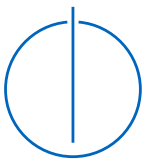
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Testing the Red-Black Tree Implementation of the Linux Kernel against a Formally Verified Variant

Mete Polat



DEPARTMENT OF INFORMATICS

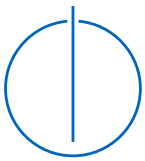
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Testing the Red-Black Tree Implementation of the Linux Kernel against a Formally Verified Variant

Testen der Rot-Schwarz-Baum Implementierung des Linux Kernels gegen eine formal verifizierte Version

Author:	Mete Polat
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Dr. Lukas Bulwahn
Submission Date:	12.08.2021



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 12.08.2021 Mete Polat

Acknowledgments

I want to thank my advisor Lukas Bulwahn for encouraging me on this thesis topic and guiding me in challenging parts with valuable input and feedback. In addition, I want to thank my supervisor Tobias Nipkow for accepting this thesis topic in the first place, for his indispensable participation in my continuous discussions with Lukas Bulwahn, and most importantly, for arousing my enthusiasm in functional programming and interactive theorem proving with his phenomenal lectures. Lastly, I want to thank my family and friends. They—as always—gave me enough space, energy, and love to focus successfully on writing.

Abstract

This thesis shows how to construct evidence of correctness for the Red-Black tree (RBT) implementation of the Linux kernel through testing and formal verification. First, it describes how to verify the results of kernel RBT operations by comparing them against the results of a formally verified RBT variant. Second, it shows how modifying this verified implementation and identifying new invariants makes it possible to prove the correctness of the kernel RBT insert algorithm in Isabelle/HOL.

Contents

1 Introduction

Linux is one of the most widely deployed operating system kernels in the world. Its feature richness and availability as free software make it a popular choice for all kinds of domains, from embedded systems to high-performance computing [15]. With the recent advances in AI, today's software systems have become more autonomous and begin establishing themselves in safety-critical and human-centered environments. While Linux is an attractive kernel for these systems, one desirable requirement it does not currently meet is a high assurance of functional correctness. The Linux kernel documentation explicitly states: "The developers' goal is to fix all known regressions before the stable release is made. In the real world, this kind of perfection is hard to achieve [...]. So most 5.x kernels go out with a handful of known regressions [...]" [16]. The Linux Foundation hosts numerous projects to accelerate the improvement and adoption of Linux in these new domains. Notable projects are "Enabling Linux in Safety Critical Applications (ELISA)," "Automotive Grade Linux (AGL)," and the "Civil Infrastructure Platform (CIP)."

To increase the confidence of a functionally correct Linux kernel, this thesis tests an integral part of it, the Red-Black tree (RBT) data structure. Various schedulers, filesystems, and the memory subsystem are (among others) users of Red-Black trees [11]. The kernel already provides some tests to check the correctness of the RBT implementation. However, these only test one specific sequence of RBT operations—insert n elements into the tree and delete them again. This testing schema cannot rule out possible errors for other sequences because the RBT invariants are ambiguous. For example, inserting and deleting the same element in a tree can—depending on the implementation—result in a structurally different tree than before executing these two operations; or, mathematically speaking, the composition of these operations can be non-idempotent. Moreover, the correctness check after every operation partly relies on the correctness of unverified auxiliary Red-Black tree functions and macros.

I tested the insert and delete operation of the kernel RBT by comparing their results against an RBT variant that was formally verified in the interactive theorem prover Isabelle. For this, I developed a testing pipeline consisting of a loadable kernel module, a verified Haskell Red-Black tree implementation, different input generation strategies, and two RBT comparison strategies. This testing pipeline did not reveal any errors after millions of executed and verified operations. Moreover, I confirmed that the testing pipeline could find a bug if one existed in the kernel implementation.

In addition, I formalized the kernel RBT insert algorithm in Isabelle/HOL and proved it correct by altering the existing HOL formalization and adding new lemmas.

These two approaches together form a higher assurance and confidence of a functionally correct Linux Red-Black tree implementation.

1.1 Outline

The following chapter introduces the interactive theorem prover Isabelle, its code generation feature, Red-Black trees, Linux kernel modules, testing facilities in Linux, and communication with the Linux kernel. Chapter 3 explains the building blocks of the testing pipeline and how I verified the kernel insert operation in Isabelle/HOL. Chapter 4 executes the testing pipeline and evaluates the results. Chapter 5 presents related work. Finally, chapter 6 discusses possible future work and the applicability of testing other kernel parts with the approach presented in this thesis.

2 Preliminaries

Throughout this thesis, I use the terms “Linux,” “kernel,” and the combination of both interchangeably unless otherwise noted. This thesis is based on kernel version 5.12, the most recent one as of writing.

2.1 Isabelle

Isabelle is an interactive proof assistant used to verify software and prove mathematical theorems. Verification works by proving that a concrete implementation follows a previously formalized specification. Users can choose between different formal languages such as “Isabelle/HOL” for higher-order logic or “Isabelle/ZF” for first-order logic with Zermelo–Fraenkel set theory [8]. Structured proofs resembling the style of classic paper ones can be constructed with “Isabelle/Isar”. All formal development relevant for this thesis is formalized in the logic Isabelle/HOL and uses Isabelle/Isar as the structured proof language.

2.1.1 Code Generation

Code generation is a feature of Isabelle “to turn HOL specifications into corresponding executable programs in the languages SML, OCaml, Haskell and Scala” [6]. I use it in this thesis to turn the existing HOL Red-Black tree implementation into Haskell code.

2.2 Red-Black Trees

A Red-Black tree (RBT) is a binary version of an order 4 B-tree, where every link from one node to the next is either “red” or “black.”[1] A B-tree is a balanced search tree with support for more than two children. The *order* of a B-tree describes the upper bound of supported children. In an order 4 B-tree, every node must have between 2 to 4 children; therefore, they are often called 2–4 (or 2–3–4) trees. Likewise, the number of keys in a

[1] In fact, a Red-Black tree can mimic a B-tree of any order [5]. The implementations of the different B-tree orderings only differ in when they rotate subtrees and when they manipulate link colors. Nowadays the term Red-Black tree typically describes the order 4 B-tree variant.

node has to be one less than the number of children. Moreover, every path from the root to a leaf has to be equally long. Thus the height of a B-tree is logarithmic in its size [5]. Red-Black trees adopt similar balancing properties as explained below.

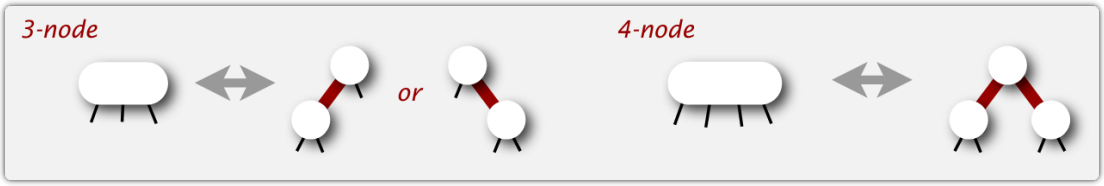


Figure 2.1: Embedding of 3-nodes and 4-nodes into an RBT [14]

Figure 2.1 shows how Red-Black trees mimic 3-nodes and 4-nodes. 3-nodes split into two 2-nodes that connect with a red link. They can either lean to the left or the right side. For 4-nodes, the middle key acts as a new parent with red links to the left and the right key. Finally, a 2-node is just a node with black links to its children. Therefore, every RBT uniquely maps to an order 4 B-tree, but not the other way round, as there are two variants to encode a 3-node. Because of this ambiguity, classic RBT implementations have to handle many cases. Left-Leaning Red-Black trees reduce this complexity by forcing 3-nodes to lean to the left. However, neither the kernel nor the Isabelle/HOL implementation uses this variant.

