Automated Verification of RISC-V Kernel Code

Project Report

Antoine Kaufmann

June 10, 2016

1 Introduction

The longer term goal for this project is to investigate automated verification of an exokernel[1]/microkernel. Assuming hardware virtualization support on the CPU as well as I/O devices, very few operations actually need to be mediated by the kernel [3]. And the parts that do need to be executed as privileged kernel code perform simple operations. Most of those operations will interact closely with hardware, be it to modify CPU control registers, or to program an IOMMU. Because of this there is only very little code where a high-level language would provide any benefits for implementation or verification. Thus we choose to verify the kernel at the machine code level for this project, and for reasons of simplicity we selected the RISC-V instruction set architecture [4] as our target. Code paths through our kernel are generally simple consisting of few conditionals and no unbounded loops. This makes our kernel code a good candidate for automated verification techniques, and SMT solvers in particular.

For this class project we will limit ourselves to a simplified kernel as follows: All applications are already assumed to be loaded into memory prior to kernel initialization, and a kernel configuration describing where they are located is passed to the kernel in memory. The kernel only implements two system calls: `yield` for switching between processes, and `sbrk` for allocating and freeing memory. One important property that an operating system needs to provide to enable reasoning about end-to-end correctness of applications running on top of it is memory isolation. If an application cannot depend on its code and data remaining unchanged at run time, any verification of it's properties will become impossible or at least much more challenging. For the class project we aim to prove the following properties about the kernel:

1. *Initialization:* if a valid configuration is provided the kernel will boot to user space and start executing the first application. At this point the kernel's in-memory state corresponds to the configuration.
2. *yield syscall:* stores current state in kernel's state for the process, switches to next application, and restores next application's state.
3. *sbrk syscall:* if possible allocate additional memory to the application else return correct error code.
4. *memory isolation:* after boot and every system call, all application's memory remains isolated.

We chose the RISC-V architecture for this project because it provides the required functionality but as a clean slate approach also has a relatively compact specification. Also useful for this project in particular is that it provides simpler mechanisms for memory access control that are suited for embedded systems. In addition to regular virtual memory based on page tables, RISC-V provides simplified segmentation where just a single base address and limit are specified for the memory accessible to the application. To verify the kernel at a machine code level, we built a Z3 model for all required instructions, their encoding, and also control registers influencing execution.

The rest of this report is structured as follows: Section 2 presents the RISC-V CPU model. Then section 3 describes our kernel including its specification, implementation, and our verification effort and results. Finally section 4 discusses some possible future work.

2 RISC-V CPU Model

In order to be able to reason about kernel code at the machine code level, we need a model that captures how to execute those instructions. This section presents our CPU model in Z3, discusses how we validated it, and the model's current limitations.

RISC-V is a modular ISA, that consists of a base instruction set to which various extensions can be added, e.g. multiplication/division, floating point, or atomic memory operations. We specify the RV64-I instructions, which are the 64 bit core instructions. These consist of control flow instructions, memory operations, linear arithmetic, and bit operations. In addition we also implement parts of the RISC-V privileged architecture, that provides the necessary mechanisms, for running a kernel with multiple applications, including control transfers and memory protection. Our model currently only implements a subset of the privileged architecture. In particular we only implement the machine and user privilege levels and simple base and bounds memory protection, that can restrict user code to a contiguous segment of physical memory. This simpler memory protection scheme only requires two CPU registers and does not require parsing of page tables.

2.1 SMT Model

Below we describe our SMT model that captures the full machine state and how it is modified by instruction.

2.1.1 Symbolic Execution

For our first try we started out building a pure Z3 expression that, given the initial machine state, captures the full process of fetching an instruction, decoding it, executing it, represents the modified machine state. We then chained those expressions together to execute multiple instructions, while using Z3's simplify tactic to cut down the expressions to only reachable parts for scenarios where most of the state was concrete. For executing simple sequences of instructions without conditional branches from a fully concrete state this approach worked reasonably well, because Z3 can basically just evaluate the expression. However we quickly found for slightly more complicated instructions sequences that this approach often leads to huge expressions because simplify is not always able to decide conditionals in the expressions, even when Z3 itself actually has sufficient information to choose a branch. If this happens for the program counter, then building up the next step of instructions is then generally not able to simplify at all, leading to a huge expression catching all possible instructions, even if in fact only one instruction is possible. This results in huge performance problems, and in practice lead to Z3 no longer being able to prove even simple things about such concrete sequences of instructions.

