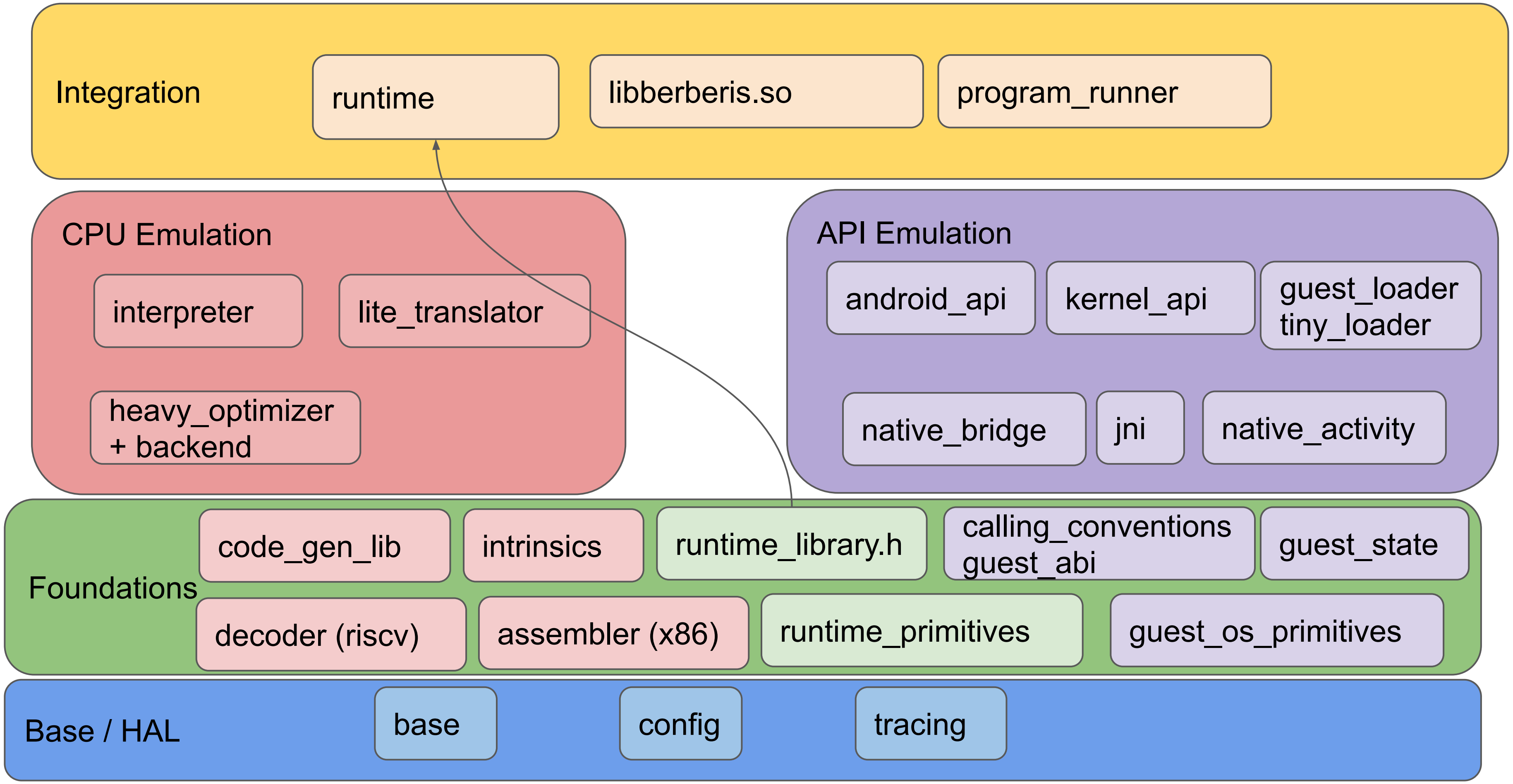
Berberis Modules Dec’23

go/berberis-modules

levarum@google.com

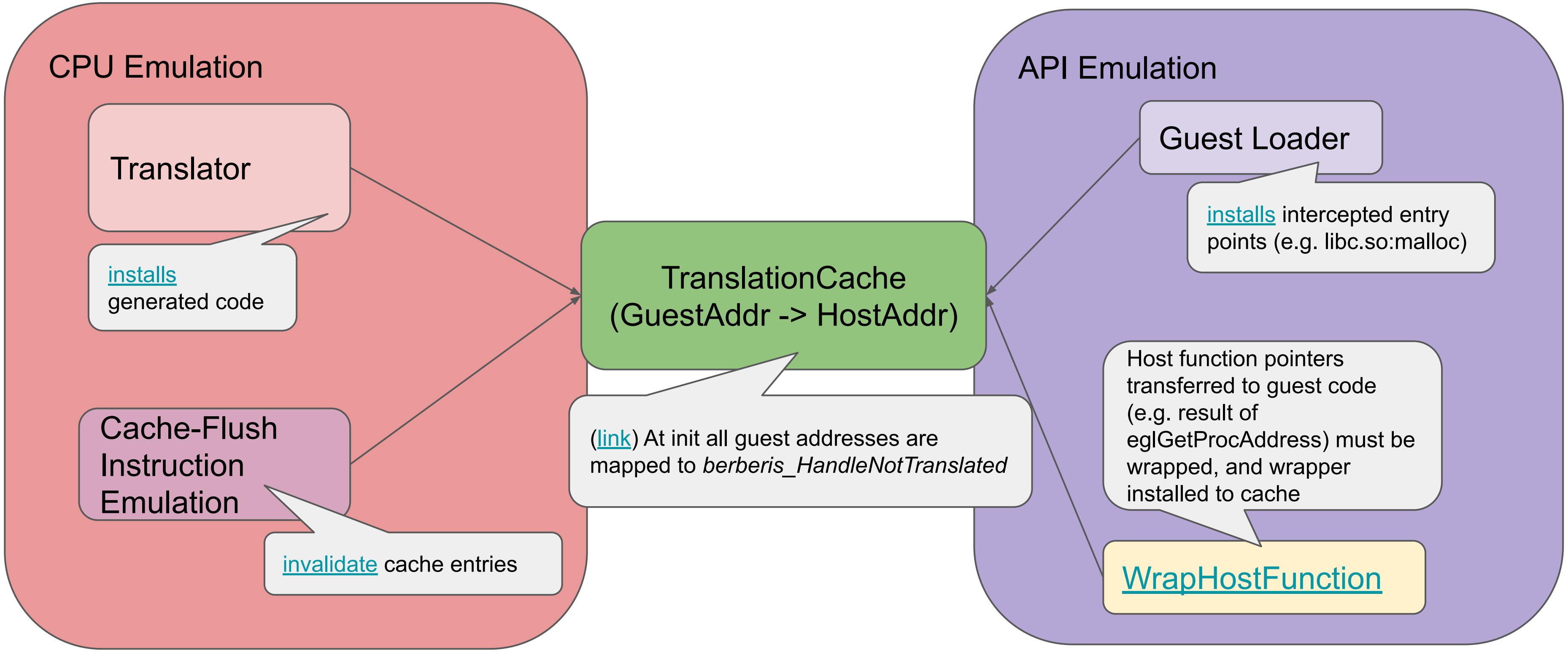

Integration
runtime
libberberis.so
program_runner
CPU Emulation
interpreter
lite_translator
heavy_optimizer
+ backend
API Emulation
android_api
kernel_api
guest_loader
tiny_loader
native_bridge
jni
native_activity
Foundations
code_gen_lib
intrinsics
runtime_library.h
calling_conventions
guest_abi
guest_state
decoder (riscv)
assembler (x86)
runtime_primitives
guest_os_primitives
Base / HAL
base
config
tracing