Based on this conversion of order 4 B-tree nodes to binary nodes with color, a Red-Black tree has to satisfy the following properties and invariants (the color of a node is the link color to its parent) [13, Red-Black Trees]:

1. The root is black.
2. Every leaf is considered black.
3. If a node is red, its children are black.
4. All paths from a node to a leaf have the same number of black nodes.

2.2.1 Height

Because an RBT is just an embedding of a B-tree, and all B-trees have a logarithmic height, RBTs are likewise logarithmically bounded. When one path beginning from the root consists of n black links, then the worst-case path length in this RBT consists of altering black and red links up to length $2n$. The colors have to alter because of invariant 3, and there have to be exactly n black links on the path because of invariant 4. Therefore, the worst-case path length is $2n$, and the height of a Red-Black tree is likewise logarithmically bounded in its size ($\leq 2\log_2 n$) [5].

2.2.2 Red-Black Trees in Isabelle

The Red-Black trees in Isabelle/HOL are encoded with an inductive data type and a predicate rbt to capture the tree's invariant:

```
datatype color = Red | Black
type_synonym 'a rbt = "('a*color) tree"

abbreviation R where "R l a r ≡ Node l (a, Red) r"
abbreviation B where "B l a r ≡ Node l (a, Black) r" fun color :: "'a rbt ⇒ color"
where
"color Leaf = Black" |
"color (Node _ (_, c) _) = c"

definition rbt :: "'a rbt ⇒ bool" where
"rbt t = (invc t ∧ invh t ∧ color t = Black)"
```

"invc" and "color t = Black" ensure that the tree is correctly colored (invariant 3 and 1). "invh" checks if all paths from a node to a leaf have the same number of black nodes (invariant 4). Finally, the "color" function enforces all leaves to be black (invariant 2). The definition "rbt" does not force the tree to be a binary search tree (BST) [8, HOL/Library/Tree.thy]:

```
fun bst_wrt :: "('a ⇒ 'a ⇒ bool) ⇒ 'a tree ⇒ bool" where
"bst_wrt P ⟨⟩ ⟷ True" |
"bst_wrt P ⟨l, a, r⟩ ⟷
 (∀x∈set_tree l. P x a) ∧ (∀x∈set_tree r. P a x) ∧ bst_wrt P l ∧ bst_wrt P r"

abbreviation bst :: "('a::linorder) tree ⇒ bool" where
"bst ≡ bst_wrt (<)"
```

Including the BST invariant is not required because every defined RBT operation creates or preserves it. This behavior was confirmed by verifying that the RBT implementation conforms to the "inorder Set" specification [8, HOL/Data_Structures/Set_Specs.thy]. This specification expects from every operation to modify the tree in a way that the inorder of the RBT is sorted with respect to $(<)$. From this, it follows that the verified RBT implementation is a BST [8, HOL/Library/Tree.thy]:

```
lemma bst_iff_sorted_wrt_less: "bst t ⟷ sorted_wrt (<) (inorder t)"
```

2.2.3 Red-Black Trees in Linux

Figure 2.2 shows the kernel implementation of an RBT node. The struct field "__rb_parent_color" stores the pointer to the parent node, and at the same time, the color of the node itself. This works because instances of "struct rb_node" are at minimum four

bytes aligned (the exact value is architecture-dependent). Thus the first two bits of an "`struct rb_node`" pointer are always zero. The kernel uses the first bit of "`__rb_parent_color`" to store the node color and provides special functions to access the actual pointer and color value [11].[2]

```
struct rb_node {
        unsigned long __rb_parent_color;
        struct rb_node *rb_right;
        struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
```

Figure 2.2: Linux kernel RBT node [11, `include/linux/rbtree.h`]

"`struct rb_node`" does not have a "`data`" pointer that could point to the data a node stores. Instead, one must define a custom struct that stores (a pointer to) the data and contains a field of type "`struct rb_node`." Data structures that do not provide a data pointer on their own but expect to get embedded into a custom struct are called *intrusive containers*. The *Fuchsia* operating system overviews the advantages and disadvantages of (non-) intrusive containers in an operating system context [7].

2.3 Linux

2.3.1 Loadable Modules

Modules are compiled kernel code that can be inserted into a running kernel to extend its capabilities. Rather than building all supported device drivers and functionality into one big kernel, they can be distributed separately and loaded on demand.

For the work of this thesis, I used a loadable kernel module that exposes the kernel RBT insert and delete operations, and a custom operation for printing out the current RBT for userspace programs.

2.3.2 Testing

The kernel documentation describes a number of different tools for testing the kernel and divides them into *unit testing, code coverage* and *dynamic analysis* tools [16, *Kernel Testing Guide*]. The first two categories are of particular interest for this thesis.

[2] The source code in Linux version 5.12 has that explicit alignment attribute, but it may be obsolete and get removed once the patch "*rbtree: remove unneeded explicit alignment in struct rb_node*" is integrated [1].

Depending on the test target, tests can either be executed in the kernel using KUnit or in a normal userspace application using kselftests. KUnit provides an easy way to test small, self-contained functionality directly in the kernel without exposing it. Kselftests are favorable for testing an exposed kernel functionality such as a system call. If the kernel does not directly expose a to-be-tested functionality, exposing it with the help of a kernel module makes it possible to write test cases in userspace. One reason for doing so is when writing a test case in the kernel would be too demanding.

Code coverage can be analyzed using gcov, the coverage testing tool shipped with the GNU C compiler suite. However, gcov does not differentiate which userspace program led to the execution of a kernel code area but collects coverage information of the whole running kernel. To collect the kernel code coverage initiated from one user process, kcov can be used instead. Kcov was initially developed for syzkaller, a coverage-guided kernel fuzzer [17].

2.3.3 Communication between kernelspace and userspace

The Linux kernel provides different interfaces for userspace programs to interact with it. The traditional one is a system call. However, alternative interfaces can be more comfortable depending on how and what part of the kernel someone wants to use. The following non-exhaustive list gives an overview of different Linux kernel interfaces [16, The Linux driver implementer's API guide, Filesystems in the Linux kernel]:

- **System calls** are suitable for functionality that is not tied to a specific physical device and is of possible interest for all userspace applications.
- **ioctl based interfaces** are commonly used to communicate with device drivers. Instead of defining an own system call for every driver, one can use the "`ioctl`" system call. A specific driver is specified using a file descriptor, and this driver can mostly decide on its own on the semantics of the remaining passed parameters. Device drivers that use ioctl have to provide a stable ABI (Application Binary Interface); that is, future kernels should not change existing interface endpoints for backward compatibility reasons.
- **sysfs** is a pseudo-filesystem commonly mounted on `/sys/` and provides an interface to expose state information of the hardware, drivers, and other kernel data structures. For example, the current temperature of every CPU core is listed there. Sysfs users also have to provide a stable ABI.
- **debugfs** is similar to sysfs but with more freedom. It does not have to provide a stable ABI and is by default only accessible as the root user. A prominent user

of debugfs is the previously mentioned KUnit framework that publishes its test results there. The testing module developed for this thesis also uses debugfs for communication between kernel- and userspace. Debugfs is typically mounted on `/sys/kernel/debug/`.

3 Approach

This thesis aims to produce a high assurance of correctness for the Linux Red-Black tree implementation. I obtain this assurance by comparing the Red-Black trees generated by the Linux kernel with Red-Black trees generated by a verified implementation. To do this, I used a flexible testing pipeline that is composed of three parts. **Test case generation** is responsible for proposing RBT test cases that cover many distinct (corner-) cases. The "test harness" program executes these test cases on the Kernel RBT and on the verified RBT. I generated the **verified RBT variant** from a proven Isabelle/HOL implementation. After the test harness executed an operation on both implementations, the test harness compares the generated trees with a specific **comparison method**. Table 3.1 lists these three categories and their supported strategies. Strategies of different categories can be freely combined, though some combinations suit better than others.

