



LINUX
PLUMBERS
CONFERENCE

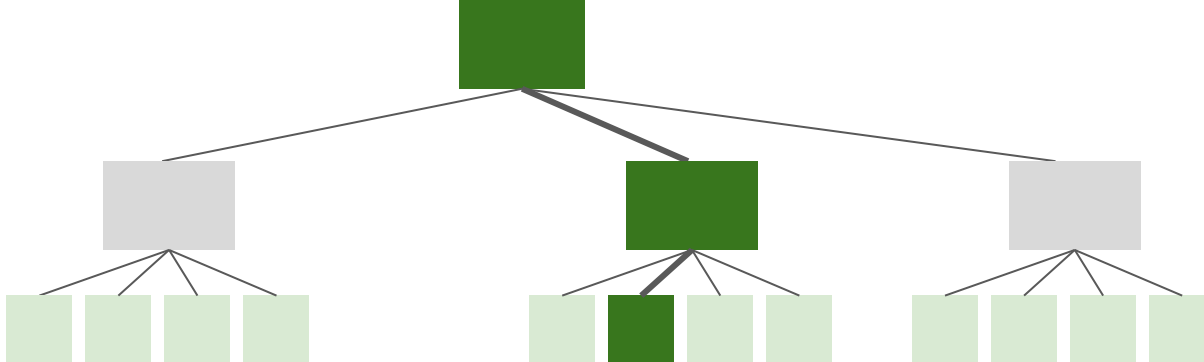
PRAGUE, CZECHIA / OCT. 5-7, 2026

Towards Verification of the Linux Kernel Library XArray

Corinn Tiffany, Julia Lawall
Inria

XArray is a core data structure in the Linux kernel

- Users interact with it like a resizable array of pointers
- Indices map to entries
- Underlying data structure is a tree; each index dictates a lookup path



Specialized functionality helps kernel clients

Examples of clients: page cache, KVM

Useful space optimization for mapping a range of indices to one entry.

In the page cache, a large range of addresses maps to a single struct representing one or more pages.

Logic bugs can break lots of kernel code

XArray: Fix xas_find returning too many entries

mm/huge_memory: Fix xarray node memory leak

XArray: Fix infinite loop with entry at ULONG_MAX

Vision: Apply program verification to the XArray implementation

Write specifications to precisely describe the API.

Use program verification tools to exhaustively check that the implementation upholds the specified contract.

Vision: Apply program verification to the XArray implementation

Write specifications to precisely describe the API.

Use program verification tools to exhaustively check that the implementation upholds the specified contract.

Benefits:

- Make requirements and behavior explicit
- Increase confidence in implementation correctness

Agenda

- Scope of current project and status of prototype
- Walk-through of relevant XArray implementation information
- Overview of our verification toolset and approach
- Discussion: AI agent use cases, verification of the Rust port, what behaviors to verify?

Target: Verify key-value mappings and absence of RTEs

Verify **key-value mappings** for `xa_load` and `xa_store`.

→ When we store a value into an XArray, we can load the same value back.

Prove **absence of runtime errors (RTEs)**.

→ Prove that there are no null pointer derefs, arithmetic overflow, etc.

Non-goals: concurrency and marks

The tool we use doesn't supporting reasoning about **concurrency**

Not trying to verify how the implementation updates **marks**

- Metadata on each entry with 3 states
- Metadata on each node about the status of its transitive children

Current status of project

- Representation of the tree that facilitates writing and proving specs
- Verification of **xa_load** and **xas_expand** (and ~45 helpers)
- Partial verification of **xa_store**

Two central challenges

1. The XArray implementation is **complex and optimized**.
 - Specifications cover diverse behaviors and tree operations
2. The verification tool that we use struggles with **separation of memory**.
 - When a function modifies one node in the tree, we have to prove that the others didn't change, too.

Overview of XArray implementation

Loading a value with xa_load

Use the index as an encoding of a path through the tree.

Cursor tracks current location in the tree.

index to load =
0b**001****1****000****11**

node A (root)

slots = [- **B** - - - - -]

node B

slots = [- - - - **C** - - -]

node C

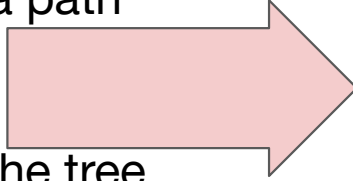
slots = [- - - **v** - - - -]

value

Loading a value with xa_load

Use the index as an encoding of a path through the tree.

