Deductive Verification of a Patricia Trie with Creusot

Téo Bernier[1][0009−0003−4834−7126], Frédéric Loulergue[2][0000−0001−9301−7829], and Nikolai Kosmatov[1][0000−0003−1557−2813]

[1] Thales Research and Technology, cortAIx Labs, Palaiseau, France
{teo.bernier,nikolai.kosmatov}@thalesgroup.com
[2] Université d'Orléans, INSA Centre Val de Loire, LIFO UR 4022, Orléans, France
{teo.bernier,frederic.loulergue}@univ-orleans.fr

Abstract. Beyond their practical value, space-efficient mutable containers provide instructive use cases for deductive verification, in particular for Rust implementations, where ownership and borrowing simplify some aliasing concerns without eliminating the need for functional proofs. This paper reports on a verification case study of a Patricia trie implemented in safe Rust, specified and verified with Creusot. The code targets a concrete setting: keys are fixed-length arrays of Boolean values, and leaf contents are generic. The verified API is a finite-map interface comprising empty trie construction, mutation-based insertion and deletion, as well as lookup. The case study exposes the main proof challenges of compressed prefix trees: destructive updates, splitting at the first differing bit, and maintaining a global correspondence between explicit node prefixes and the extensional key–value relation represented by the trie. We prove functional correctness of lookup and updates, extensional properties such as the uniqueness of the value associated with a key, and client scenarios that validate the contracts on both symbolic and concrete executions.

1 Introduction

Beyond their practical value, mutable data structures provide an important source of case studies for formal methods. In particular, they expose the interaction between complex memory behaviors, logical abstractions, and the limits of proof automation [42,25,36,19,39,38,8,9,22,31]. They are used as building blocks in numerous service modules and applications, and therefore the correctness of their implementations is essential.

Compared to unsafe memory languages like C/C++, Rust brings a strong ownership discipline and eliminates large classes of memory errors. Yet, deductive verification is still needed to establish correct functional behavior of low-level containers. Recent work on Rust verification has explored a variety of approaches, ranging from semantic foundations to automated and auto-active verification tools [27,1,26,33,24]. In this landscape, Creusot [18,21,2] appears particularly attractive because it covers a large subset of Rust features [17] and relies on Why3 [11] to generate proof obligations (or verification conditions), which enables them to be proved by multiple solvers including Alt-Ergo [14], Z3 [16] and cvc5 [4].

Among various mutable data structures, Patricia tries [23,35] provide an interesting target for formal verification. They offer fast prefix lookup and a compact representation of bitstring keys. From a practical point of view, they are used, for example, in Linux kernel routing, in the Haskell maps library, which is extensively used in GHC, the Glasgow Haskell Compiler, in Ethereum (Merkle Patricia Tries), and for runtime support for dynamic verification in the E-ACSL plugin [28] of the Frama-C industrial analysis platform. The latter is implemented in C and initially motivated our choice for this specific structure since the correctness of the analysis depends on the correctness of the Patricia trie implementation.

These examples illustrate their importance for industrial applications. From a verification perspective, Patricia tries combine several specification and proof difficulties: the specification has to encode the *common prefix logic* and the *key-based structural invariants*, and the proof must show that internal nodes always store *compressed prefix* information, updates perform *structural rewiring* correctly, and the intended semantics is that of a *finite map* from keys to values.

This paper reports on a deductive verification case study of a Patricia trie written in *safe* Rust and annotated for Creusot. The companion artifact [6] provides the considered implementation of a Patricia trie, where keys are represented by arrays (of fixed size) of Booleans and values can be of any type. The verified interface is a small map-like API with constructors, lookup, insertion, and removal. We estimate the overall effort required to perform this case study as approximately 6 person-months, which included learning the Creusot tool and proving several simpler versions first. The full verification can be replayed using the artifact Dockerfile, which relies on Creusot version 0.11.0, Why3, Alt-Ergo, Z3 and cvc5.

Contributions. The contributions of this case study include:

- formal verification using Creusot of a Patricia trie implementation in Rust with a finite-map-style API;
- a specification based on an abstract key–value relation and an explicit prefix-classification data type that bridges compressed branching and extensional map reasoning;
- a presentation of the verification methodology combining logical/executable pairings, bit-level lemmas, snapshots, and localized assertions to prove destructive updates in Creusot;
- a discussion of the current scope and limitations.

Outline. The remainder of the paper is structured as follows. Section 2 recalls the Rust and Creusot notions needed in the following sections. Section 3 presents the target data structure, Patricia tries. Section 4 describes the verified API and its specification. Section 5 explains the verification approach and the main proof results. Section 6 discusses related work, and Section 7 concludes and gives future research directions.

2 Background: Rust and Creusot

Since its introduction, the Rust programming language has attracted significant attention. It is designed as a low-level systems language, offering an alternative to traditional languages such as C and C++, while providing stronger safety guarantees and richer abstractions without incurring performance penalties. In particular, Rust offers a variety of high-level features, including ownership and borrowing, algebraic data types and pattern matching, generic programming, traits and dynamic dispatch, as well as support for iterators, combinators, and higher-order functions.

These features, together with the goal of ensuring memory safety without relying on a garbage collector, are enabled by integrating underlying concepts such as linearity [34,3], ownership [12] and borrowing [37] directly into the type system. By encoding resource management within types, Rust enables static tracking of variable usage and memory access patterns. This approach allows the compiler to automatically free memory at appropriate points in the program, thereby achieving safe and efficient memory management without garbage-collection overhead.

Creusot [18] is a deductive verifier for Rust that targets ordinary source code annotated with contracts, logical functions, and proof-oriented assertions. From a practical perspective, its main advantage is that verification remains close to the implementation: the verification engineer works directly in Rust, without translating the code into a separate proof language in a proof assistant.

Creusot translates annotated Rust programs to the intermediate verification framework Why3 [11], where verification conditions are discharged by SMT solvers. Its specification language, Pearlite, supports first-order reasoning over contracts, algebraic data types, and ghost code, while snapshots and proof assertions provide explicit control at mutation points. Creusot follows an auto-active methodology: most proof obligations are generated automatically, but the user can still guide the provers with auxiliary lemmas and local assertions.

Our target implementation of a Patricia trie stays within a tame fragment of the Rust language: it uses mutable references and heap allocation through `Box`, but no *unsafe* code. Yet, its correctness depends on nontrivial semantic properties: prefix-based routing, extensional map behavior, and preservation of invariants under destructive updates. Creusot is expressive enough to state these properties at the source-code level, yet lightweight enough to keep the proof architecture tied to the actual code.

