Ghost in the Android Shell: Pragmatic Test-oracle Specification of a Production Hypervisor

Kayvan Memarian[1] Ben Simner[1] David Kaloper-Meršinjak Thibaut Pérami Peter Sewell

University of Cambridge

[1] These authors contributed equally

SOSP, 2025-10

Idea

How can we cheaply and easily improve assurance ?
Write executable-as-test-oracle specs
... but inline, in C!
... and test them!

Making systems code secure remains very challenging: conventional practice doesn't suffice, and full functional verification has substantial barriers to use.

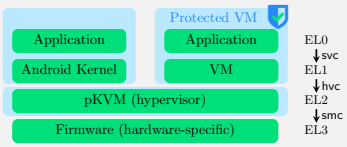
We explore a more lightweight approach to building confidence for a production hypervisor: we specify the desired behaviour in a way that can be used as a test oracle and check it at runtime. The setting makes that hard: it's intertwined with the underlying architecture; it's concurrent, with nontrivial ownership; the spec must be loose; the hypervisor runs bare-metal; naive random testing would quickly crash the whole system; and the hypervisor is written in C.

We show how all of these can be overcome to make a practically useful specification and find critical bugs.

This is not at all what conventional developers (nor what formal verifiers) normally do – but we argue that, with the appropriate mindset, they easily could and should.

Target: pKVM, Android's hypervisor

pKVM enforces isolation between the 'host' Android kernel and guest VMs.



with a limited API for managing guests and memory:

Creation	init_vm	init_vcpu
Memory	host_share_hyp	host_unshare_hyp
	host_map_guest	guest_share_host
	guest_unshare_host	
Destruction	teardown_vm	host_reclaim_page
Context switching	vcpu_run	
Communication	host_mem_abort	handle_pvm_entry_dabt

Checking Specs Dynamically — Specifying a Production Hypervisor

(1) — Computing an Abstraction

We define abstraction functions ...

We do this for all data structures defining the hypervisor state:

- ▷ pKVM's own page table (🔒).
- ▷ The Android guest page table (🔒).
- ▷ Each guest page table (🔒).
- ▷ The set of VMIDs (🔒), the guest VM metadata (🔒), and the vCPUs (🔒+).
- ▷ Thread-local register state.

... by defining a mathematical abstraction of the implementation state ...

e.g. the page tables in memory encode a simple mathematical function representing the translation.

[Cartoon of the abstraction of a page table.]

... and computing it at runtime, in C.

```
void …interpret_pgtable(mapping *mapping_out, kvm_pte_t *pgd,
                        ghost_stage_t stage, u8 level,...)
{ ...
  for (u64 idx = 0; idx < 512; idx++) {
    u64 va_offset_in_region = idx * nr_pages * PAGE_SIZE;
    u64 va_partial_new = va_partial | va_offset_in_region;
    u64 pte = pgd[idx];
    enum entry_kind ek = entry_kind(pte, level);
    switch(ek) {
    case EK_BLOCK: {
      u64 oa = pte & PTE_FIELD_OA_MASK[level];
      u64 attr = pte & PTE_FIELD_ATTRS_MASK;
      struct maplet_target_mapped m =
        parse_mapped(stage, mair, level, oa,
                     nr_pages, attr, next_level_aal);
      struct maplet_target t =
        maplet_target_mapped(va_partial_new, nr_pages, m);
      extend_mapping_coalesce(mapping_out, stage,
                              va_partial_new, nr_pages, t);
      break;
    }
    ...
```

(2) — Specifying a hypercall

Define partial abstract states

The abstractions of each of the data structures partitioned by the ownership discipline form the whole abstract state, but may only be partially known at runtime:

```
recorded post ghost state diff from recorded pre:
host.share +ipa :...101b18000 phys:101b18000 SO RWX M
pkvm.pgt   +virt:8000c1b18000 phys:101b18000 SB RW- M
regs       -r0=.....c600000d r1=........101b18
regs       +r0=.............0 r1=............0
```

[Cartoon of a partial abstract state, with only pKVM and host pagetables (the rest of the state is absent in this view) and an example runtime diff over that partial state.]