3.1 Test Case Generation

The primary responsibility of the test case generation stage is to propose test cases that cover a large part of the kernel RBT codebase. A *test case* consists of a sequence of RBT operations. An RBT *operation* consists of a *command* and an optional *key*. The generation stage supports three RBT-mutable commands:

```
data Cmd = Reset | Insert | Delete
```

Resetting a kernel tree means removing all keys from the tree and freeing the occupied space. The analog verified operation is just returning an empty tree (a leaf). Immutable commands like lookup or inorder could also be explicitly added to "`Cmd`". However, as these commands are trivial and adding them would considerably increase the testing space, I left them out. Nevertheless, deleting a key in the kernel implementation requires a lookup first. Therefore, delete tests lookup implicitly.

The test harness stores 64-bit unsigned integers (Word64 in Haskell) as the keys in the Red-Black tree. This is mainly because the kernel provides convenient helper functions to parse userspace string values of this type in kernelspace. The actual type of stored data does not play any role in the kernel RBT implementation as it uses an intrinsic data structure for the RBT nodes (see subsection 2.2.3). It is the responsibility of the kernel RBT user to provide appropriate comparison functions (see subsection 3.2.1).

Strategy	Description
Test Case Generation	Strategies for generating test cases
Random	Generate test cases randomly using a pseudo-random number generator
Exhaustive	Generate almost all possible test cases up to a limit
Symbolic	Generate test cases with a symbolic execution engine
Comparison Method	How to compare the kernel RBT against the verified RBT
Structural	The kernel RBT is only correct if it is equal to the verified RBT
Invariant	RBTs can differ, but the kernel RBT must satisfy all RBT invariants and must have the same inorder as the verified RBT
Verified RBT Invariant	Which verified Isabelle/HOL RBT variant to use for comparison
Default	The RBT variant distributed with Isabelle2021 and described in *Functional Algorithms, Verified!* [13]
Linux to HOL	The RBT variant whose insert operation equals the kernel RBT insert

Table 3.1: An overview of all implemented strategies for the testing pipeline.

If the test harness is supposed to generate test cases on its own rather than importing these generated test cases from external tools like symbolic execution engines (see subsection 3.1.3), then combining the list of commands with input keys is done with the "`buildInput`" function:

```
type Input = (Cmd,Word64)
type TestCase a = [a]

buildInput :: [Word64] -> [Word64] -> [Cmd] -> TestCase Input
buildInput _ _ [] = []
buildInput (i:is) ds (Insert : cs) = (Insert, i) : buildInput is ds cs
buildInput is (d:ds) (Delete : cs) = (Delete, d) : buildInput is ds cs
buildInput _ _ _ = undefined
```

The first two lists—one for the insert and one for the delete command—provide the keys that will be combined with the commands from the third list. The two lists have to be long enough to match all occurrences of their respective command. The keys for the delete operation (the second list) should optimally be the same as the keys for the insert

operation (the first list). Otherwise, many delete operations will not change anything because the chance could be high (depending on how the lists are constructed) that a key to delete has not been inserted into the tree previously. "`buildinput`" is undefined for reset commands because they do not expect a key. It is the responsibility of the input generation strategy to populate this function with meaningful lists.

After building a list of test cases, they have to get executed on the verified Haskell RBT, and kernel RBT:

```
type IRBT = Tree (Word64, Color)

data Result = Result {
  input :: Input,
  vTree :: IRBT,
  kTreeIO :: Kernel.Handle -> IO IRBT }

vCmd :: IRBT -> Input -> IRBT
kCmd :: Input -> Kernel.Handle -> IO IRBT

buildResults :: [TestCase Input] -> [TestCase Result]
buildResults testCases = do
  inputs <- testCases
  let vTrees = tail $ scanl vCmd RBT.empty inputs
  let kTrees = map kCmd inputs
  return $ zipWith3 Result inputs vTrees kTrees
```

Functions "`kCmd`" and "`vCmd`" execute the right RBT operation based on the passed command. Data type "`Result`" stores the input (command & key), the resulting verified Haskell RBT, and a function that returns the resulting kernel RBT. The "`Kernel.Handle`" parameter is used for communication with the kernel (see subsection 3.2.3). Subsection 3.2.3 explains in detail how the execution and communication with the kernel module works.

3.1.1 Random

The *random* test case generation strategy populates the lists for "`buildInput`" in the following way:

```
random :: Word64 -> Int -> [TestCase Result]
random n seed = do
  let rndCmds = randoms seed
  let rndXs = randoms seed
```

```
  let inputs = genericTake n (buildInput rndXs rndXs rndCmds)
  buildResults [inputs]
```

The "randoms" function is a pseudo-random number generator that expects a seed value from the test harness user. It generates an infinite list of values for types implementing a corresponding type class that specifies the values (or a range) to consider in the generation procedure. For the data type "Cmd", these are the insert and delete commands. The reset command is not required because the test harness resets the kernel RBT every time before executing a new test case.

3.1.2 Exhaustive (scoped)

Exhaustive testing describes the procedure to test a program with all possible inputs. In almost all cases, this is impossible. In the context of data structures, there always exists an infinite sequence of operations to test. One solution to tackle this problem is limiting the number of operations per test case on an RBT, and partitioning test cases into equivalence classes [12]. It is reasonable to assume that if a test case hits an error, similar test cases do as well. Therefore, these cases belong into one equivalence class and testing just one of them is fine. This significantly reduces the testing space. For example, in the case of a BST, if the test case "[(Insert,1), (Insert,2)]" leads to an error, then "[(Insert,4), (Insert,8)]" probably also does because BSTs insert elements depending on their linear order.

The "exhaustive" test case generation strategy limits the number of operations on an RBT and puts all test cases with the same sequence of commands and linear ordering into one equivalence class:

```
exhaustive :: Word64 -> [TestCase Result]
exhaustive n = do
  let rep = genericReplicate
  let distributions = [rep i Insert ++ rep (n-i) Delete | i <- [n,n-1..]]
  let testCases =
    concatMap (perm . buildInput [1..n] [1..n]) distributions
  buildResults testCases
```

For every command distribution of length n, it builds a test case using the previously introduced "buildInput" function. Then it generates all permutations of this test case. In total, "exhaustive" generates $(n+1)n!$ cases. For $n = 3$, following cases are generated (for space reason, I = Insert and D = Delete):

```
distributions = [ [I,I,I], [I,I,D], [I,D,D], [D,D,D] ]
testCases = [ [(I,1),(I,2),(I,3)], [(I,2),(I,1),(I,3)],
  [(I,3),(I,2),(I,1)], [(I,2),(I,3),(I,1)], ... , [(D,1),(D,3),(D,2)] ]
```

Because I use for both—the insert and delete command—the keys from one to n to pair with the commands, a sequence like "`[(I,1),(I,2),(D,2)]`" is not an element of "`testCases`". For the "`[I,I,D]`" command distribution, The "`buildInput`" function pairs the first and only delete command it encounters with the first key from the delete list—a one. Therefore, the pair "`(D,2)`" is never created for the distribution "`[I,I,D]`." Considering this case would unnecessarily complicate the simple structure of the exhaustive strategy. Instead, using a higher n generates a test case in which the previous test case is part of: "`[(I,1),(I,2),(D,2),(D,1)]`." The test harness compares the kernel RBT against the verified RBT after each operation, not just after executing all operations in a test case. Therefore this sequence of length four covers the missing case.