3 Verification Target: Patricia Tries

A binary trie stores keys as root-to-leaf paths whose edges are labeled by bits. If many keys share a long prefix, an ordinary trie contains a chain of unary internal nodes before the first genuine branch. A *Patricia trie* (also known as a *radix tree* or *compressed trie*) is a variant of the trie data structure in which chains of unary nodes are compressed into single edges, yielding a more space-efficient representation [23,35]. Thus, each internal node corresponds to the longest common prefix shared by the keys in its subtree. Figure 1 illustrates this compression.

Fig. 1. Compression in a Patricia trie. The target implementation stores at each internal node the corresponding prefix: the longest common prefix of the keys in its subtree.

The verified implementation uses an explicit-prefix view, where each internal node stores the longest prefix shared by all keys in the corresponding subtree. The main exported structure is `PTrie<K,V>`, implemented as a wrapper around an optional root node, `Option<PNode<K,V>>`. The recursive data type `PNode<K,V>`, which remains internal to the implementation, has three constructors: a branching node `Node{prefix,left,right}`, a leaf `Leaf{key,value}`, and a transient constructor `Tmp`, which is used during updates. Keys are abstracted by the trait `Key`; the artifact provides, in particular, an array implementation `AKey([bool; KEY_SIZE])` with `KEY_SIZE = 32` and a machine-word implementation `UKey(usize)`. A prefix is represented as a full key together with a length indicating how many leading bits are significant. The remaining bits are ignored. The values stored in the PTrie have type parameter `V`.

Lookup starts at the root and repeatedly compares the given key with the prefix stored in the current internal node. If the key extends it, the next bit determines whether the search continues on the left or on the right. If the key diverges before the end of the prefix, the search stops immediately.

Insertion follows prefixes until the first mismatch with the given key, and then rewrites the tree so that the mismatch becomes a new branching point. Insertion modifies the PTrie and returns an optional value. At the top level, if the PTrie is empty, insertion creates a single leaf root and returns `None`. Otherwise, it delegates to recursive node insertion. In a leaf, if the inserted key is equal to the stored key, the value is updated in place and the old value is returned. If the keys differ, insertion computes their longest common prefix, replaces the current leaf by a branching node, and places the old and new leaves on the left or right according to the first differing bit. In an internal node, insertion compares the new key against the stored prefix. If the key still belongs below the node, insertion recurses into the designated child; if it diverges before the end of the current prefix, insertion creates a fresh parent node above the current subtree.

Deletion of a given key works by following the compressed prefixes until the algorithm can decide one of three cases: the key is absent in the current subtree, the key is deeper in one subtree, or the key is stored in an immediate leaf. At each internal node, the algorithm compares the query key with the prefix stored

in that node. If the query disagrees with that prefix before the prefix ends, then the key cannot occur anywhere in that subtree, so deletion stops immediately and returns failure. If the query key agrees with the stored prefix, then the next relevant bit determines whether the algorithm continues in the left subtree or in the right subtree. Once the algorithm has chosen a side, two cases may occur. If the selected child is itself an internal node, deletion continues recursively there. If the selected child is a leaf, the algorithm compares the full key stored in that leaf with the query key. In case they differ, the key is not present and deletion fails; otherwise, the leaf is removed.

The Patricia-trie-specific step occurs after a successful removal of such a leaf. Since the current internal node then has only one remaining child, the algorithm immediately compresses the Patricia trie by removing that internal node as well and replacing it with the surviving sibling. In other words, deletion not only removes the key, but also restores the compressed Patricia shape instead of leaving a useless unary branch.

At the top level, the same principle applies to the root: if the PTrie is empty, deletion fails. If the PTrie consists of a single leaf, deletion either removes that leaf or leaves the PTrie unchanged. Otherwise, deletion proceeds with the recursive process described above.

A Representative Scenario from the Artifact. To validate the expected results, we exercise the verified API on various sequences of operations on a PTrie. In function `main_akey` of the companion artifact [6], this is done concretely on a fresh PTrie using four distinct 32-bit keys:

$$a = 0^{32}, \quad b = 0000\,1100\,1\,0^{23}, \quad c = 0000\,1000\,1\,0^{23}, \quad d = 1^{32},$$

where 0^k and 1^k denote runs of k identical bits. These keys induce the compact Patricia trie shown in Fig. 2. This example is useful throughout the case study because it exhibits two characteristic update patterns of the implementation: a local split between b and c below the prefix 00001, and a top-level split separating the mostly-zero family $\{a, b, c\}$ from the all-one key d.

The generic symbolic client function `generic_symbolic_client` checks the same sequence of insertions, updates, lookups, and removals using four arbitrary distinct keys on an arbitrary PTrie. The function `main_ukey` instantiates the same client for machine-word keys.

4 API Specification

The contracts shown in this section are written in Pearlite, the logical specification language of Creusot. API preconditions and postconditions are given by `#[requires(...)]` and `#[ensures(...)]` attributes, respectively. Functions annotated with `#[logic]` are logical functions: they are used in specifications and proofs, but are not part of the executable API. In the code, we systematically use the suffix `_log` for logical functions, pretty-printed as a lower subscript "$_l$" in the paper. The attribute `#[check(terminates)]` asks Creusot to prove termination of the corresponding executable function.

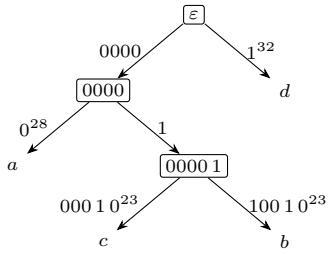

Fig. 2. Patricia trie storing keys $a = 0^{32}$, $b = 0000\,1100\,1\,0^{23}$, $c = 0000\,1000\,1\,0^{23}$, $d = 1^{32}$. This trie is used in the concrete client scenario implemented by `main_akey` in the companion artifact [6]. Internal nodes are labeled by the corresponding prefixes.

Pearlite formulas reuse a Rust-like expression syntax, extended with logical constructs such as universal and existential quantification, written for instance `forall<var: Type>` and `exists<var: Type>`. They also introduce new logic operators used in the contracts below. For an expression `e`, `e@` denotes its logical view, as provided by the `View` trait, for example, the view of the `usize` type is the logical `Int` type. For an expression denoting a mutable borrow, `^e` denotes its final, or prophesied, value, that is, the value it will have when the borrow is released. Finally, `==>` denotes logical implication. These notations are used only in specifications and proofs, and do not affect the executable behavior of the verified API.