CPU Emulation

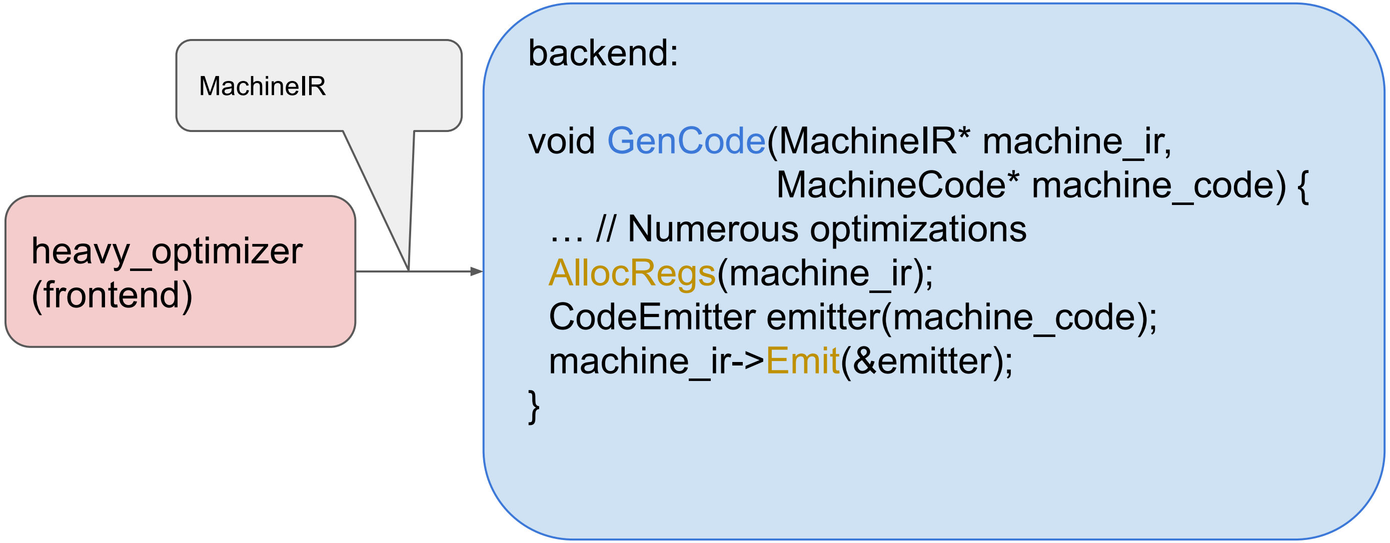
Main loop

```
// Simplified
void ExecuteGuest(ProcessState* state) {
  TranslationCache* cache = TranslationCache::GetInstance();

  for (;;) {
    // Current guest PC.
    auto pc = state->cpu.insn_addr;

    // Lookup host PC in cache.
    auto code = cache->GetHostCodePtr(pc)->load();
    if (code == kEntryStop) {
      break;
    }

    // Assembly-written entry point to generated code with custom internal ABI.
    berberis_RunGeneratedCode(state, code);
  }

}
```

Translation cache


CPU Emulation
Translator
installs generated code
Cache-Flush Instruction Emulation
invalidate cache entries
TranslationCache (GuestAddr -> HostAddr)
(link) At init all guest addresses are mapped to *berberis_HandleNotTranslated*
API Emulation
Guest Loader
installs intercepted entry points (e.g. libc.so:malloc)
Host function pointers transferred to guest code (e.g. result of eglGetProcAddress) must be wrapped, and wrapper installed to cache
WrapHostFunction