Cursor tracks current location in the tree.



index to load =
0b**001****1****000****11**

node A (root)

slots = [- **B** - - - - -]

node B

slots = [- - - - **C** - - -]

node C

slots = [- - - **v** - - - -]

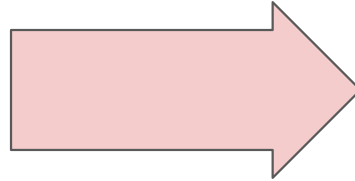
value

Loading a value with xa_load

Use the index as an encoding of a path through the tree.

Cursor tracks current location in the tree.

index to load =
0b001100011



node A (root)

slots = [- **B** - - - - -]

node B

slots = [- - - - **C** - - -]

node C

slots = [- - - **v** - - - -]

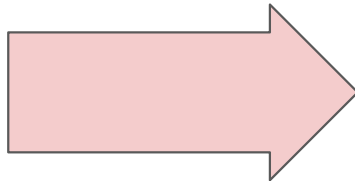
value

Loading a value with xa_load

Use the index as an encoding of a path through the tree.

Cursor tracks current location in the tree.

index to load =
0b001100011



node A (root)

slots = [- **B** - - - - -]

node B

slots = [- - - - **C** - - -]

node C

slots = [- - - **v** - - - -]

value

Loading a value with xa_load

Use the index as an encoding of a path through the tree.

Cursor tracks current location in the tree.

index to load =
0b**001****1****000****11**

node A (root)

slots = [- **B** - - - - -]

node B

slots = [- - - - **C** - - -]

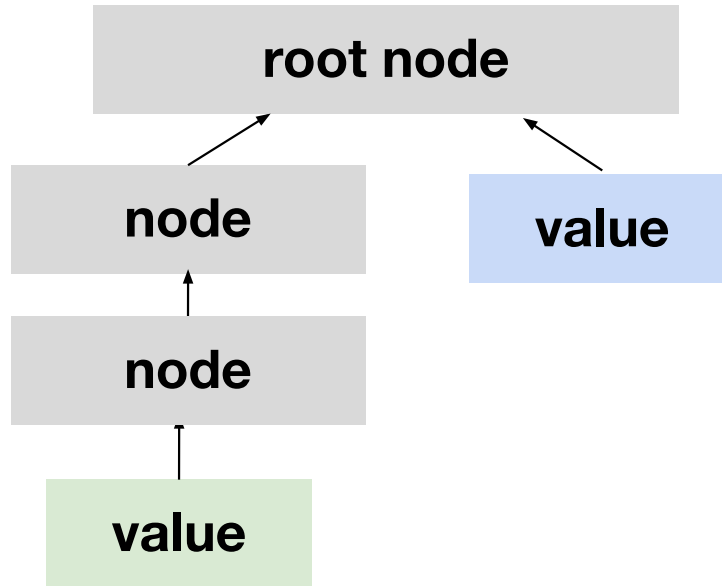
node C

slots = [- - - **v** - - - -]

value

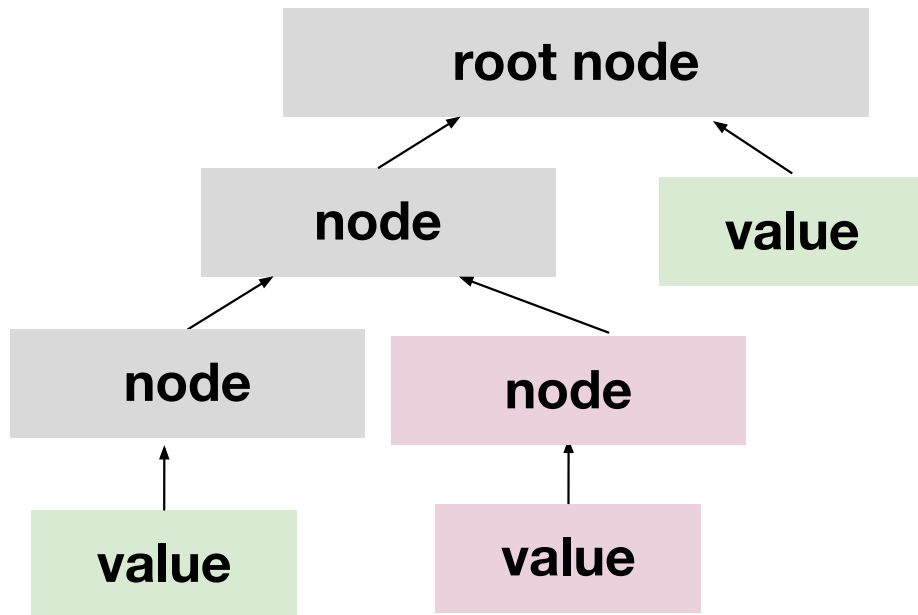
A **multi-index entry** maps a range of indices to one entry.