With this notation in place, we now describe the public specification of the Patricia trie. The API has two main components: the key abstraction `Key` and the Patricia-trie structure `PTrie<K: Key, V>`.

`Key` extends both `Copy` and `View`. In the case of keys, the corresponding logic model is a sequence of bits (i.e. `Seq<bool>`) . The `Key` trait requires both logical and executable functions for: the length of a key (`len`), getting the i^{th} bit of a key (`bit(i)`), and getting the index of the first different bit of two keys (`diff(key)`). The trait also requires two lemmas that establish the equivalence of model equality and extensional bitwise equality of two keys.

We provide two implementations of this trait: one for fixed-size Boolean arrays and one for unsigned integers.

The central abstraction of the specification is the predicate:

$$\texttt{ptrie.contains}_l\texttt{(k, v)}$$

which states that the PTrie stores the association of key k with value v.

The logical function $\texttt{get}_l$ is specified as the functional view of $\texttt{contains}_l$: it returns `Some(v)` exactly when the predicate holds for `v`, and `None` otherwise. At this level, the PTrie is understood extensionally as a partial map.

Figure 3 presents the first two verified functions of the API. Lines 1118–1123 define $\texttt{is\_empty}_l$: `#[logic]` indicates it is a logic function whose purpose is to be called in contracts or logical annotations. It is used on Lines 1126 and 1136 in the contracts of the executable function `is_empty` and the constructor of empty PTrie, `new`.

```
1118  #[logic(open)]
1119  pub fn is_empty_l(&self) -> bool {
1120    pearlite! {
1121        forall<key: &K, value: V> !self.contains_l(key, value)
1122    }
1123  }
1124
1125  #[requires(self.inv_l())]
1126  #[ensures(result == self.is_empty_l())]
1127  pub fn is_empty(&self) -> bool
1134
1135  #[ensures(result.inv_l())]
1136  #[ensures(result.is_empty_l())]
1137  pub fn new() -> Self
```

Fig. 3. Verified PTrie API: vacuity test and creation.

```
1223  #[requires(self.inv_l())]
1224  #[ensures(forall<value: &V> result == Some(value) ==>
1225                          self.get_l(key) == Some(*value))]
1226  #[ensures(result == None ==> self.get_l(key) == None)]
1227  #[ensures(forall<value: &V>
1228    self.contains_l(key, *value) == (result == Some(value)))]
1229  #[ensures(!(exists<other_value: V>
1230    self.contains_l(key, other_value)) == (result == None))]
1231  pub fn get(&self, key: &K) -> Option<&V>
```

Fig. 4. Verified PTrie API: lookup.

Note that both operations refer to $\texttt{inv}_l$. This predicate expresses formally conditions on the `PTrie` structure so that it has the properties of a Patricia trie, in particular that the routing using bits is correct. This invariant is detailed in the next section because it is a key element of the verification.

Figure 4 shows the contract for the lookup function. Either the PTrie contains a value associated with the given key and a reference to this value is returned (Lines 1224-1225), or there is no value associated with this key and `None` is returned (Line 1226). It is also expressed using $\texttt{contains}_l$. This can only be ensured for well-formed tries respecting the invariant (Line 1223).

There is a variant when one is interested in a mutable reference to the searched value: `get_mut`, shown in Fig. 5. The Patricia trie itself is borrowed mutably. Creusot models a mutable borrow as carrying two logical views of the borrowed place: its current value, written `*r`, and its final prophesied value, written `^r`. For `get_mut`, the returned `value: &mut V` escapes to the caller, so `^value` is not fixed at the moment `get_mut` returns. It denotes the value that the mutable reference will have when the caller eventually stops using it, once it has reached its final value. This is why `*value == ^value` is not a valid post-condition: it would claim that no future write through the returned reference can change the value of the reference. The situation for `^self` is related: because `value` is a mutable borrow into `self`, the final view `^self` must account for the future final value of that borrowed subpart. So `^self` is not merely "the trie at the instant of return" either. It is the trie after the outstanding mutable borrow has been resolved, with the borrowed entry updated according to `^value`. The invariant is preserved (Line 1240). If the function returns a reference then its value is related to the key in the pre-state and its prophesied value is also re-

```
1239  #[requires(self.inv_l())]
1240  #[ensures((^self).inv_l())]
1241  #[ensures(forall<value: &mut V> result == Some(value) ==>
1242      self.get_l(key) == Some(*value) &&
1243      (^self).get_l(key) == Some(^value))]
1244  #[ensures(result == None ==>
1245        self.get_l(key) == None &&
1246        (^self).get_l(key) == None)]
1247  #[ensures(forall<other_key: &K, other_value: V>
1248      key@ != other_key@ ==>
1249      ((^self).contains_l(other_key, other_value) ==
1250      self.contains_l(other_key, other_value)))]
1251  pub fn get_mut(&mut self, key: &K) -> Option<&mut V>
```

Fig. 5. Verified PTrie API: lookup for mutable reference.

```
1276  #[requires(self.inv_l())]
1277  #[ensures((^self).inv_l())]
1278  #[ensures((^self).contains_l(&key, value))]
1279  #[ensures(forall<other_key: K, other_value: V>
1280    key@ != other_key@ ==>
1281      ((^self).contains_l(&other_key, other_value) ==
1282        self.contains_l(&other_key, other_value)))]
1283  #[ensures(forall<other_value: V>
1284    self.contains_l(&key, other_value) ==
1285    (result == Some(other_value)))]
1286  #[ensures(!(exists<other_value: V>
1287    self.contains_l(&key, other_value)) == (result == None))]
1288  #[check(terminates)]
1289  pub fn insert(&mut self, key: K, value: V) -> Option<V>
```

Fig. 6. Verified PTrie API: insertion.

lated to the same key (Lines 1241-1243), otherwise it is absent from both (Lines 1244-1246). For all the other keys, the pre-state and prophesied state for the Patricia trie are equivalent in the sense for a given key, either they contain the same value or they both do not associate a value to this key (Lines 1247-1250).

Insertion (Fig. 6) preserves the invariant (Line 1277), adds the new binding, i.e. it is contained in the trie at the end of the call (Line 1278), preserves all associations for all keys logically different from the given key (Lines 1279-1282), and returns the previous value if one existed (Lines 1283-1285) or returns `None` otherwise (Lines 1286-1287).