3.1.3 Symbolic

A completely different approach to automatically generating a set of input sequences is to use a *symbolic execution* tool [2]:

> Instead of running code on manually- or randomly-constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

After generating test cases by symbolically executing an unverified program variant, they can be automatically replayed on the unverified and verified program variant to compare their results. Symbolic execution can generate a high code coverage and is therefore also used in this thesis. Nevertheless, even confirming that all test cases for every code path lead to the same results when replaying on an unverified and verified program does not mean that the previously unverified variant is now verified. The unverified version could lack a code path and thus a test case for this path is never generated [12].

Two concerns that symbolic execution tools commonly face are how to deal with the outer execution environment correctly and that every branch at least doubles the number of paths the system has to follow (exponential growth; also called the path explosion problem) [2]. Concerning this thesis, two additional difficulties arise. First, the Linux

Red-Black tree library lives in kernelspace, but most symbolic execution tools only work in userspace. Second, there is the question of how to model unbounded symbolic data structures. I will elaborate on both difficulties in the following two subsections.

Difficulty: Symbolic Execution of Kernel Code

Classic symbolic execution tools only work in userspace, and symbolically executing a big monolithic kernel like Linux suffers from the path explosion problem. To overcome these issues, I isolated and lifted the kernel RBT implementation to userspace. The implementation was already largely independent of the rest of the kernel. To successfully compile the code in userspace, I had to remove the concurrency support from the RBT implementation because it caused a lot of dependencies.

An alternative would be to use a tool that natively supports symbolic execution of kernel code. According to my research, the S^2E platform is the only tool that is capable of doing so. Under the hood, it combines modified versions of the KLEE symbolic execution engine and the QEMU machine virtualizer [3]. To tackle the path explosion problem, it trades off the completeness, soundness (or both) of the generated paths for a lower time complexity. Depending on the chosen consistency model, execution of code paths can seamlessly switch between symbolic (generation of multiple paths when branching) and concrete (follow just one branch) execution, possibly generating unsound and incomplete paths when switching between these modes [3].

While lifting the kernel code to userspace required modifications to the actual code, most of them were trivial and just removed code fragments. The advantage of symbolically executing the RBT code in userspace is that it does not require trade-offs as with S^2E. Therefore, I symbolically analyzed the lifted code in userspace using KLEE.

Difficulty: Unbounded Symbolic Data Structures

My initial goal was to have a fully symbolic Red-Black tree and let KLEE figure out all possible code paths. However, this is not possible because (in theory) data structures can be infinitely large. Symbolically executing a lookup function on a fully symbolic tree would never terminate because the symbolic execution engine always has to assume that the left and the right subtrees are not leaves and the element to lookup is in one of them. Thus, I had to specify some bound.

There are different options to bound an RBT. For example, one could define a limit on the number of nodes, the height, or the sequence of operations to execute on the RBT. One significant advantage of defining a limit on the sequence of operations is that it fits nicely into the existing testing pipeline, which is why I followed this approach.

To symbolically execute the RBT implementation, KLEE needs a wrapper with an entry point:

```
1  int main() {
2          struct rb_root rbt = RB_ROOT; struct rb_node *node;
3          struct data *data; struct stat stat;
4          off_t size; char cmd, value;
5
6          fstat(STDIN_FILENO, &stat);
7          size = stat.st_size;
8          assert(size % 2 == 0);
9
10         for (int i = 0; i < size / 2; i++) {
11                 cmd = getchar();
12                 value = getchar();
13
14                 if (cmd) {
15                         // insert value into RBT, similar to kernel module
16                 } else {
17                         // delete value from RBT, similar to kernel module
18                 }
19         }
20 }
```

This program iteratively reads two bytes from stdin until it reaches its end. They are used to build the command–value pairs. The first byte indicates the command (insert or delete), and the second one its value[1]. Line 6 to 8 assert that the file size is even so that pairing does not fail later. An assertion statement is preferable here because it does not branch while symbolically executing, like an "`if`" check would do. KLEE can now use a symbolic stdin file with a specific size N from which the wrapper, starting from an empty tree, can build up all possible RBTs using $N/2$ operations (assuming N is even):

```
$ klee --only-output-states-covering-new --libc=uclibc --posix-runtime \
       userspace_RBT_wrapper.bc -sym-stdin <N>
```

The "`libc`" and "`posix`" arguments are required because the wrapper is interacting with the environment (using "`fstat()`" and "`getchar()`").

In essence, the "symbolic" and "exhaustive" strategies are considerably similar. They both try to hit every path given a bound on the test case length. In the exhaustive strategy, I used my intuition and knowledge about the algorithm to partition test cases into equivalence classes to lower the number of test cases that follow the same path. In the symbolic strategy, this happens automatically using the "`--only-output-states-covering-new`" KLEE flag. For $n = 6$ (3 operations), KLEE generates the following statistics:

[1] The reason why I use 8 bit and not 64-bit for the value type like I did in the kernel module and test harness is because of a long-standing bug in KLEE that makes parsing longer values impossible without unwanted changes [9][Issue #30]. Nevertheless, 8-bit should be large enough to not restrict the possibility for achieving a high code coverage.

```
KLEE: done: completed paths = 48
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 15
```

KLEE pruned 33 out of 48 paths because they equal one of the 15 existing ones. After running the above command, one has to extract the generated concrete stdin files for all paths from KLEE's binary test case format "`*.ktest`" which KLEE stored in the "`klee-last`" directory:

```
$ ktest-tool --extract stdin klee-last/*.ktest
```

I have integrated a parser into the test harness that reads (from a specified directory) all these stdin files that contain sequences of command–value pairs. After parsing them using "`parseStdins`," one can directly build the results with the "`buildResults`" function like in the previous test strategies:

```
parseStdins :: FilePath ->IO (Either String [TestCase Input])

symbolic :: FilePath -> IO (Either String [TestCase Result])
symbolic = fmap (buildResults <$>) . parseStdins
```

The only difference to the previous strategies is that symbolic returns an "`IO (Either String [TestCase Result])`" rather than just "`[TestCase Result]`" because it parses a file (IO), and this can fail (Either)[2]. The "`main`" function handles a possible parsing error.

3.2 Comparing Kernel and Verified Trees

3.2.1 RBT Kernel Module

Building every supported device driver and kernel feature into one big kernel can unnecessarily impact performance and available storage space if only a small subset of these drivers and functionality is used. As an alternative to one big kernel, Linux supports a modular approach. Kernel builders (like Linux distributions) can decide which non-fundamental features and device drivers (called modules) they want to include into their kernel directly. The other modules can be inserted (automatically) into a running kernel when needed. Therefore they are also called *loadable* modules.

I developed a module that supports three kernel Red-Black tree operations: Reset (empty RBT), insert, and delete. Additionally, I added a function to print out the kernel

[2] One could arguable use Haskell's "`ExceptT`" monad transformer to handle the nested IO and Either monad, but given the simple structure of the test harness, it turned out that just nesting them keeps the code cleaner.

RBT so the verified Haskell implementation can parse it. To use these operations and the print function, I expose an interface that consists of two virtual files on the debugfs filesystem. Userspace programs can then interact with these files to alter the kernel RBT[3]:

- **/sys/kernel/debug/rbt_if/key** Reads or writes to this file set or get the key used for insert and delete.
- **/sys/kernel/debug/rbt_if/cmd** Writing `0` to this file resets the kernel RBT, `1` inserts the previously specified key, and `2` deletes the key from the kernel RBT. Reading this file prints out the current RBT.