Therefor we adapted our model to use a symbolic execution approach. So for all conditionals in the expression we use Z3 to determine which branches are reachable, and then only generating separate expressions for the reachable branches remembering the path condition for each. With this we basically build up expressions top down, and only actually build parts of the expression that will be needed for evaluation. This basically means that we

```
def split(self, cond, then_f, else_f,
    *args):
    assert self.split_cond is None

    # Check if we know which branch to
        take
    cc, sc = check_cond(cond, self.s)
    s = self.s
    if cc == True:
        then_f(self, *args)
    elif cc == False:
        else_f(self, *args)
    else:
        t_ms = self._copy()
        e_ms = self._copy()

        self.split_cond = sc
        self.split_then = t_ms
        self.split_else = e_ms
        self._mem = None
        self._cpu = None

        s.push()
        s.add(sc)
        then_f(t_ms, *args)
        s.pop()

        s.push()
        s.add(Not(sc))
        else_f(e_ms, *args)
        s.pop()
```

Figure 1: Method in our state class for generating conditional expressions. It takes a condition, the two functions that operate on the respective branches, as well as a set of arguments to pass to the branches.

```
def _condbranch(ms, instr, cond):
    pc = ms.cpu.read_pc()
    def then_f(ms_t, instr, pc):
        ms_t.cpu.write_pc(pc + instr.
            imm_sextx())
    def else_f(ms_f, instr, pc):
        ms_f.cpu.write_pc(pc + 4)
    ms.split(cond, then_f, else_f,
        instr, pc)
```

Figure 2: Function that implements handling for conditional jump instructions based on our state split mechanism.

are now no longer operating on a single expression representing the current state, but on a set of expressions capturing possible state, and then potentially splitting individual expressions whenever a non-decidable conditional occurs. Figure 1 shows the code in our state class for handling this splitting of the state based on which branches are reachable Unfortunately this splitting of expressions makes code somewhat harder to write, because separate functions for generating the individual branches need to be provided, and after splitting, the state can no longer be directly modified, but we provide a `do_with` method that applies a function to each child state recursively. Figure 2 shows an excerpt of our model for conditional jumps where the program counter is modified based on whether the branch is taken or not.

2.1.2 Machine State

When reasoning about code behavior we're really actually reasoning about possible machine states, i.e. what values CPU registers can contain, what the next instruction to be executed is, or values in memory. Thus our model is describes how executing instructions transforms the machine state. Figure 3 shows our Z3 definition of the machine state, it consists of the memory state, and the CPU state. Memory state is currently modeled as an array, and the CPU state consists of the program counter, general purpose registers, and a numbers of configuration registers (CSRs). When executing an instruction we start out with the initial machine state, and get the resulting machine state.

2.1.3 Instruction Execution

Executing an instruction means fetching the instruction from memory at the current program counter, then decoding it to determine which instruction it is, and finally based on that calculating the updated machine state.