Removal is specified symmetrically: it preserves the invariant (see Fig. 7, Line 1304), removes the queried binding if present (Line 1306), leaves all values of other keys untouched (Lines 1307-1310), and returns the deleted value if the key was previously stored, or `None` if it was not (Lines 1311-1314).

5 Verification Approach and Results

The verification follows the implementation layers, where each layer introduces the logical facts needed by the next one. The key layer provides the bit-level view of keys, the prefix layer specifies the relation between keys and prefixes, the node layer specifies and proves the recursive algorithms, and the wrapper layer exposes the public finite-map API.

The first layer is the key abstraction described previously.

The second layer is the prefix layer, centred on the `APrefix<K>` type. A value `APrefix(k,n)` represents the prefix of length `n` of the key `k`. Bits of `k` at positions

```
#[requires(self.inv_l())]
#[ensures((^self).inv_l())]
#[ensures(forall<value: V> !(^self).contains_l(key, value))]
#[ensures(forall<other_key: K, other_value: V>
  key@ != other_key@ ==>
    ((^self).contains_l(&other_key, other_value) ==
    self.contains_l(&other_key, other_value)))]
#[ensures(forall<value: V>
  self.contains_l(key, value) == (result == Some(value)))]
#[ensures(!(exists<value: V>
  self.contains_l(key, value)) == (result == None))]
pub fn remove(&mut self, key: &K) -> Option<V>
```

Fig. 7. Verified PTrie API: removal.

Layer	Main proof facts	Role
Keys	`lemma_eq_bits`, `lemma_bits_eq`	Relate equality of logical key views to equality of all bits.
Prefixes	`lemma_kp_eq_cmp_none`$_l$, `same_view_cmp_none`$_l$, `lemma_same_view_cmp_none`$_l$	Recover the branch-classification facts needed when a key matches a prefix.
Nodes	`lemma_leaf_contains_exact`, `lemma_node_always_contains_at_least_one`, `lemma_only_one_value_per_key`, `lemma_contains_key_eq`	Establish subtree-level properties of `contains`$_l$: exact leaf membership, non-emptiness, determinism, and key-view extensionality.
Wrapper	`lemma_empty_is_none`, `lemma_only_one_value_per_key`, `lemma_contains_key_eq`	Lift node-level properties to the public `PTrie` interface.

Fig. 8. Overview of lemmas

greater than or equal to `n` are ignored. Thus, several values of type `APrefix<K>` can represent the same prefix, as long as their keys match on their first `n` bits. The invariant of `APrefix<K>` requires `n` to be strictly smaller than the key length, which ensures that there is always a next bit after the prefix.

The comparison operation on prefixes is central to the implementation. Comparing a key with a prefix returns a value of type `KPCmp<K>`. In a tuple `KPCmp(is_right,opt_prefix)`, `opt_prefix` indicates whether the key matches the prefix or diverges from it. If the key matches the prefix up to its length, then `opt_prefix` is `None` and `is_right` is the first bit of the key after the prefix, and determines the child to follow. If instead the key diverges before the end of the compared prefix, then `opt_prefix` is `Some(prefix)` where `prefix` is the longest common prefix of the key and the compared prefix. In this case, `is_right` is the bit of the key at the divergence position, and is used when insertion creates a new branching node. In both cases, `is_right` indicates whether the key should be routed to the left or to the right branch, with `false` for the left branch and `true` for the right branch.

The third layer is the internal tree `PNode<K,V>`. This is where the recursive algorithms for lookup, mutable lookup, insertion, and deletion are verified. Its constructors are described in Sect. 3. The public wrapper `PTrie<K,V>` forms the fourth layer: it handles the empty-root case and exposes the verified API described in Sect. 4.

```
807   #[logic]
808   #[requires(self.inv_l())]
809   #[ensures(exists<key: &K, value: V> self.contains_l(key, value))]
810   pub fn lemma_node_always_contains_at_least_one_element(&self) {
811     match self {
812       Node { left, .. } => left.lemma_node_always_contains_at_least_one_element(),
813       Leaf { key, value } => {
814         proof_assert!(self.contains_l(key, *value));
815       }
816       _ => (),
817     }
818   }
```

Fig. 9. Definition and proof of `lemma_node_always_contains_at_least_one_element`

Invariant. The main invariant is defined first on internal nodes and then lifted to the wrapper. For nodes, $\texttt{inv}_l$ is the conjunction of two predicates. Predicate $\texttt{only\_nodes\_and\_leaves}_l$ excludes the transient constructor `Tmp` from stable states. This constructor is introduced only by destructive updates. Predicate $\texttt{prefix\_includes}_l$ states the routing principle of Patricia tries. For every binding contained in a node, the stored prefix must classify the key consistently: a key routed left can occur only in the left subtree, a key routed right can occur only in the right subtree, and a key that does not match the stored prefix cannot occur below the node at all. The wrapper invariant $\texttt{PTrie::inv}_l$ applies the same node invariant to the optional root, with the empty trie represented by `None`.

Proof-guiding constructs. Besides contracts, the proof can be guided using two constructs provided by Creusot. Macro `snapshot!(e)` records the logical value of expression `e` at a given program point. This is particularly useful before destructive updates, it allows to reason about previous states, but it can also be used to add specific lemmas into the proof context. Macro `proof_assert!(P)` introduces an intermediate assertion: Creusot generates a verification condition for proposition `P`, and once it is proved, the proposition is available to prove the following obligations. These constructs do not affect the executable Patricia-trie operations.

Auxiliary lemmas. The verification relies on a few lemmas, summarized in Fig. 8. These lemmas are defined and proved using lemma functions. As already mentioned, the key lemmas connect equality of logical views with pointwise equality of bits. Prefix lemmas explain the case where a key has the same logical view as the key used to build a prefix, which is needed when the prover must recover that the prefix comparison returns the `None` case. Node lemmas establish elementary facts about the represented relation, in particular exact membership for a leaf, non-emptiness of well-formed nodes, functionality of the map relation, and extensionality with respect to key views. Wrapper lemmas lift these facts from nodes to `PTrie`.