The following listing shows the module code. I removed some bits of minor relevance like the include header lines.

```
1  // #include ...
2  enum Cmd { RESET, INSERT, DELETE };
3
4  static struct dentry *rbt_if_root;
5  static struct rb_root rbt = RB_ROOT;
6  static u64 input_key;
7
8  struct data {
9          struct rb_node node;
10         u64 key;
11 };
12
13 #define data_from_node(from) (rb_entry(from, struct data, node))
14
15 #define print(...) seq_printf(m, __VA_ARGS__)
16 #define print_parent print(" (%llu,%s) ", \
17                 data_from_node(parent)->key, rb_is_red(parent) ? "Red" : "Black")
18
19 static int cmd_show(struct seq_file *m, void *p) {
20         struct rb_node *node = rbt.rb_node, *parent = NULL;
21         bool left = false;
22         while(true) {
23                 if (!node) {
24                         print("Leaf");
25
26                         if (!parent)
27                                 break;
28
29                         if (left) {
30                                 print_parent;
31                                 node = parent->rb_right;
32                                 left = false;
33                         } else {
34                                 while (rb_parent(parent) && parent->rb_right == node) {
```

[3]Debugfs is typically mounted on "`/sys/kernel/debug`" but it does not have to; the user or the Linux distribution can choose another path [16, Filesystems in the Linux kernel].

```
35                                      print(")");
36                                      node = parent;
37                                      parent = rb_parent(node);
38                              }
39
40                              if (parent->rb_right == node)
41                                      break;
42
43                              print_parent;
44                              node = parent->rb_right;
45                              left = false;
46                      }
47              } else {
48                      if (parent)
49                              print("(");
50
51                      print("Node ");
52                      parent = node;
53                      node = node->rb_left;
54                      left = true;
55              }
56
57      }
58      print("\n");
59      return 0;
60  }
```

Function "`cmd_show`" iteratively prints the kernel RBT in the style "`Node (Leaf (3,Black) (Node ...))`" once the "`cmd`" file is read. To not burden the kernel stack with too many function frames, I chose an iterative algorithm over a recursive one.

```
61  static int key_cmp(const void *_a, const struct rb_node *_b)
62  {
63      u64 a = *(typeof(a) *) _a;
64      u64 b = data_from_node(_b)->key;
65      if (a < b)
66              return -1;
67      if (a > b)
68              return 1;
69      else
70              return 0;
71  }
72
73  static int node_cmp(struct rb_node *a, const struct rb_node *b)
74  {
75      return key_cmp(&data_from_node(a)->key, b);
76  }
```

The internal kernel RBT interface expects a comparison function for the data stored in the RBT. Function "`key_cmp`" is used to find the correct node for deletion given some key. Function "`node_cmp`" is used to find the correct position to insert a new node.

```
77  ssize_t cmd_exec(struct file *file, const char __user *ubuf, size_t len, loff_t *offp) {
78      int cmd;
```

```
79          struct data *data, *_n;
80          struct rb_node *node;
81          int ret = kstrtoint_from_user(ubuf, len, 10, &cmd);
82          if (ret)
83                  return ret;
84
85          switch (cmd) {
86          case RESET:
87                  rbtree_postorder_for_each_entry_safe(data, _n, &rbt, node)
88                          kfree(data);
89                  rbt = RB_ROOT;
90                  break;
91          case INSERT:
92                  data = kzalloc(sizeof(*data), GFP_KERNEL);
93                  data->key = input_key;
94                  rb_find_add(&data->node, &rbt, node_cmp);
95                  break;
96          case DELETE:
97                  node = rb_find(&input_key, &rbt, key_cmp);
98                  if (!node)
99                          break;
100                 rb_erase(node, &rbt);
101                 kfree(data_from_node(node));
102                 break;
103         default:
104                 return -EINVAL;
105         }
106         return len;
107 }
```

Function "`cmd_exec`" is responsible for executing a requested operation on the kernel RBT. Like the Isabelle/HOL variant, the function "`rb_find_add`" only inserts new elements into the tree and does not allow duplicates. In the delete case, the kernel does not provide a dedicated function to delete a specific key, but one must first find the right node separately before "erasing" it.

```
108 static int cmd_open(struct inode *inode, struct file *file) {
109         return single_open(file, cmd_show, inode->i_private);
110 }
111
112 static const struct file_operations cmd_fops = {
113         .owner  = THIS_MODULE,
114         .open   = cmd_open,
115         .read   = seq_read,
116         .write  = cmd_exec,
117         .llseek  = seq_lseek,
118         .release = single_release,
119 };
120
121 int __init rbt_if_init(void) {
122         rbt_if_root = debugfs_create_dir("rbt_if", NULL);
123         if (IS_ERR(rbt_if_root)) {
124                 if (rbt_if_root == ERR_PTR(-ENODEV))
125                         pr_err("debugfs support is missing for this kernel. \
```

```
126                                     The test harness uses it for communication with this module");
127             return PTR_ERR(rbt_if_root);
128     }
129
130     debugfs_create_file("cmd", 0644, rbt_if_root, NULL, &cmd_fops);
131     debugfs_create_u64("key", 0644, rbt_if_root, &input_key);
132     return 0;
133 }
134
135 void __exit rbt_if_exit(void) {
136     struct data *_n, *pos;
137     debugfs_remove_recursive(rbt_if_root);
138     rbtree_postorder_for_each_entry_safe(pos, _n, &rbt, node)
139             kfree(pos);
140 }
141
142 // module information (license, author, entry & exit point)
```

The kernel calls the functions "`rbt_if_init`" and "`rbt_if_exit`" on module load and unload—they set up and clean up the two virtual files on the debugfs filesystem. The debugfs filesystem is a module on its own and does not have to be present on a running kernel. Line 123 checks if debugfs is available and, if not, fails with an error message. Moreover, on module exit, the RBT is freed.

3.2.2 Generating the Haskell RBT Implementation

To compare the kernel RBT to a verified RBT, one must first export the verified Isabelle/HOL to a target-language like Haskell. I chose Haskell because out of the languages the Isabelle code generator supports, Haskell is the one I am most comfortable with. To export the required RBT parts, I created the dedicated theory file "`RBT_export`":

```
theory RBT_export
imports
  "HOL-Data_Structures.RBT_Set"
  "HOL-Data_Structures.Tree2"
  "HOL-Library.Code_Target_Numeral"
   HOL.Orderings
begin
```

When importing "`HOL-Library.Code_Target_Numeral`", the code generator implements the HOL types "`nat`" and "`int`" by the target-language built-in integers [6][Adaption to target language].

```
code_printing
  type_class ord ⇀ (Haskell) "Prelude.Ord"
| constant Orderings.less ⇀ (Haskell) infixl 4 "<"
| constant Orderings.less_eq ⇀ (Haskell) infixl 4 "<="
| type_class preorder ⇀ (Haskell) "Prelude.Ord"
```

```
| type_class order ⇀ (Haskell) "Prelude.Ord"
| type_class linorder ⇀ (Haskell) "Prelude.Ord"
```

One problem when using the exported code is that it defines its own type classes for ordering elements and does not use Haskell's built-in one ("`Prelude.Ord`"). Thus, to store values of a specific type in the RBT, one must first create instances of these new type classes for the type to store. For example, to insert values of the Haskell type Word64, one must first create instances of the type classes "`Ord`", "`Preorder`", "`Order`", and "`Linorder`" because inserting an element into the RBT requires a linear ordering, and all previously listed classes are superclasses of "`Linorder`". To overcome this inconvenience and keep the RBT interface still generic, I mapped the different HOL ordering type classes and "compare" operators to Haskell's built-in ones. These mappings are not safe because a mistake could alter the behavior and invalidate the correctness of the Red-Black tree algorithm. Nevertheless, the mappings are straightforward and allow me to store in the RBT any Haskell type that is an instance of the built-in Ord type class. Keeping the functions generic allowed me to match the type stored in the Haskell RBT to the type stored in the kernel module RBT (unsigned 64-bit integer in the C kernel module; Word64 in Haskell).