Insert an entry higher in the tree → multiple lookups terminate at the same entry



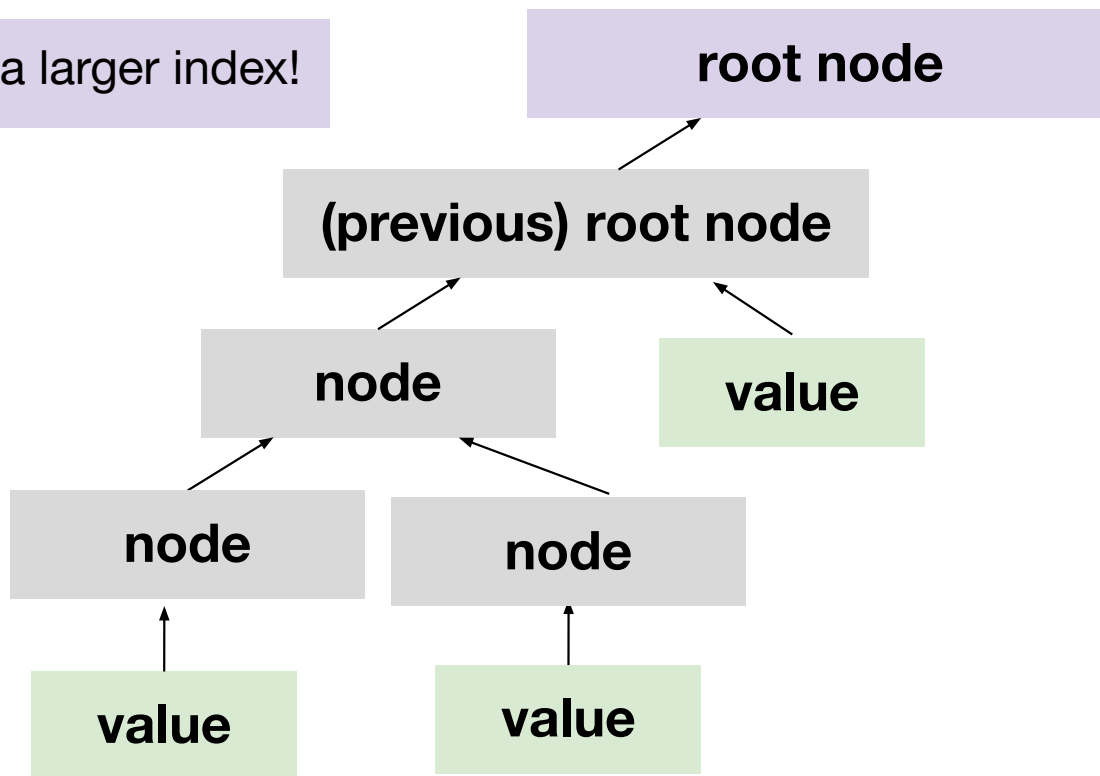
xa_store adds nodes as needed

Insert new internal nodes if mapping the target index requires it



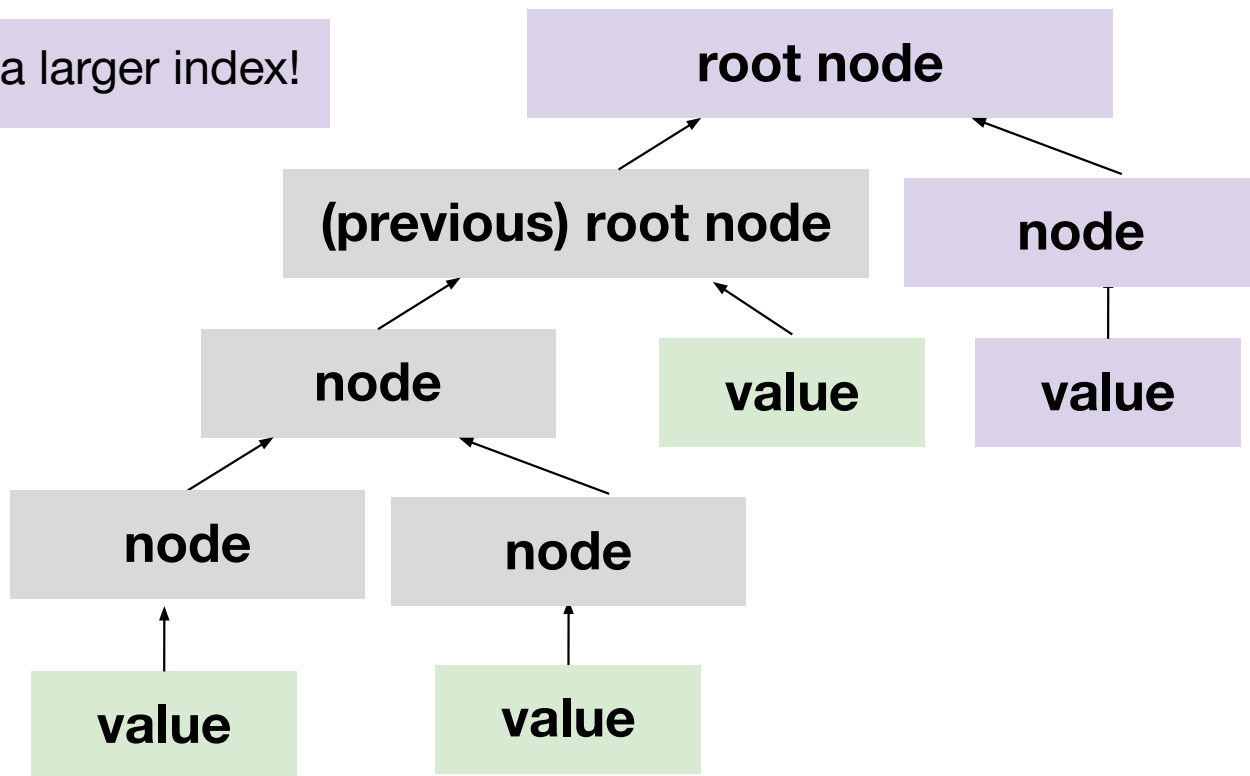
xa_store adds nodes as needed

Grow to fit a larger index!



xa_store adds nodes as needed

Grow to fit a larger index!



Verification Toolset and Approach

Write contracts in a structured specification language

Contracts describe (1) the function's **requirements**, (2) what the function **ensures** after a call, and (3) side-effects (what memory is written).

We use the ACSL specification language:

- Preconditions as `requires`, postconditions as `ensures`
- Pointer idioms: `\separated(p, q)`, `\valid(p)`, `\valid_read(p)`
- Quantifiers: `\forall`, `\exists`

We use the Frama-C and WP verification toolset

Frama-C is a program analysis tool; WP is a plug-in for deductive verification.

Input: the source code, our specification, and our supplementary annotations

→ WP translates input to logic formulas and sends to SMT solvers to prove

Output: “Yes” or “Maybe” for each postcondition

We use the Frama-C and WP verification toolset

Frama-C is a program analysis tool; WP is a plug-in for deductive verification.

Input: the source code, our specification, and our supplementary annotations

→ WP translates input to logic formulas and sends to SMT solvers to prove

Output: “Yes” or “Maybe” for each postcondition

Most of the engineering effort is adding annotations.

To verify `xas_expand()`, ~ 7:1 ratio of Lines of Annotation to LoC.

Challenge: specifications over the pointer-based tree

ex) Precondition of “All nodes are separated,” then modify one node.

How can we prove that the other nodes are unchanged?

Naive approach: Use a recursive precondition to describe separate nodes. The SMT solvers then need to recur over the tree to prove that the others are distinct, and therefore unchanged.

Challenge: specifications over the pointer-based tree

ex) Precondition of “All nodes are separated,” then modify one node.

How can we prove that the other nodes are unchanged?

