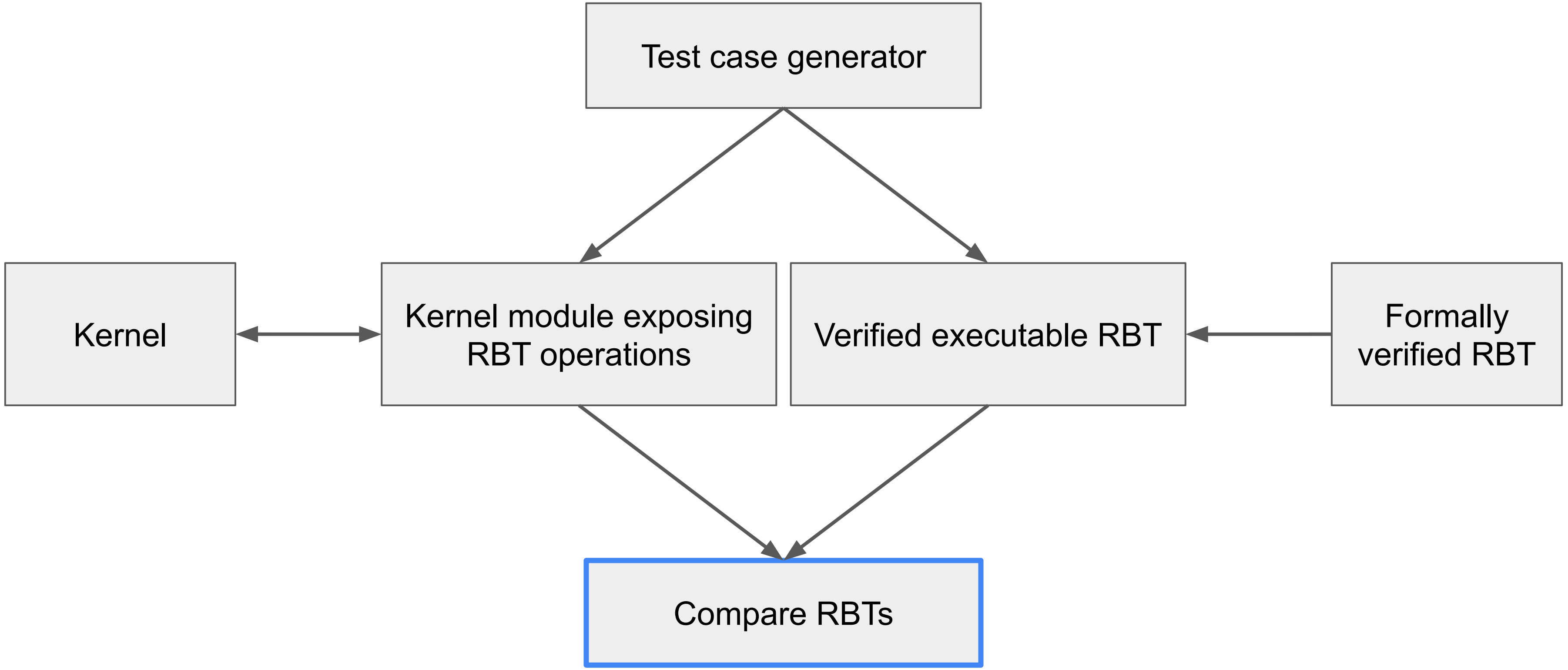
If you are really interested: https://isabelle.in.tum.de/library/HOL/HOL-Data_Structures/RBT_Set.html

```
fun baliL :: "'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt" where
"baliL (R (R t1 a t2) b t3) c t4 = R (B t1 a t2) b (B t3 c t4)" |
"baliL (R t1 a (R t2 b t3)) c t4 = R (B t1 a t2) b (B t3 c t4)" |
"baliL t1 a t2 = B t1 a t2"

fun baliR :: "'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt" where
"baliR t1 a (R t2 b (R t3 c t4)) = R (B t1 a t2) b (B t3 c t4)" |
"baliR t1 a (R (R t2 b t3) c t4) = R (B t1 a t2) b (B t3 c t4)" |
"baliR t1 a t2 = B t1 a t2"

fun ins :: "'a::linorder ⇒ 'a rbt ⇒ 'a rbt" where
"ins x Leaf = R Leaf x Leaf" |
"ins x (B l a r) =
  (case cmp x a of
     LT ⇒ baliL (ins x l) a r |
     GT ⇒ baliR l a (ins x r) |
     EQ ⇒ B l a r)" |
"ins x (R l a r) =
  (case cmp x a of
    LT ⇒ R (ins x l) a r |
    GT ⇒ R l a (ins x r) |
    EQ ⇒ R l a r)"

fun paint :: "color ⇒ 'a rbt ⇒ 'a rbt" where
"paint c Leaf = Leaf" |
"paint c (Node l (a,_) r) = Node l (a,c) r"

definition insert :: "'a::linorder ⇒ 'a rbt ⇒ 'a rbt" where
"insert x t = paint Black (ins x t)"
```

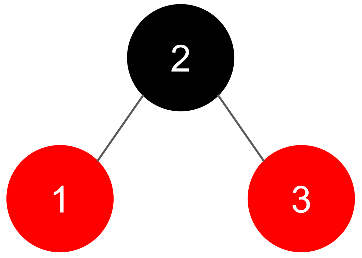
In the end, these equations are just mathematical functions, so we can use standard proof techniques.

source code: https://github.com/metp/kernel-vs-verified-rbt

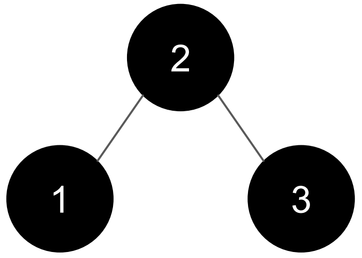
RBT testing pipeline



Just compare the verified trees against the Linux ones, right?