Mapping all the different HOL orderings to Haskell's linear ordering is strictly speaking incorrect. Functions that previously worked on less strict orderings now only work for linear ordered types. However, all the Isabelle/HOL RBT functions that need an ordering expect linear ordered types, so this does not restrict anything in the RBT case.

```
definition rootBlack :: "'a rbt ⇒ bool" where
"rootBlack t ≡ color t = Black"

definition rbt :: "'a rbt ⇒ bool" where
"rbt t ≡ invc t ∧ invh t ∧ rootBlack t"

lemma "RBT_Set.rbt t ≡ rbt t"
  by (simp add: RBT_Set.rbt_def rbt_def rootBlack_def)
```

When the test harness detects a kernel tree that violates the Red-Black tree invariants and properties, it prints out which one exactly got violated. For this, every invariant must have a dedicated function that returns if this invariant holds for a given tree. The invariant that the root has to be black is directly checked in the original HOL "`rbt`" definition rather than in a dedicated function. Therefore, I moved that check into the "`rootBlack`" function, which is then called in my modified "`rbt`" definition. The lemma proves that my "`rbt`" definition is equivalent to the original one.

```
export_code
  rootBlack rbt
  RBT_Set.invc RBT_Set.invh
```

```
  RBT_Set.empty RBT_Set.insert RBT_Set.delete
  Tree2.inorder
  equal_tree_inst.equal_tree
in Haskell module_name RBT.Verified (string_classes)
```

end

The command "`export_code`" exports all specified functions, definitions, and dependencies and places them into one Haskell module. Everything explicitly defined is also exported in the Haskell module itself to make it available to other modules. To have a "compare trees" function, I exported "`equal_tree_inst.equal_tree`". While the code generator creates the Haskell instance "`Eq Color`", it does not create the instance "`Eq a => Eq (Tree a)`". Therefore, I have to compare trees with "`equal_tree`" and not with "`(==)`" in Haskell.

The "`string_classes`" parameter is important for this thesis. "It adds a '`deriving (Read, Show)`' clause to each appropriate datatype declaration" [6]. The "`Read a => Read (Tree a)`" instance is used for parsing the kernel trees and in the following subsection, I explain this usage in detail.

3.2.3 Interaction with the Kernel Module from Haskell

```
1  {-# LANGUAGE RecordWildCards #-}
2  module RBT.Kernel({- exports ... -}) where
3
4  -- imports ...
5
6  type IRBT = Tree (Word64, Color)
7
8  keyFile = "/sys/kernel/debug/rbt_if/key"
9  cmdFile = "/sys/kernel/debug/rbt_if/cmd"
10
11 data Cmd = Reset | Insert | Delete deriving (Enum, Bounded)
12
13 printCmd hdl = hPrint hdl . fromEnum
14
15 instance Show Cmd where
16   show Reset = "resetting"
17   show Insert = "inserting"
18   show Delete = "deleting"
```

The "`Cmd`" data constructors are in the same order as the "`enum Cmd`" in the kernel module. Thus, deriving "`Enum`" maps the same integer value to them. "`printCmd`" writes this value to a file associated with a kernel handle and thus executes a command on the kernel RBT.

```
19 data Handle = Handle {
20   keyHdl :: GHC.IO.Handle.Handle,
21   cmdHdl :: GHC.IO.Handle.Handle }
22
23 init :: IO RBT.Kernel.Handle
24 init = do
25   keyHdl <- openFile keyFile WriteMode
26   cmdHdl <- openFile cmdFile ReadWriteMode
27   let hdl = Handle{..}
28   hSetBuffering keyHdl LineBuffering
29   hSetBuffering cmdHdl LineBuffering
30   reset hdl
31   return hdl
32
33 cleanup :: RBT.Kernel.Handle -> IO ()
34 cleanup hdl = do
35   reset hdl
36   hClose $ keyHdl hdl
37   hClose $ cmdHdl hdl
```

Function "`init`" resets the kernel tree and returns two file handles—one for the "`key`" file and one for the "`cmd`" file. Function "`cleanup`" resets the kernel tree again and closes the handles.

```
38 exec :: Cmd -> Maybe Word64 -> RBT.Kernel.Handle -> IO IRBT
39 exec cmd x Handle{..} = do
40   whenJust x $ hPrint keyHdl
41   printCmd cmdHdl cmd
42   hSeek cmdHdl AbsoluteSeek 0
43   read <$> hGetLine cmdHdl
44
45 reset :: RBT.Kernel.Handle -> IO IRBT
46 reset hdl = do
47   tree <- exec Reset Nothing hdl
48   assert (RBT.equal_tree RBT.empty tree) $ return tree
49
50 insert :: RBT.Kernel.Handle -> Word64 -> IO IRBT
51 insert hdl x = exec Insert (Just x) hdl
52
53 delete :: RBT.Kernel.Handle -> Word64 -> IO IRBT
54 delete hdl x = exec Delete (Just x) hdl
```

Function "`exec`" first optionally sends a key to the kernel module by writing into the appropriate "`key`" file and then executes the given command. After execution, it parses the RBT string with the "`read`" function, which is only possible because the data type "`Tree a`" is deriving the "`Read`" type class.

3.2.4 Comparison Strategies

In general, there exist two methods to compare a kernel with a verified RBT. Both require that the test harness executes the same list of operations (insert/delete element) on the

verified Haskell implementation and on the kernel implementation. After each operation, the test harness has to compare the state of the kernel RBT with the correct Haskell RBT state. This comparison can either happen on a *structural* or *invariant* level.

On the structural level, the test harness expects that the kernel produces the same trees as the verified RBT after each operation. This approach has the advantage that it barely has to compare if both trees are built in exactly the same way (including node color and key), which is possible in linear time. A disadvantage is that it expects both algorithms to behave in the exact same way. Unideal, the RBT invariants introduced in Section 2.2 allow a certain degree of freedom. As an example, after inserting the numbers from 1 to 3, the kernel and original Isabelle/HOL implementation produce the following trees:

Verified: `B (B Leaf 1 Leaf) 2 (B Leaf 3 Leaf)`
Kernel: `B (R Leaf 1 Leaf) 2 (R Leaf 3 Leaf)`

Both are valid RBTs. In this case, only the colors differ. Inserting additional elements can lead to more structural differences. In subsection 3.2.5, I ported the kernel RBT insert variant to HOL and proved it correct. This makes structural comparisons at least for the insert function possible.

With the invariant method, the test harness checks if the kernel trees satisfy the RBT and BST invariants, and store the expected elements. To check the RBT invariant, the test harness uses the Isabelle/HOL "`rbt`" definition introduced in section 2.2. To check if all expected elements exist in the kernel RBT and that the tree satisfies the binary search tree invariant, it only has to compare the inorder of the kernel tree with that of the verified tree. From the same inorder, it follows that the kernel RBT also has to be a BST (see section 2.2 for the proof). If the kernel RBT does not satisfy all invariants and properties, the test harness prints which one(s) got violated.

3.2.5 Verifying the Kernel RBT Variant

As the Red-Black trees generated by the kernel differ from the default Isabelle/HOL RB-trees, I aimed to formalize the kernel variant in HOL and prove it correct. Within the time scope of this thesis, I succeeded in formalizing and proving the kernel insert operation in HOL by modifying the original Isabelle/HOL implementation and proof.

