Automated Verification of Systems Code

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

<table>
<thead>
<tr>
<th>System</th>
<th>Verifier</th>
<th>Annotation to code ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>seL4 Kernel</td>
<td>Isabelle/HOL</td>
<td>20 lines per LOC</td>
</tr>
<tr>
<td>pKVM memory alloc</td>
<td>CN</td>
<td>7.6 lines per LOC</td>
</tr>
<tr>
<td>IronClad apps</td>
<td>Dafny</td>
<td>4.8 lines per LOC</td>
</tr>
</tbody>
</table>

Cause: Low-level programming idioms
- Pointer arithmetic
- Type casting
- Physical addresses
- Bit twiddling
- Dynamic allocation

Cause: Strict boundaries for proof modularity
- Function arithmetic
- Framing conditions
- Predicate packing/unpacking

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

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

Verification as a CI process

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

Challenge: Avoiding solver explosion

// Excerpt from the TPot spec for // // pKVM's memory allocator. // // -- helpers -- // // ... bool list_head_well_formed(struct list_head *h, int64_t i) { // The list is empty. prev must be h. if (h->next == 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); }