RV64-I instructions are encoded in one of 6 formats, depending on what operands they take (immediates, source and destination registers). Some instructions are uniquely identified based on the opcode in the lowest 7 bits, but most instructions also require looking at additional bits in the instruction word to discover what instruction to perform. To express decoding in a somewhat readable manner, we specify the process as a two stage table. The first stage maps from the opcode to either an instruction handler or a second stage table. In the second stage we specify a set of instruction bits to match on, and then provide a match table that based on those bits maps to instruction handlers.

```
mem_sort = ArraySort(mach_bv_sort,
    byte_bv_sort)
cpustate_sort = Datatype('CPUState')
cpustate_sort.declare('cpu_state',
         ('pc', mach_bv_sort),
         ('regs', ArraySort(regidx_sort
             , mach_bv_sort)),
         # trap setup csrs
         ('csr_mstatus', mach_bv_sort),
         ('csr_mtvec', BitVecSort(xlen
             - 2)),
         # trap handling csrs
         ('csr_mscratch', mach_bv_sort)
             ,
         ('csr_mepc', BitVecSort(xlen -
             2)),
         ('csr_mcause', mach_bv_sort),
         ('csr_mbadaddr', mach_bv_sort)
             ,
         # Machine protection
         ('csr_mbase', mach_bv_sort),
         ('csr_mbound', mach_bv_sort),
         ('csr_mibase', mach_bv_sort),
         ('csr_mibound', mach_bv_sort),
         ('csr_mdbase', mach_bv_sort),
         ('csr_mdbound', mach_bv_sort),
         # Other privilege levels
         ('csr_sepc', mach_bv_sort),
         ('csr_hepc', mach_bv_sort),
         )
machstate_sort = Datatype('
    MachineState')
machstate_sort.declare('machine_state'
    ,
         ('mem', mem_sort),
         ('cpu_state', cpustate_sort))
```

Figure 3: Definition of the Z3 sorts representing the machine state.

```
@d_advance_pc
@d_store_rd
@d_read_rs1
def op_addiw(ms, instr, rs1):
    rs1 = Extract(31, 0, rs1)
    val = rs1 + instr.imm_sext(32)
    return SignExt(xlen - 32, val)
```

Figure 4: Instruction handler for the `addiw` instruction.

Our instruction handlers are written in an imperative manner, modifying the machine state, which is internally translated to building up new expressions for the machine state. We provide a number of python decorators that allow many of the basic expressions to be expressed fairly concisely. Figure 4 shows the handler for the `addiw` instruction that sign extends the provided immediate to 32 bits and adds it to the lower 32 bits of the source register, and then sign extends the result to 64 bits and stores it in the destination register. Here we used decorators to indicate that this instruction just advances the program counter, that the return value should be written to the destination register, and that the content of the source register should be passed as a parameter. Other instructions have more complicated handlers, e.g. instructions for writing the configuration registers do a range of permission checks and can behave differently based on what register is written.

2.2 Validation

To validate our model we use a set of test cases that is provided by the RISC-V project for validating RISC-V implementations: `https://github.com/riscv/riscv-tests` These test cases are just compiled to ELF binaries, and generally contain a few 10s or 100s of instructions. The only modifications we made were two smaller changes to the framework that is used for running the tests: 1) we change it to use the `mscratch` configuration register to indicate success or failure of the test instead of a vendor specific register, and 2) we removed two memory fence instructions because we do not support them.

We ran all the test cases for the instructions our model supports. The test cases for the supported RV64-I user space instructions all pass. And running those tests helped us catch a number of errors when moving to our symbolic execution based framework. The test cases also include a number of tests for the privileged architecture, but for most of those we currently do not support all the features that are tested.

2.3 Limitations

Our model for the RV64-I instructions is fairly complete. We are currently missing the two memory fence instructions, and instructions for reading the timer, cycles, and instructions retired counter. The latter group should be easy to add, modelling the former correctly will require adding an instruction or data cache.

For the privileged architecture our support is much less complete. We support the necessary mechanisms for traps, system calls, transfers between privilege levels, and base-and-bounds memory protection. The major features that are missing are modelling interrupts, support for supervisor protection level (we currently only have user and machine), page table based virtual memory, and support for delegating traps.

None of those missing features are currently required for our kernel. But especially support for page table based virtual memory would be interesting to add, and include in the verification.

3 Kernel

The following section first provides a rationale for our kernel design decisions, and next we discuss our specification. After this we provide a few comments on our implementation before moving on to our verification methodology.

3.1 Structure

Kernel structure has a significant impact on verification and verification effort in particular, e.g. a monolithic kernel includes much more functionality and generally has a much broader interface than a micro kernel. To be fair, a micro kernel based system providing equivalent functionality would need to provide a lot of functionality as user space services. But we would expect that verifying the same functionality as individual services would be significantly easier because of the modularity and because individual services can rely on the isolation guarantees provided by the kernel. Another reason why modularity would help make things easier is that services in a micro kernel could potentially be written in a higher level language. For a monolithic kernel this is often more difficult because it also needs to be able to manage low-level hardware details, and often has many parts where overheads incurred by higher level languages, e.g. garbage collection, are not acceptable. In a micro kernel individual services can easily be built using different languages and methodologies.