The kernel insert operation only differs from the original Isabelle one in how RB-trees are balanced and recolored. Modifying the "`baliL`" and "`baliR`" (balanceInsert) was enough to equalize the variants on the structural level. The following shows first the original "`baliL`" [8, HOL/Data_Structures/RBT.thy] and second my kernel variant. The "`baliR`" function equations are symmetric for both variants:

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 baliL :: "'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt" where
"baliL (R (R t₁ n t₂) p t₃) g (R t₄ u t₅) = R (B (R t₁ n t₂) p t₃) g (B t₄ u t₅)" |
"baliL (R t₁ p (R t₂ n t₃)) g (R t₄ u t₅) = R (B t₁ p (R t₂ n t₃)) g (B t₄ u t₅)" |
"baliL (R t₁ p (R t₂ n t₃)) g t₄ = B (R t₁ p t₂) n (R t₃ g t₄)" |
"baliL (R (R t₁ n t₂) p t₃) g t₄ = B (R t₁ n t₂) p (R t₃ g t₄)" |
"baliL t₁ a t₂ = B t₁ a t₂"
```

The proof for this alternative balance is based on the original Isabelle one [8, HOL/-Data_Structures/RBT_Set.thy, 13]. I only had to verify that the kernel variant preserves all color requirements because the rest of the proof for the insert function was still valid. The following shows the Red-Black tree color invariant that "baliL" and "baliR", and in the end, the overall "ins" function have to preserve:

```
fun invc :: "'a rbt ⇒ bool" where
"invc Leaf = True" |
"invc (Node l (a,c) r) =
  ((c = Red ⟶ color l = Black ∧ color r = Black) ∧ invc l ∧ invc r)"
```

The check if the root is black (which is a requirement for an RBT) does not happen in "invc" but in the "rbt" invariant because this simplifies verification.

One of the two critical parts was to find the weaker color invariant that holds while balancing the tree after a new element has been inserted:

```
fun invc2 :: "'a rbt ⇒ bool" where
"invc2 Leaf = True" |
"invc2 (Node l (_,c) r) =
  ((c = Red ⟶ color l = Black ∨ color r = Black) ∧ invc l ∧ invc r)"
```

This invariant is defined in exactly the same way as the stronger "invc," with the exception that if the root node is red, just one of the subtrees has to be black. For the subtrees themselves, the stronger "invc" must hold again.

With this modified weak invariant, the lemma from the original proof that "baliL" produces a correctly colored RBT—given the left tree is only weakly colored—holds again. The proof for "baliR" is symmetric:

```
lemma invc_baliL:
  "⟦ invc2 l; invc r ⟧ ⟹ invc (baliL l a r)"
by (induct l a r rule: baliL.induct) auto
```

For the subsequent proof, I needed another weak version of "invc," which was also used in the original Isabelle RBT proof. After painting the root black, "invc" must hold again:

```
abbreviation invc3 :: "'a rbt ⇒ bool" where
"invc3 t ≡ invc(paint Black t)"
```

In addition, I had to add the lemma that "invc" implies "invc3", similar to the existing "invc2I" lemma from the original proof:

```
lemma invc2I: "invc t ⟹ invc2 t"
  by (cases t rule: tree2_cases) simp+

lemma invc3I: "invc t ⟹ invc3 t"
  by (cases t rule: tree2_cases) simp+
```

For the final insert theorem, I needed a lemma that relates all three color invariants to inserting an element into a tree with a red root:

```
lemma ins_red_invcs:
  "⟦ invc t; invc3 (ins x t); color t = Red ⟧ ⟹ invc2 (ins x t)"
by (induct x t rule: ins.induct) auto
```

"ins" does not necessarily produce a black root node which is the reason why the lemma does not imply "invc (ins x t)". Though, as an RBT is required to have a black root node, the root is recolored afterward:

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

In the end, I added this lemma as a simplification rule to the original "invc_ins" lemma which states that after inserting a value into an properly colored RBT, "ins" returns a properly colored tree back (optionally painting a previous red root to black after insert):

```
lemma invc_ins: "invc t ⟹ invc3 (ins x t) ∧ (color t = Black ⟶ invc (ins x t))"
by (induct x t rule: ins.induct)
   (auto simp: invc_baliL invc_baliR invc2I invc3I ins_red_invcs)
```

This finishes the proof that "ins" preserves the color invariant. The rest of the RBT insert proof is equal to the original one.

4 Evaluation

To evaluate the correctness of the kernel RBT implementation, I ran the test harness with all three input generation strategies. It did not detect any malformed Red-Black trees. Table 4.1 summarizes the (total) executed and compared operations.

				Coverage %	
Strategy	Parameter	Operations	Time (min)	core	core+auxiliary
random	n=50,000	50,000	134	92.35	72.22
exhaustive	n=9	36,288,000	13	93.44	73.08
symbolic	see sec. 3.1.3	248	110	95.08	74.36
Total		36,338,248	257	99.45	77.78

Table 4.1: Overview of all successfully compared operations per test case generation strategy. For testing the symbolic strategy, I used the previously generated test cases from subsection 3.1.3. The time datum includes the symbolic execution with KLEE.

In comparison to random, one can execute and compare far more operations in less time with the exhaustive generation strategy. The reason for this is that the random RBTs can get very large and checking all invariants on the unverified RBTs takes nonlinear time. On the other hand, the trees in the exhaustive strategy do not get large. Therefore, the comparison time is negligible. In the case of this evaluation, the kernel RBT is reset to an empty tree after every 9th operation in the exhaustive strategy. The major drawback of exhaustive is that it exponentially grows. For higher n, execution takes significantly longer.

The table also lists the achieved line coverage per strategy in the kernel RBT that I previously lifted into userspace (see section 4.1 for the methodology). Column "`core+auxiliary`" describes the line coverage of the whole Red-Black tree file, including auxiliary functions that the test harness is not expected to test and use (like "`rb_first`", "`rb_next`", "`rb_last`", ...). Column "`core`" describes the coverage without these unused auxiliary functions. All strategies achieve an almost identical high line coverage, however, the exhaustive strategy achieves this in a fraction of time in comparison to the others. Random was the only strategy in this test that achieved a 100% coverage for the insert

and delete routine, but because it only calls the reset command once (after executing all other commands) it did not achieve a full coverage for the reset routine as well. Unifying the coverage data of all strategies leads to 99.45% for "`core`." In fact, just one line in the reset routine is not hit.

This high total line coverage and the millions of successful test runs increase the confidence that the kernel RBT implementation is correct and that the test pipeline built in this thesis successfully reached this goal of a higher assurance. It also suggests that testing with different strategies, as I did in this thesis, is a viable option that can help in covering many different cases.

4.1 Collecting Code Coverage

The kernel currently provides two ways to gather coverage information (see subsection 2.3.2). Unfortunately, none of them is suitable for this thesis. "gcov" collects coverage data for the whole kernel. Therefore, I cannot differentiate if a line in the Red-Black tree was executed by my kernel module or another kernel subsystem that also uses Red-Black trees. The same applies to "kcov." With kcov, one can specify a process whose system calls shall be used as a starting point to collect coverage data. The problem is that when my test harness issues a "`read`" or "`write`" system call, my kernel module that executes the RBT operations does not get immediately called. The kernel has to interpret the system call first and find the correct function to execute. In between these two steps, the kernel can use other RBTs to fulfill this task. These RBT operations are also counted into coverage, even though I am only interested in the RBT code coverage initiated by my kernel module. For this exact same reason, the kernel developers disabled the kcov coverage collection for the RBT implementation [11][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
```

However, I can use my userspace kernel RBT implementation and analyze its code coverage in userspace using gcov. This will not give the same results as a coverage analysis in kernelspace because I had to modify the lifted implementation to compile it in userspace. Nevertheless, it gives a good overview of what is (not) tested and an acceptable way to compare the different strategies.

In order to pipe the "`random`" and "`exhaustive`" test cases into the userspace RBT, I had to extend the test harness with a function to print them out rather than executing

them. By adding a simple parser to the existing userspace RBT wrapper, these test cases could then be executed in the userspace RBT and gcov is able to simultaneously collect coverage information. For they symbolic strategy, the wrapper does not need any changes because it was initially built exactly for this strategy (see subsection 3.1.3).

