Automated Verification of Systems Code

Systems and Formalisms Lab


Can Cebeci


Yonghao Zou


Diyu Zhou


George Candea


Clément Pit-Claudel

Formal verification of low-level systems code can be largely automated through domain-specific logic encodings

Problem: verifying systems code requires too many code annotations

State of the art in verifying systems

System	Verifier	Annotation to code ratio
seL4 Kernel	Isabelle/HOL	20 lines per LOC
pKVM memory allocator	CN	7.6 lines per LOC
IronClad apps	Dafny	4.8 lines per LOC

Cause: Low-level programming idioms

- Pointer arithmetic
- Type casting
- Physical addresses
- Bit twiddling
- Dynamic allocation

```
phys_addr_t addr = page_to_phys(p);

/*@ apply find_buddy_
   cn_hyp_page_to_pfn(__hyp_vmemmap,p),
   order); @*/
addr ^= (PAGE_SIZE << order);
```

Cause: Strict boundaries for proof modularity

- Function contracts
- Framing conditions
- Predicate packing/unpacking

```
struct hyp_page *node_to_page(struct list_head *node)
/*@ accesses __hyp_vmemmap; hyp_physvirt_offset @*/
/*@ requires let phys=((integer)node)
+hyp_physvirt_offset@*/
/*@ requires phys < power(2, 64) @*/
   . . .
/*@ ensures return == page @*/
/*@ ensures {__hyp_vmemmap} unchanged;
{hyp_physvirt_offset} unchanged @*/
{ return hyp_virt_to_page(node); }
```

TPot: an automated verifier for low-level C programs

Automation over modularity

- Specifications at the system level, not the function level
- Rely more heavily on the solver instead of manually managing proof state

Verification as a CI process

- Separation between verification and debugging
- Larger time budget allows for more room for automation
- Property-based testing for free through executable specs

Untyped & lazy memory model

- Representing objects as arrays of bytes automates type casting
- Pointers as numerical values automate pointer arithmetic
- Lazily instantiating objects automates dynamic allocation

Challenge: Avoiding solver explosion

Long-running solver queries: ✓
Non-terminating solver queries: ✗

Need to avoid instability in order to push more work to the solver without causing non-termination.

Biggest culprits: interactions between ∀ quantifiers, comparisons involving bit vectors

Key technique: Co-design of a spec language and the underlying logic encoding

Specifications:
- Limited quantification: over array elements, not generic
- Memory ownership through a naming abstraction

Encoding:
- Most quantifiers are handled by the verifier, not the solver
- Conversion between logical bit vectors and integers and lazy instantiation of axioms for this conversion

```
//    Excerpt from the TPot spec for    //
//    pKVM's memory allocator.          //

//  -- helpers --  //
// ...
bool list_head_well_formed(struct list_head *h,
                           int64_t i) {
    if (h->next == h) {
        // The list is empty. prev must be h.
        return h->prev == h;
    };

    // Next is a list node with the correct order,
    // and its prev is h.
    return list_node_well_formed(h->next) &&
        get_order(h->next) == i &&
        h->next->prev == h;
}

// -- invariants -- //
bool inv__pool_alloc() {
    names_obj(pool, struct mem_pool);
}
bool inv__free_area() {
    return forall_elem(pool->free_lists,
        &list_head_well_formed);
}
```