Figure 9 shows one example of a lemma function. The precondition on Line 808 requires the node to be well formed. The postcondition on Line 809 states that such a node contains at least one key-value pair. The proof on Lines 811–817 follows the recursive structure of the `PNode` type: for an internal node, it is enough

```
1003 impl<K: Key, V> PNode<K, V> {
1008   #[requires(self.inv_l())]
1009   #[requires(match self {Node {..} => true, _ => false})]
1010   #[ensures((^self).inv_l())]
1011   #[ensures(forall<value: V> !(^self).contains_l(key, value))]
1012   #[ensures(forall<other_key: K, other_value: V> key@ != other_key@ ==>
1013     ((^self).contains_l(&other_key, other_value) == self.contains_l(&other_key, other_value)))]
1014   #[ensures(forall<value: V> self.contains_l(key, value) == (result == Some(value)))]
1015   #[ensures(!(exists<value: V> self.contains_l(key, value)) == (result == None))]
1016   #[check(terminates)]
1017   #[variant(self.height())]
1018   pub fn remove(&mut self, key: &K) -> Option<V> {
1019     if let Node { prefix, left, right } = self {
1020       snapshot! { key.lemma_eq_bits(key) };
1021       match prefix.kp_cmp(key) {
1022         KPCmp(false, None) => match &mut **left {
1023           Leaf { key: leaf_k, value: leaf_v } => {
1024             let diff = leaf_k.diff(key);
1025             if diff == K::len() {
1026               snapshot!{left.lemma_leaf_contains_exact(key, *leaf_v)};
1027               proof_assert!(left.contains_l(key, *leaf_v));
1028               let removed_leaf = left.take();
1029               let Leaf { value: leaf_v, .. } = removed_leaf else {
1030                  unreachable!()
1031               };
1032               *self = right.take();
1033               Some(leaf_v)
1034             } else {
1035               None
1036             }
1037           }
1038           _ => left.remove(key),
1039         },
1040         // ...
```

Fig. 10. Deletion on internal Patricia-trie nodes.

to recurse on one child (Line 812), for a leaf, the stored key-value pair is a direct witness of the existential quantifier. However, a `proof_assert!` (Line 814) is still needed to make this fact explicit to the SMT solvers. This proof illustrates a common proof pattern used in the node layer, several other lemmas follow the same style: they perform case analysis, proceed by recursive calls when needed, and expose intermediate facts through local assertions.

One role of lemma functions is to state facts that SMT solvers do not derive directly from recursive definitions. This is particularly visible in the update operations, where the proof must relate the pre-state of a mutable node to the rebuilt post-state.

Another role is to isolate complex facts in smaller proof contexts with carefully chosen hypotheses, which can make the resulting verification conditions easier for SMT solvers to discharge than in more saturated contexts.

A representative proof. Deletion is the most characteristic update proof because it combines prefix routing, ownership-preserving mutation, and compression of the tree shape. Figure 10 shows the core recursive operation on `PNode`.

The proof follows the algorithmic structure. The current prefix first classifies the queried key (Fig. 10, Line 1021). If the key belongs to the left branch, the proof inspects the left child (Line 1022). If this child is a leaf with the same key view, the leaf is removed and the current node is replaced by the right sibling

```
1315  pub fn remove(&mut self, key: &K) -> Option<V>
1316  {
1317    let root = self.root.take();
1318    match root {
1319      Some(node) => match node {
1320        Leaf {
1321          key: leaf_key,
1322          value: leaf_value,
1323        } => {
1324          let diff = leaf_key.diff(key);
1325          if diff == K::len() {
1326            Some(leaf_value)
1327          } else {
1328            self.root = Some(Leaf {
1329              key: leaf_key,
1330              value: leaf_value,
1331            });
1332            None
1333          }
1334        }
1335        mut node => {
1336          let res = node.remove(key);
1337          self.root = Some(node);
1338          res
1339        }
1340      },
1341      None => None,
1342    }
1343  }
```

Fig. 11. Deletion on a Patricia trie

(Lines 1023–1032). This is the Patricia-specific compression step: deletion does not leave a unary internal node behind. The right branch is symmetric.

The proof annotations in this code are concentrated at the points where the shape changes. The call to `take` moves a child or the current node out of place while leaving `Tmp` (Lines 1028 and 1032). The post-state invariant excludes this temporary value once reconstruction is complete. The snapshot of `lemma_leaf_contains_exact` records that the selected leaf denotes exactly one binding before it is moved (Line 1026). The local `proof_assert!` steps make explicit the membership and absence facts needed by the post-condition after the sibling has been promoted (Line 1027).

At the wrapper level, `PTrie::remove` deals with the cases that do not belong to the recursive node algorithm (see Fig. 11, Line 1315). If the root is absent, removal returns `None` (Line 1341). If the root is a single leaf, it either removes that leaf or restores it unchanged (Lines 1320–1332). Otherwise, it delegates to `PNode::remove` and wraps the resulting root again (Lines 1335–1338).

Clients. Verifying the different algorithms is necessary, but the proven contracts also need to be usable in practice. To ensure that the specification is sufficiently complete and usable, we define several scenarios manipulating Patricia tries.

First, to check that the public contracts expose the expected map-like behavior, the proof-oriented wrappers `map_new`, `map_insert`, and `map_remove` restate the API in a functional style. They show that the imperative operations can be composed as map updates over the logical function $\texttt{get}_l$.

Layer	Ex. Body	Ex. Spec	Ex. Proof	Log. Def.	Lemmas	Total	Verif.	Ratio
Keys	74	40	1	118	32	265	191	2.58
Prefixes	37	35	0	121	44	237	200	5.41
Nodes	154	36	12	83	60	345	191	1.24
Wrapper	82	53	6	41	27	209	127	1.55
Clients	81	15	72	0	0	168	87	1.07
Total	428	179	91	363	163	1224	796	1.86

Fig. 12. Detailed statistics of the specification and the implementation (lines of code).

Then, the generic client executes a longer scenario with four pairwise distinct symbolic keys. It checks insertion into an arbitrary initial trie, deletion of a freshly inserted binding, repeated insertion of existing keys, lookup after updates, and removal of the four distinguished keys. The concrete clients instantiate the same scenario for Boolean-array keys and unsigned-integer keys.

Finally, the integer-key client uses `#[bitwise_proof]` and six trusted assertions to record the concrete bit inequalities that make its four test keys distinct. The core PTrie API itself does not rely on these trusted assertions. This scenario shows that the public API remains usable with a machine-word key representation.