4.2 Testing the Test Harness

Because the test harness did not report any error, it is important to test if the error detection works as expected. My initial approach was to introduce a bug into the kernel RBT implementation, such as removing one required tree rotation and checking if any testing strategy can find the error. Moreover, this would allow me to measure the effectiveness of the different strategies. Interestingly, however, whenever I introduced an error into the insert or delete routine that only violates the Red-Black tree invariants but does not alter the stored elements, the kernel crashed while booting ("`NULL pointer dereference`"). The error happens in the memory subsystem that uses augmented Red-Black trees, which in turn expect from the core Red-Black tree implementation to behave correctly. Therefore, I decided not to add an error into the kernel RBT implementation but in my custom kernel module used as an interface between the kernel RBT and the test harness. After encountering the second insert operation, the kernel module maliciously paints the root node red. This procedure does not help in comparing the effectiveness of the different test case generation strategies to find an error, but at least it allows me to check the correctness of the test harness to some extent. For the first failed test case, command "`# rbtTestHarness exhaustive -n 2`" returns the following error message:

```
After inserting 2 following invariants are violated: color, root_black
Kernel tree before:  Node Leaf (1,Black) Leaf
Kernel tree after:   Node Leaf (1,Red) (Node Leaf (2,Red) Leaf)
Verified tree after: Node Leaf (1,Black) (Node Leaf (2,Red) Leaf)
```

The test harness lists as expected the violated invariants, the kernel trees before and after executing the operation, and the verified RBT after executing the operation on the verified implementation. Thus, the error detection by the test harness was successful.

5 Related work

Since Linux version 3.7 (released in December 2012), the kernel provides its own testing module for its Red-Black tree implementation [11][lib/rbtree_test.c]. KUnit, introduced in subsection 2.3.2, is a unit testing framework for the Linux kernel and was released in January 2020 with Linux version 5.5 [11]. The in-kernel RBT tests have not been ported to KUnit yet and instead use a kernel module that, when loaded, instantly executes a test suite, prints the test results and unloads itself after it.

The module performs a set of insert operations and deletes these elements again afterward. In between every operation, it checks all Red-Black tree invariants. Moreover, it measures the latency of these operations.

In comparison to my approach, the in-kernel module tests only a single test case that manipulates the tree—inserting a previously defined number of nodes and removing all of them afterward again. Thus, the in-kernel module cannot find potential bugs caused by a different ordering of insertion and deletion. Moreover, the in-kernel test module does not use and test the new helper functions introduced in kernel version 5.12 for RBTs (as my module does) but uses unverified RBT helper functions to check the correctness of the RBT implementation itself. Lastly, the in-kernel testing module does not support a structural comparison, while my testing pipeline supports it at least for insert operations. Directly comparing the structure of the kernel RBT against the verified RBT takes only linear time and makes tests with larger trees efficient.

The major advantage of the in-kernel module is that it also tests the auxiliary functions used to build augmented RBTs; and that Haskell is not a requirement to compile the test suite.

Kernel verification

To the best of my knowledge, this is the first attempt to test a Linux data structure against a formally verified one. Nevertheless, formal verification is not a novel topic in kernel research. Liang et al. verified, using a model checker, a significant part of the Linux RCU implementation, a high-performance synchronization mechanism used throughout the kernel, including in the Red-Black tree implementation [10]. Efremov et al. deductively verified 23 memory- and string-related kernel library functions like "`memcpy`" and "`strcmp`" [4]. Zakharov et al. developed a set of Linux driver verification

tools that allow to create a set of specification rules for the kernel core API and verify their correct usage by drivers.

6 Conclusion

In this thesis, I executed and verified the correctness of over 30 million kernel Red-Black tree operations by comparing their results against verified ones. For this, I implemented and evaluated different testing strategies, including translating kernel C code to Isabelle/HOL and proving it correct. In the end, this effort constructed a high assurance of a functionally correct Linux Red-Black tree algorithm.

6.1 Similar Test Procedures for other Kernel Subsystems

The kernel Red-Black tree implementation has existed for more than two decades and is an integral part of the kernel. For a data structure implementation like this, the time of existence without known bugs is in itself a high assurance of correctness. Nevertheless, the kernel RBT implementation underwent some modifications in this long period. A testing pipeline like the one presented in this thesis can significantly help in assuring that a modification or a newly implemented data structure is functionally correct. If the invariants of a data structure are not as ambiguous as the RBTs are, directly comparing the results of a kernel implementation against the results of a verified one is most often a simple task, especially when generating the test cases randomly.

For kernel code other than data structures, a verified variant or at least a formal specification hardly ever exists; and in connection with the testing procedure of this thesis, it would not be worth creating one in my opinion. Modeling and proving complex kernel code is much more demanding than it is for most data structures. Moreover, the Linux kernel changes at a high rate, and as long as the Linux developers do not want to maintain formal models and proofs, they will get obsolete quickly.

6.2 Future Work

In this thesis, I concentrated on the correctness of the core Red-Black tree algorithm. However, Linux also supports concurrent access and modification of Red-Black trees using the RCU synchronization mechanism. It is an open question of how to extend the testing pipeline of this thesis to assure the correctness of RBTs in a concurrent context with high confidence.

Moreover, the Linux RBT insert to Isabelle/HOL verification leads to the task of also formalizing the kernel RBT delete variant in HOL and proving it correct. This would allow comparing all RBT operations efficiently on a structural level.

List of Figures

List of Tables

Bibliography

[1] L. Bulwahn and M. Polat. *rbtree: remove unneeded explicit alignment in struct rb_node*. URL: https://lore.kernel.org/lkml/YQ1ToK8EMdAO4CyH@precision/ (visited on 08/11/2021).

[2] C. Cadar, D. Dunbar, D. R. Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[3] V. Chipounov, V. Kuznetsov, and G. Candea. "S2E: A platform for in-vivo multi-path analysis of software systems." In: *Acm Sigplan Notices* 46.3 (2011), pp. 265–278.

[4] D. Efremov, M. Mandrykin, and A. Khoroshilov. "Deductive Verification of Unmodified Linux Kernel Library Functions." In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2018, pp. 216–234. ISBN: 978-3-030-03421-4.

[5] L. J. Guibas and R. Sedgewick. "A dichromatic framework for balanced trees." In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3.

[6] F. Haftmann and L. Bulwahn. *Code generation from Isabelle/HOL theories*. 2013. URL: http://isabelle.in.tum.de/dist/Isabelle2021/doc/codegen.pdf (visited on 06/28/2021).

[7] *Introduction to fbl intrusive containers*. URL: https://fuchsia.dev/fuchsia-src/development/languages/c-cpp/fbl_containers_guide/introduction (visited on 06/29/2021).

[8] *Isabelle*. URL: https://isabelle.in.tum.de/ (visited on 06/28/2021).

[9] *KLEE source code repository*. URL: https://github.com/klee/klee (visited on 07/22/2021).

[10] L. Liang, P. E. McKenney, D. Kroening, and T. Melham. "Verification of tree-based hierarchical read-copy update in the Linux kernel." In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 61–66. DOI: 10.23919/DATE.2018.8341980.

[11] *Linux Kernel 5.12 source code*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v5.12.

[12] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[13] T. Nipkow, J. Blanchette, M. Eberl, A. Gómez-Londoño, P. Lammich, C. Sternagel, S. Wimmer, and B. Zhan. *Functional Algorithms, Verified!* 2021.

[14] R. Sedgewick. *Left-Leaning Red-Black Trees*. URL: https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf (visited on 06/25/2021).

[15] *The Linux Foundation*. URL: https://linuxfoundation.org/ (visited on 08/11/2021).

[16] *The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/v5.13 (visited on 06/29/2021).

[17] D. Vyukov. *kernel: add kcov code coverage*. URL: https://lore.kernel.org/lkml/1452615352-117784-1-git-send-email-dvyukov@google.com/ (visited on 07/26/2021).

[18] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov. "Configurable toolset for static verification of operating systems kernel modules." In: *Programming and Computer Software* 41.1 (2015), pp. 49–64.