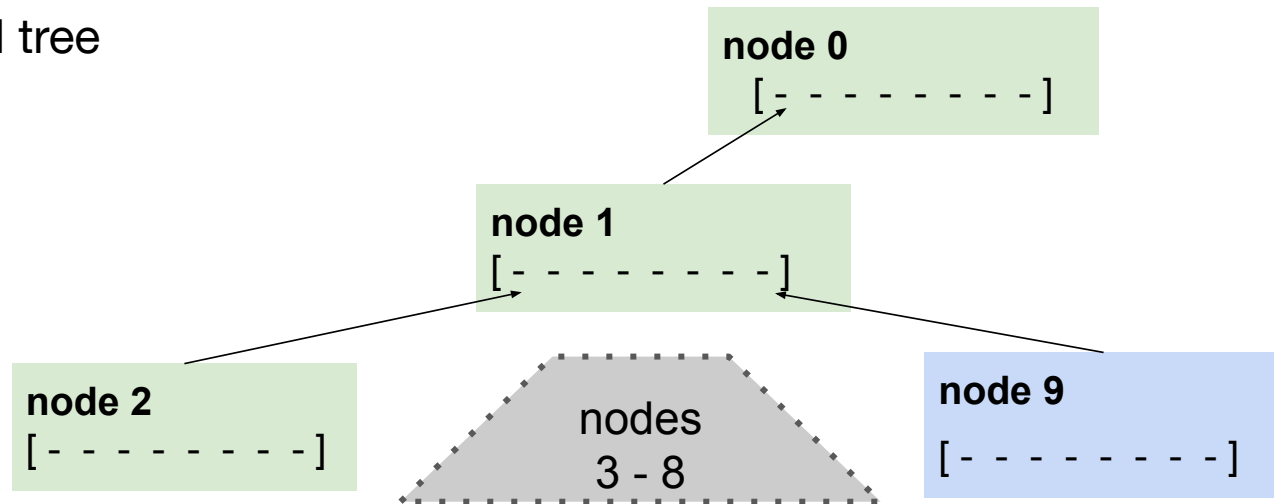
Naive approach: Use a recursive precondition to describe separate nodes. The SMT solvers then need to recur over the tree to prove that the others are distinct, and therefore unchanged.

Model the tree in a flat *companion ghost array* containing the tree's nodes.

→ “All nodes are separated” becomes a `\forallall` over the array.

The ghost array stores each tree node at a constant index

Real tree



ghost struct xa_node* []

0	1	2	3	4	5	6	7	8	9	...
---	---	---	---	---	---	---	---	---	---	-----

Iterative process to develop supplementary annotations

1. Write simplified implementation and a contract of the expected behavior.
2. Iterate on annotations until the goals prove.
3. Add complexity back into the simplified function. Existing goals stop proving
→ Update annotations.

Doing this manually is tedious and time-consuming.

Iterative process to develop supplementary annotations

1. Write simplified implementation and a contract of the expected behavior.
2. Iterate on annotations until the goals prove.
3. Add complexity back into the simplified function. Existing goals stop proving
→ Update annotations.

Doing this manually is tedious and time-consuming.

Coding agents can automate much of the repetitive work.

Agents can debug failing goals effectively

Created an agent skill to **triage failing goals**.

- Provides a workflow to choose which goals to debug first and a list of project-specific failures and fixes.

Agents can debug failing goals effectively

Created an agent skill to **triage failing goals**.

- Provides a workflow to choose which goals to debug first and a list of project-specific failures and fixes.

Agents can **read the tool's internal representation** of the program.

- WP dumps a representation of the failing goal that shows the different memory states and facts that are known at each program point.
- Agents are capable of searching through this output and suggesting targeted fixes.

Potential to verify the Rust XArray port

Rust XArray is under active development

- If we want to call the Rust impl from C clients, the contracts need to match!

Can we use the C specification as a reference to write spec for analogous functions in the Rust implementation?

Preliminary experiment suggests that verifying Rust version is much easier; Rust's aliasing rules simplify separation between structures.

Discussion

- Contracts describe key-value mapping behavior
- Verified `xa_load()` and `xas_expand()`; progress on `xa_store()`
- Agents can automate or facilitate much of the annotation development
- Comparative specification of the Rust port could facilitate checking parity between the implementations

What other kernel data structures could benefit from this approach?

Which requirements and behaviors would be useful to formalize?