Specify hypercalls as a computable function, in C!

Then define functions, in C, from an initial state to an expected final state, specifying precisely what parts of the state the hypercall must update and how, but crucially *without mentioning parts the implementation need not touch*.

An example hypercall spec

```
static bool compute_post_…pkvm_host_share_hyp(
  struct ghost_state *g_post, struct ghost_state *g_pre,
  struct ghost_call_data *call)
{
  // Address space conversions
  u64 pfn = ghost_read_gpr(g_pre, 1);
  phys_addr_t phys = hyp_pfn_to_phys(pfn);
  host_ipa_t host_addr = host_ipa_of_phys(phys);
  hyp_va_t hyp_addr = hyp_va_of_phys(g_pre, phys);
  int ret = 0;

  // Permissions checks
  if (!is_owned_exclusively_by(g_pre, GHOST_HOST, phys)) {
    ret = -EPERM;
    goto out;
  }

  // Initialisation of the (partial) post-state
  copy_abstraction_host(g_post, g_pre);
  copy_abstraction_pkvm(g_post, g_pre);

  // Construction of abstract mapping attributes
  bool is_memory = ghost_addr_is_allowed_memory(g_pre, phys);
  struct maplet_attributes host_attrs =
    ghost_host_memory_attributes(is_memory, SHARED_OWNED);
  struct maplet_attributes hyp_attrs =
    ghost_hyp_memory_attributes(is_memory, SHARED_BORROWED);

  // Update abstract mappings with new targets
  mapping_update(
    &g_post->host.shared,
    g_pre->host.shared,
    MAP_INSERT_PAGE, GHOST_STAGE2, host_addr, 1,
    maplet_target_mapped_attrs(phys, 1, host_attrs)
  );
  mapping_update(
    &g_post->pkvm.pgt.mapping,
    g_pre->pkvm.pgt.mapping,
    MAP_INSERT_PAGE, GHOST_STAGE1, hyp_addr, 1,
    maplet_target_mapped_attrs(phys, 1, hyp_attrs)
  );

  // Epilogue: update the host register state
out:
  ghost_write_gpr(g_post, 1, ret);
  copy_registers_to_host(g_post);
  return true;
}
```

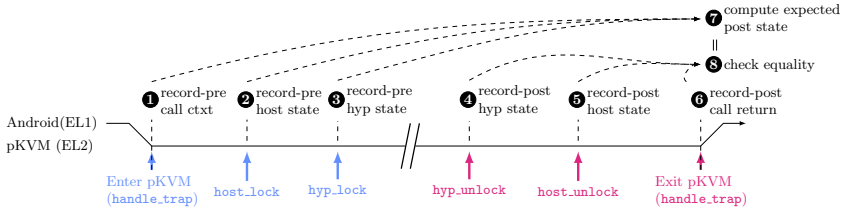
(3) — Checking it

States can be recorded, and specs computed and checked, at runtime:

(1-6) Compute abstract state on entry and exit to hypervisor, and 🔒 and 🔓 of data structures.

(7-8) Run spec function to compute expected state, and compare against abstract state of implementation.

- No fancy tools required.
- No experts needed.



[Execution of an instrumented pKVM hypercall with spec checking.]

We thank the pKVM development team, especially Will Deacon and Keir Fraser, for extensive discussions, and Ben Laurie and Sarah de Haas for their support. We thank Jean Pichon-Pharabod for the stimulus to write the paper, and for the first part of the title. We thank the reviewers for their helpful comments.
This work was funded in part by Google. This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER). This work is supported by ERC-2024-POC grant ELVER-CHECK, 101189371. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694. The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported by EPSRC grant EP/Z000580/1.