Verification results. Figure 12 gives a breakdown of the nonblank, noncomment lines of the verified Rust artifact. The count separates the executable body from the lines introduced for specification and proof, and reports this decomposition for each implementation layer. The columns are defined as follows: Ex. Body represents the executable functions' implementation, i.e. the executable source code. Ex. Spec. denotes the contracts and tool-specific annotations attached to executable functions. This is the part most relevant to users who instantiate new key types or rely on the public API. Ex. Proof refers to the local proof-guiding constructs used inside executable functions. It reflects the additional proof effort needed to help SMT solvers discharge the verification conditions. Log. Def. captures the logical functions' specifications and bodies. These functions define the logical model used by the proof and reflect part of the complexity of each layer. This category also includes the proof-guiding lines that are zero for all layers except the prefix layer, where it takes two lines. Lemmas counts the lemma functions' definitions, and proof artifacts. It gives an indication of the amount and complexity of the auxiliary facts needed to prove each layer. Total gives the total number of nonblank, noncomment lines for the layer. Verif. contains all lines that are not executable-body lines, i.e. the lines added to specify the program, define its logical model, and guide the proof. Ratio measures the number of verification lines per executable-body line. It gives an indication of the verification overhead for each layer.

Overall, the artifact contains 428 lines of executable body and 796 verification-related lines, giving a ratio of 1.86 verification lines per executable-body line. However, the distribution is not uniform across layers. The prefix layer has the highest ratio, because it defines prefixes, their relation with the keys, and the routing logic. It has the largest number of verification lines, even though it is the smallest layer in terms of executable code.

Prover	Calls	Min	Avg	Max
`alt-ergo@2.6.2`	394	5 ms	30 ms	1100 ms
`z3@4.15.8`	27	7 ms	29 ms	240 ms
`cvc5@1.3.3`	13	22 ms	339 ms	1100 ms
Overall	434	5 ms	39 ms	1100 ms

Fig. 13. Proof timing statistics reported by `why3find`.

In contrast, the node layer has the largest executable body, but a limited verification overhead. It is the first layer that substantially uses lemmas and local assertions, since the recursive algorithms require some proof guidance. Compared to the amount of executable code analyzed, however, the proof effort remains relatively low, and many lemmas simply follow basic case analysis and recursive calls.

The wrapper layer mostly reuses the node-layer specifications and instantiates wrapper versions of the node lemmas.

Finally, the client layer contains most of the executable proof-guiding lines, as it checks that the verified contracts work as expected. This also suggests that the current public API could be more optimized for SMT solvers, as most of the difficulty comes from the executable code proof effort.

Experiments were run on a MacBook Pro (Apple M4 Max chip, 16 cores and 128 GB of memory), running macOS version 26.3.1 on `arm64`, with Creusot version 0.11.0, Why3 version 1.8.2 and WHY3FIND version 1.2.0, and the following SMT provers: Alt-Ergo version 2.6.2, Z3 version 4.15.8, cvc5 version 1.3.3. The wall clock for proving the 434 goals the first time (i.e. first instance of `cargo creusot prove`), using only one job, is around 4m30s. Table 13 shows, for each prover, how many verification conditions were proved and the minimum, average and maximum time spent to prove these conditions. The provers are listed in the order they were tried. The timeout was set to 6s.

Lessons learned. Prior to this verification case study on a Rust implementation of Patricia tries, we conducted an attempt to verify a similar implementation in C using the Frama-C verification platform [29]. As in the present work, we did not strictly focus on an existing real-life code, but took the liberty to create a simplified version inspired by real-life implementations. We allocated a similar level of effort to the verification of the C version. Although we already had experience with Frama-C, we were unable to complete the verification within the allocated time. Only the insertion function was partially verified without addressing complete functional properties of Patricia tries. The inability of solvers to automatically discharge the generated proof obligations prevented us from going further: even for our partial specification, there were around 70 interactive proofs to create and maintain (mostly via proof scripts in the interactive proof assistant of Frama-C) out of around 700 proof obligations. This was due in particular to a significant number of separation annotations and the need to maintain a companion ghost structure, similar to the one used for linked list [8,9] but adapted to match the tree topology. Similar difficulties were also encountered in the `TPM2-TSS` verification case study [44] involving linked lists. Targeting a Patricia trie, more complex structural invariants further increased the difficulty.

The Rust verification avoids these problems thanks to the Rust type system. As the internal node only uses a simple recursive definition, the type system invariants ensure the separation between a node and its children. These experiments suggest that Rust implementations of complex memory structures are easier to verify than comparable C implementations. Furthermore, instead of lemmas proven in the Rocq proof assistant [7] which can be necessary when using Frama-C [44], the Rust case study only required lemma functions which we found more convenient and easier to write and prove, since only the inductive step must be explicit, and the solvers complete the proof.

The Rust implementation considered here is not entirely idiomatic. While it constitutes a credible safe realization, some parts have been simplified and are not fully representative of a real-world implementation. For example, some functions on the keys are implemented recursively instead of using iterators. These choices were made to ease the specification development and the proof effort, but with additional effort, it would be possible to reach more realistic versions. Overall, we encountered very few limitations of Creusot in this case study, and we found the familiarization with the tool reasonably easy and fast.

6 Related Work

Verified Data Structures. Data structures and their manipulations are a common target of verification efforts, such as linked lists [8,9], trees [19], linked data structures including lists and trees [25,22], sets [20], hash tables [39] and generic containers [38] to name a few. Beyond the programming language and specification language, two important aspects vary across these verification efforts. First, how are the specifications written? In particular, is the data structure related to a *model* in the logic specification language to express the data structure API contracts? Second, to what extent is the verification automated: fully automated, partially automated, or mostly interactive?

Blanchard et al. verified a linked list API written in C using as a model either a ghost C array [8] or logic lists in the ACSL specification language [9]. In both cases, the verification is carried out with Frama-C [5,29], which is largely automated but may rely on interactive theorem proving, in particular for lemmas proved by induction. This approach was later made significantly more automated through an auto-active verification [32] methodology for Frama-C [10]. Dross et al. [19] verified red-black trees in SPARK (a subset of Ada), using arrays as the implementation structure. Their proof relies on ghost code and auto-active verification. For JML, Gladisch and Tyszberowicz [25] specify methods on a linked list structure using a pure observer rather than a model. This observer takes a list object and an index and returns the element at the position. They also study a tree structure, where the observer is `boolean reach(Tree a, int depth, Tree b)`, meaning that `b` is a subtree of `a` at depth exactly `depth`.

Filliâtre et al. [22] use an integer-indexed ghost data structure to represent both linked lists and trees implemented in WhyML and proved with Why3. In terms of list representation, their approach is close to the ghost arrays of Blanchard et al. [8], with one significant difference: the predicate relating the data structure to its model is not inductive but universally quantified over integers.