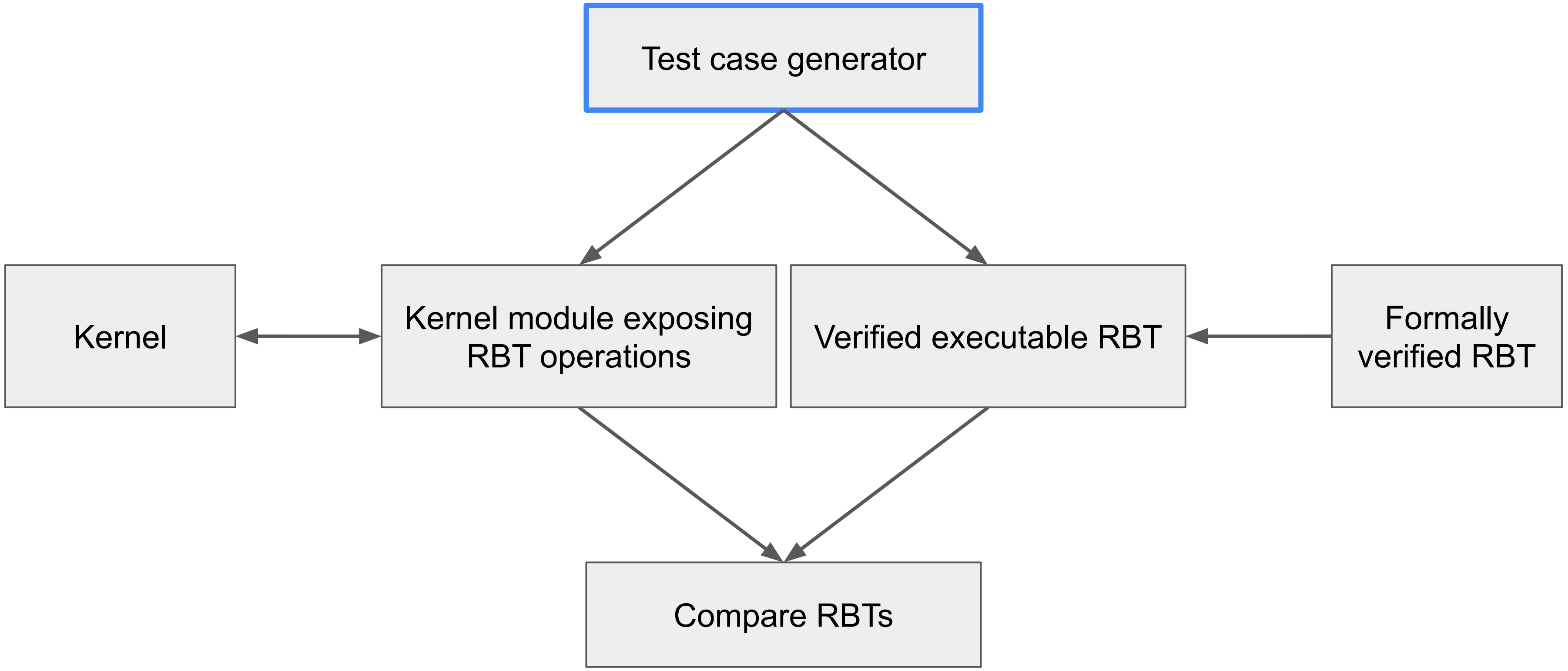
Linux RBT



Verified RBT

source code: https://github.com/metp/kernel-vs-verified-rbt

RBT testing pipeline



Three different test case generators

- Random: use random values as input
- Exhaustive: use all values within a small scope as input
- Symbolic: use symbolic values as input and refine symbolic values as needed

			Coverage %	
Strategy	Operations	Time (min)	core	core+aux
random	50,000	134	92.35	72.22
exhaustive	36,288,000	13	93.44	73.08
symbolic	248	110	95.08	74.36
Total	36,338,248	257	99.45	77.78

Discussions

1. Test oracles accepted by the kernel community?

    How to get this upstream?

2. How to collect coverage of globally used functions?

```
lib/Makefile:
# These files are disabled because they produce lots of non-interesting and/or
# flaky coverage that is not a function of syscall inputs.
# For example, rbtree can be global and individual rotations don't
# correlate with inputs.
KCOV_INSTRUMENT_rbtree.o := n
```

3. Symbolic execution kernel pipeline in combination with oracles?