Translator

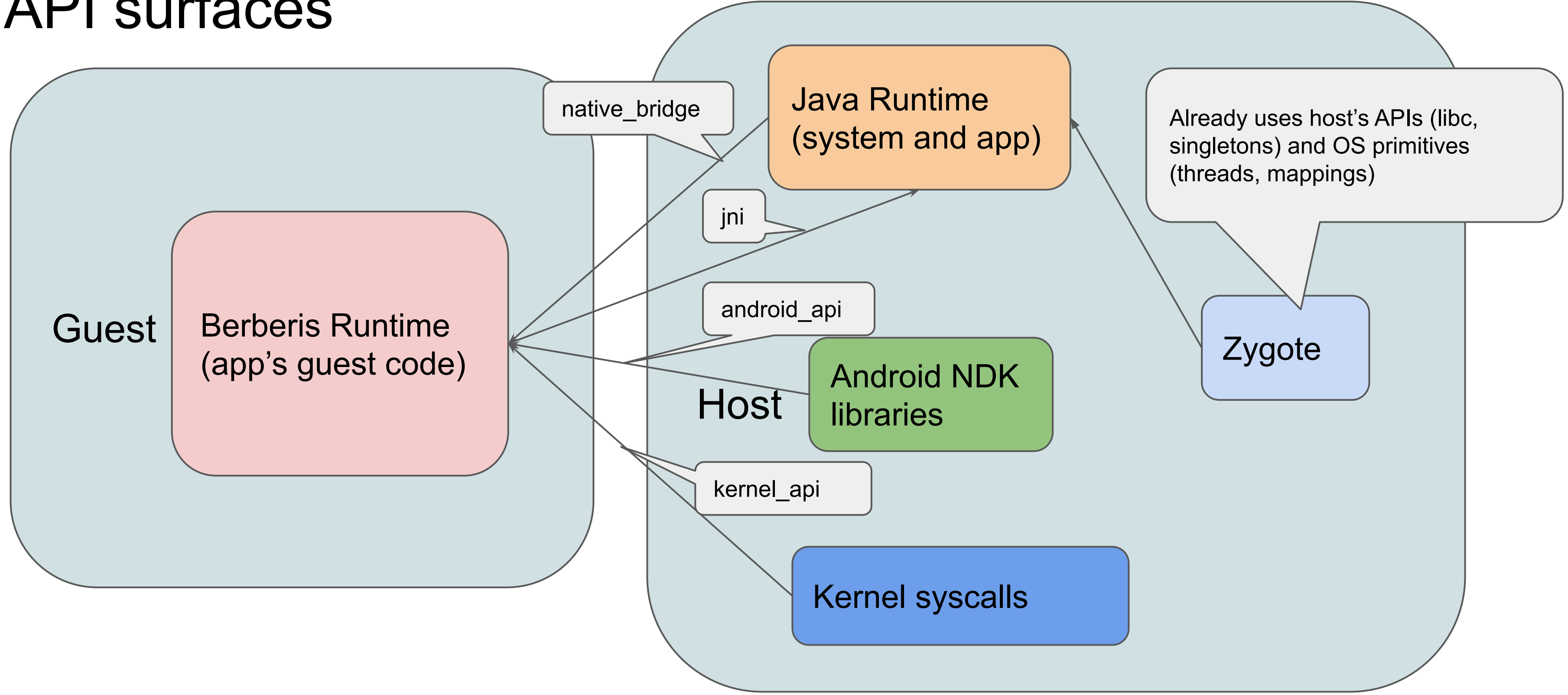


backend:

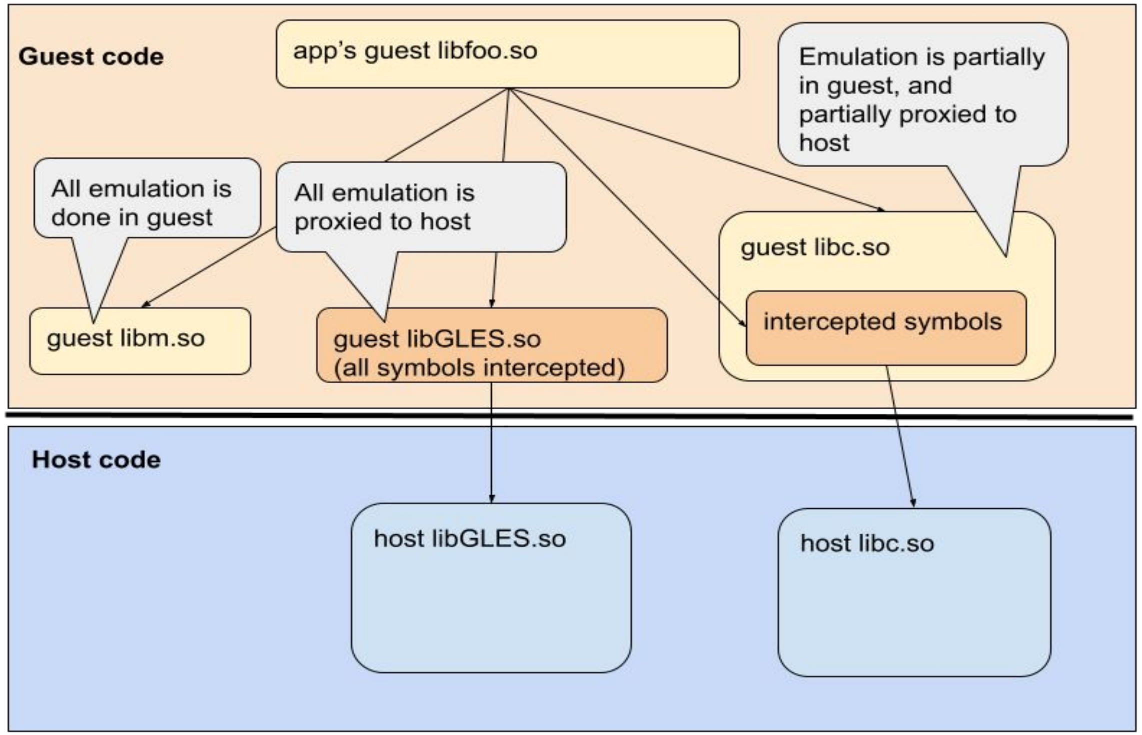
```
void GenCode(MachineIR* machine_ir,
             MachineCode* machine_code) {
  … // Numerous optimizations
  AllocRegs(machine_ir);
  CodeEmitter emitter(machine_code);
  machine_ir->Emit(&emitter);
}
```