This makes the proofs easier, avoiding the need to assist SMT provers with lemmas proved by induction, whether interactively or automatically. Dubois uses a logic set [20] and auto-active verification in Why3. Pottier verifies an imperative hash table data structure implemented in OCaml and verified with CFML, a translator to Rocq (formerly Coq). This approach is mostly interactive.

For object-oriented languages such as Java and Eiffel, a common verification technique consists in attaching abstract models to classes under verification, as in the work of Polikarpova et al. [38]. These models are immutable and introduced solely for specification purposes. Nevertheless, they are expressed as regular classes of the source language, which makes them suitable for runtime assertion checking. In deductive verification, such classes can be translated into logical entities of the underlying provers. The faithfulness of this translation can itself be subject to verification [15].

In our work, we adopt an auto-active verification approach, relying on a few helper lemmas; only three require simple proofs by induction. For keys, we use the Creusot-provided `View` trait to relate a key to its model `Seq<bool>`. In this respect, our approach is close to works that rely on explicit models. For the trie, however, we do not use an explicit model, but instead rely on the $\texttt{contains}_l$ predicate. This is closer to the work of Gladisch and Tyszberowicz, in that the predicate observes the leaves of the trie. The approach of Creusot, unlike that of Polikarpova et al. [38], does not provide `Seq` as a regular pure functional implementation. Nevertheless, in both approaches, the model is translated into logical entities for the underlying provers (in the case of Creusot, through Why3).

Deductive Verification of Rust Programs. There are two main categories of deductive verifiers for Rust, depending on whether they trust the Rust type system and borrow checker. In the first case, the verifier uses the typing and borrowing information produced by the Rust compiler, together with user annotations, to generate proof obligations. Tools such as Prusti [1] and RefinedRust [24] use permission- or separation-logic-inspired reasoning to track ownership and aliasing properties. They differ substantially in their target fragments: Prusti focuses primarily on safe Rust, whereas RefinedRust is designed for high-assurance reasoning about Rust programs including unsafe components. This style can express low-level aliasing properties, but typically requires more explicit proof structure than SMT-oriented verification of safe Rust fragments. In the second case, the verifier relies on the guarantees enforced by the Rust type system and borrow checker: a large fragment of safe Rust can then be viewed through a functional lens. This makes proof obligations easier to generate and discharge. In practice, this leads to obligations solved either by SMT solvers, as in Creusot [18] and Verus [31], or by proof assistants, as in Aeneas [26].

Recent case studies using Verus also show that Rust verification is scaling toward larger systems [30], including OS and kernel components [13], cluster-management controllers [41], and security-oriented components [43]. Recent work also explores automated specification and proof generation for several data structures [40]. These works are complementary to ours. They demonstrate scale and systems-level verification in Rust, whereas our case study focuses on a compact

but proof-intensive mutable data structure, where the central difficulty is the interaction between compressed-prefix reasoning, finite-map specifications, and destructive tree updates in Creusot. Porting our case study to Verus would offer a useful comparison point for evaluating the maturity and scalability of the two tools.

Mutable borrows remain one of the main challenges. To the best of our knowledge, Aeneas and Creusot are currently the only mature representatives of this line of work that handle mutable borrows without severe usability restrictions, such as disallowing them in return positions. This is necessary to deal with functions such as `get_mut`. Moreover, we favor a more automated approach, which motivates our choice of Creusot over Aeneas in this work.

7 Conclusion and Future Work

This paper presents a deductive verification case study of a Patricia trie implemented in safe Rust and verified with Creusot. The verified code establishes the functional correctness of a small but meaningful API. Its specification is organized around an abstract key–value relation, while the implementation features some idiomatic imperative Rust, with boxed nodes and destructive updates. The central proof idea is to make prefix classification explicit, both operationally and logically, and to combine this abstraction with a global invariant that partitions the represented keys between subtrees. Automation works well once the right lemmas are in place, but update proofs still require a focused prefix-lemma layer, snapshots of prior states, and a few local assertions. We consider this a positive result for auto-active verification in Rust: even for a compressed mutable tree, the proof remains structured, modular, and close to the code.

A Rust crate implementing Patricia tries exists, namely `patricia_tree` (https://github.com/sile/patricia_tree). While its API is significantly richer, its implementation heavily relies on `unsafe`. This suggests several natural directions for future work. A first direction is to retain the safe implementation of a Patricia trie while extending the API with iterators and prefix-related operations such as splitting a Patricia trie at a given prefix and returning the entry whose key shares the longest common prefix with a query key. A second direction is to study a more low-level implementation of Patricia tries. Full direct verification of arbitrary unsafe Rust is not currently within Creusot's scope. Instead, Creusot supports reasoning about selected unsafe idioms, especially raw pointers and interior-mutable state, through ghost permissions complemented by pointer-liveness information. These features could be explored to verify low-level code. Exploring the combination of Creusot and Gilian is another promising direction for reasoning about code that uses `unsafe` [2].

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018, ANR-24-ASM2-0001). We thank the anonymous referees for helpful comments.

References

1. Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The Prusti project: Formal verification for Rust. In: NASA Formal Methods (NFM). LNCS, vol. 13260, pp. 88–108. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_5
2. Ayoun, S., Denis, X., Maksimovic, P., Gardner, P.: A hybrid approach to semi-automated Rust verification. Proc. ACM Program. Lang. **9**(PLDI), 970–992 (2025). https://doi.org/10.1145/3729289
3. Baker, H.G.: Use-Once variables and linear objects: Storage management, reflection and multi-threading. ACM SIGPLAN Notices **30**(1), 45–52 (1995). https://doi.org/10.1145/199818.199860
4. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
5. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
6. Bernier, T., Loulergue, F., Kosmatov, N.: Vptrie: Deductive verification of a Patricia trie with Creusot. Companion artifact. (May 2026). https://doi.org/10.5281/zenodo.19932306
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
8. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for lists: A critical module of contiki verified in Frama-C. In: NASA Formal Methods (NFM). LNCS, vol. 10811, pp. 37–53. Springer (2018). https://doi.org/10.1007/978-3-319-77935-5_3
9. Blanchard, A., Kosmatov, N., Loulergue, F.: Logic against ghosts: Comparison of two proof approaches for a list module. In: ACM Symposium on Applied Computing (SAC). pp. 2186–2195. ACM (2019). https://doi.org/10.1145/3297280.3297495
10. Blanchard, A., Loulergue, F., Kosmatov, N.: Towards full proof automation in Frama-C using auto-active verification. In: NASA Formal Methods (NFM). LNCS, vol. 11460, pp. 88–105. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_6
11. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. Int. J. Softw. Tools Technol. Transf. pp. 709–727 (2014). https://doi.org/10.1007/s10009-014-0314-5
12. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time Java. In: Programming Language Design and Implementation (PLDI). pp. 324–337. ACM (2003). https://doi.org/10.1145/781131.781168
13. Chen, X., Li, Z., Zhang, J., Narayanan, V., Burtsev, A.: Atmosphere: Practical verified kernels with rust and verus. In: Symposium on Operating Systems Principles. SOSP '25, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3731569.3764821, https://doi.org/10.1145/3731569.3764821