API Emulation

API surfaces


Guest
Berberis Runtime (app's guest code)
native_bridge
jni
android_api
kernel_api
Host
Java Runtime (system and app)
Android NDK libraries
Kernel syscalls
Zygote
Already uses host's APIs (libc, singletons) and OS primitives (threads, mappings)

Three ways to emulate NDK library


Guest code
app's guest libfoo.so
Emulation is partially in guest, and partially proxied to host
All emulation is done in guest
All emulation is proxied to host
guest libc.so
intercepted symbols
guest libm.so
guest libGLES.so (all symbols intercepted)
Host code
host libGLES.so
host libc.so

source link

Factors to decide which one to use for each lib

- Lean towards fully guest for fidelity and less maintenance cost
- Singletons in guest (jni) and native (java) may not co-exist (e.g. malloc)
- Library is hardware/driver specific (GLES)
- Performance (GLES)

Auxiliary libraries

The configuration makefile

1. **Fully guest library** just needs to be built for guest as if it was host
2. **Fully proxied library** has guest lib with specially cooked symbol stubs, and host proxy lib. Whenever a stub is invoked we add its address to TranslationCache with the corresponding proxy function as data.
3. **Partially proxied** is same as proxied, but part of guest symbols is not intercepted and are executed in guest code

How to proxy a function call?

- Analyze whether arguments and results are compatible or require conversion (struct layout, presence of function pointers)
- Convert arguments ABI from Guest to Host, and result ABI from Host to Guest (guest_abi, calling_conventions)
- The tools we developed collect and compare APIs compatibility between architectures, based on DWARF info in NDK libraries (extracted by tools/nogrod elf to json reader). Central point: **gen_proxy_libraries.py** (links to scripts TBD after we open-source them - expect by EOY)

Trampolines: Automatic and Custom

- Compatible trampolines or those needing only trivial conversions are generated automatically, others are required to be implemented manually (example of generated proxy lib code)
  - Compatible : `{"glAlphaFuncQCOM", GetTrampolineFunc<auto(uint32_t, float) -> void>(), reinterpret_cast<void*>(NULL)}`
  - Custom: `{"glGetPointervKHR", DoCustomTrampoline_glGetPointervKHR, reinterpret_cast<void*>(DoBadThunk)}`

Guest OS Primitives

The list

guest_map_shadow.cc
guest_signal_action.cc
guest_signal_handling.cc
guest_thread.cc
guest_thread_clone.cc
guest_thread_key.cc
guest_thread_manager.cc
guest_thread_map.cc
guest_thread_pthread_create.cc

These files need to be compiled with guest-specific headers:

guest_signal_action_arch.cc
guest_signal_handling_arch.cc
guest_thread_pthread_create_arch.cc

Repositories

Repositories

- frameworks/libs/native_bridge_support/
  - Configuration for guest loader and guest NDK libraries
  - Template configuration for proxy libraries (instantiated for specific translator, like berberis or ndk-translation)
  - Almost all NDK libraries are open-sourced
    - all except two, including libvulkan
    - also planning to open-source generating scripts
- frameworks/libs/binary_translation/
  - Everything else is here

Reusing API translation

Approximate list of tasks

- Disclaimer: some items are likely not listed
- Need implementations for runtime_library.h
  - `void RunGuestCall (GuestAddr pc, GuestArgumentBuffer * buf);`
  - `void ExecuteGuestCall (ThreadState * state);`
- If using alternative translation cache implementation, then also
    - `void InvalidateGuestRange (GuestAddr start, GuestAddr end);`
    - `void WrapHostFunctionImpl (HostCode func, TrampolineFunc trampoline_func , const char* name)`
- Define guest_state, guest_abi and calling_conventions for guest
- Generate configs for guest and proxy NDK libs
  - Need to open-source **gen_proxy_libraries.py**
  - Maybe implement some incompatible trampolines manually
- Implement guest_os_primitivies bits specific to guest arch