14. Conchon, S., Iguernelala, M., Mebsout, A.: A collaborative framework for non-linear integer arithmetic reasoning in Alt-Ergo. In: Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 161–168. IEEE Computer Society (2013). https://doi.org/10.1109/SYNASC.2013.29
15. Darvas, Á., Müller, P.: Proving consistency and completeness of model classes using theory interpretation. In: Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 6013, pp. 218–232. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12029-9_16
16. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). p. 337–340. LNCS, Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Denis, X., Jourdan, J.H.: Specifying and verifying higher-order Rust iterators. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 13994, pp. 93–110. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_9
18. Denis, X., Jourdan, J.H., Marché, C.: Creusot: A foundry for the deductive verification of rust programs. In: Formal Engineering Methods (ICFEM). LNCS, vol. 13478, pp. 90–105. Springer (2022). https://doi.org/10.1007/978-3-031-17244-1_6
19. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: NASA Formal Methods (NFM). pp. 68–83 (2017). https://doi.org/10.1007/978-3-319-57288-8_5
20. Dubois, C.: Deductive verification of sparse sets in Why3. In: Verified Software. Theories, Tools and Experiments (VSTTE). pp. 28–46. LNCS, Springer (2024). https://doi.org/10.1007/978-3-031-86695-1_3
21. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: Programming Languages and Systems (ESOP). LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
22. Filliâtre, J., Paskevich, A., Danvy, O.: When separation arithmetic is enough. In: Integrated Formal Methods (iFM). LNCS, vol. 16194, pp. 23–37. Springer (2025). https://doi.org/10.1007/978-3-032-10794-7_2
23. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960). https://doi.org/10.1145/367390.367400
24. Gäher, L., Sammler, M., Jung, R., Krebbers, R., Dreyer, D.: RefinedRust: A type system for high-assurance verification of Rust programs. Proc. ACM Program. Lang. **8**(PLDI), 1115–1139 (2024). https://doi.org/10.1145/3656422
25. Gladisch, C., Tyszberowicz, S.: Specifying linked data structures in JML for combining formal verification and testing. Sci. Comput. Program. **107-108**, 19 – 40 (2015). https://doi.org/10.1016/j.scico.2015.02.005
26. Ho, S., Protzenko, J.: Aeneas: Rust verification by functional translation. Proc. ACM Program. Lang. **6**(ICFP), 711–741 (2022). https://doi.org/10.1145/3547647
27. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 66:1–66:34 (2018). https://doi.org/10.1145/3158154
28. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of C programs. In: Runtime Verification (RV). LNCS, vol. 8174, pp. 167–182. Springer (2013). https://doi.org/10.1007/978-3-642-40787-1_10
29. Kosmatov, N., Prevosto, V., Signoles, J. (eds.): Guide to Software Verification with Frama-C. Core Components, Usages, and Applications. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1

30. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: 2024 Symposium on Operating Systems Principles. https://doi.org/10.1145/3694715.3695952
31. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA), 286–315 (2023). https://doi.org/10.1145/3586037
32. Leino, K.R.M., Moskal, M.: Usable auto-active verification (2010), http://fm.csl.sri.com/UV10/
33. Matsushita, Y., Denis, X., Jourdan, J.H., Dreyer, D.: RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In: Programming Language Design and Implementation (PLDI). pp. 841–856. ACM (2022). https://doi.org/10.1145/3519939.3523704
34. Minsky, N.H.: Towards alias-free pointers. In: Object-Oriented Programming (ECOOP), LNCS, vol. 1098, pp. 189–209. Springer (1996). https://doi.org/10.1007/BFb0053062
35. Morrison, D.R.: PATRICIA – practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968). https://doi.org/10.1145/321479.321481
36. Mühlberg, J.T., White, D.H., Dodds, M., Lüttgen, G., Piessens, F.: Learning assertions to verify linked-list programs. In: Calinescu, R., Rumpe, B. (eds.) Software Engineering and Formal Methods (SEFM). pp. 37–52. Springer (2015). https://doi.org/10.1007/978-3-319-22969-0_3
37. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Object-Oriented Programming (ECOOP), LNCS, vol. 1445, pp. 158–185. Springer (1998). https://doi.org/10.1007/BFb0054091
38. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. Formal Asp. Comput. **30**(5), 495–523 (2018). https://doi.org/10.1007/s00165-017-0435-1
39. Pottier, F.: Verifying a hash table and its iterators in higher-order separation logic. In: Certified Programs and Proofs (CPP). pp. 3–16. ACM (2017). https://doi.org/10.1145/3018610.3018624
40. Sun, C., Sun, Y., Amrollahi, D., Zhang, E., Lahiri, S., Lu, S., Dill, D., Barrett, C.: Veristruct: Ai-assisted automated verification of data-structure modules in verus. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-032-22749-2_6, https://doi.org/10.1007/978-3-032-22749-2_6
41. Sun, X., Ma, W., Gu, J.T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., Xu, T.: Anvil: verifying liveness of cluster management controllers. In: Conference on Operating Systems Design and Implementation. OSDI'24, USENIX Association, USA
42. Tafat, A., Marché, C.: Binary Heaps Formally Verified in Why3. Research Report RR-7780, INRIA (2011), https://hal.inria.fr/inria-00636083
43. Zhou, Z., Anjali, Chen, W., Gong, S., Hawblitzel, C., Cui, W.: Verismo: a verified security module for confidential vms. In: Conference on Operating Systems Design and Implementation. OSDI'24, USENIX Association, USA
44. Ziani, Y., Bernier, T., Kosmatov, N., Loulergue, F., Gracia Pérez, D.: Towards formal verification of a tpm software stack: Achievements and opportunities. Formal Asp. Comput. **37**(4) (2025). https://doi.org/10.1145/3743153