The seL4 project [2] was the first project to achieve full formal verification of a micro kernel implementation. But even though the micro kernel consists of relatively little code, they still report a verification effort of more than 20 person years. And while automated verification tools have made a lot of progress over the recent years, it seems unlikely that they would reduce this time to something reachable by a class project. We aim to reduce the burden by reducing kernel functionality even further, by opting for a kernel model that is closer to an Exokernel, which provides no abstractions to applications at all, but only a mere mechanism to enforce resource management decisions. The goal of this design is to keep only the minimal necessary functionality inside the kernel and push the rest of the functionality out to user space.

Next we will discuss an example of where an Exokernel makes verification easier compared to other approaches: A traditional kernel would provide a system call to request a range of memory of specified size. This requires kernel-data structures to track free memory and kernel code to maintain them. An exokernel on the other hand provides a system call to request particular memory pages. For simplification we assume no memory sharing is required. In this case all that the kernel needs to track for each physical page which application (if any) it is currently assigned to. So an allocation request just requires checking whether that particular page is currently unused, and then change the assignment, and then perform the necessary CPU operation to make the page accessible. This simplified operation is easier to verify in an automated fashion because all code paths are trivially bounded (no complex data structures needed).

3.2 Specification

As described earlier, our kernel provides memory isolation between applications using the RISC-V base and bounds mechanism. We also want the kernel to provide two system calls: `yield` for switching between applications, and `sbrk` for allocating and freeing memory. To provide this functionality the kernel needs to keep track of which applications are running, and what their current state is. The application state includes the contents of the CPU registers (x1-x31) and the program counter when the application was suspended, as well as the base address and length of the application's memory segment. And we refer to this state as valid if none of the memory segments overlap with each other or the kernel. This is our first invariant: the application state is always valid. To connect this reasoning to the concrete runtime in-memory state of the kernel, the second invariant that is required for all reasoning is that the kernel code and any read-only data remains unmodified.

The correctness proof of the kernel is primarily inductive: initialization establishes those invariants, and each of the steps, the two system calls and exceptions, preserve those invariants. All of those steps basically execute instructions until the CPU switches to user mode, and reason about the state then.

```
#define CONFIG_ADDR 0x800
#define CONFIG_MAX_APPS 8

struct config {
    uint64_t num;
    struct {
        uint64_t mem_base;
        uint64_t mem_bound;
        uint64_t pc_entry;
        uint64_t rsvd;
    } entries[CONFIG_MAX_APPS];
};
```

Figure 5: Definition of run-time configuration for kernel initialization. Up to 8 applications, each with memory segment and entry program counter.

Initialization After kernel initialization we expect the kernel to have read the configuration of applications to run, to have set up its internal state accordingly, to enter the first application, and to have established the invariants as discussed above. The actual in-memory configuration format is shown in fig. 5. As shown it includes the number of applications to start, and each application's memory segment and entry point, and the kernel expects to find it at address `0x800` in memory. So if the configuration is valid, i.e. there are no overlapping applications, the kernel state after initializing accordingly should also be valid. And, assuming that there is at least one application, the kernel should start executing the first application in user space. This includes setting up the base and bounds CPU registers, switching to user space, and jumping to the entry point program counter. Overall this leaves us with three theorems to prove:

- *Init-1:* The internal kernel state corresponds to the configuration i.e. there are the same number of applications, their program counters and memory segments match.
- *Init-2:* Given a valid boot configuration (non-overlapping), the resulting kernel state is also valid.
- *Init-3:* The read-only parts of the kernel in memory are still intact after initialization.
- *Init-4:* The CPU is executing in user space, has the first applications memory base and bounds set, and is executing at the entry program counter.
- *Init-5:* The kernel only modified its own writable memory.

`yield` syscall The yield system call should switch to the next available application or keep executing the same application if there is no other application. If it switches between applications, it should store the current application's state, i.e. it's register contents and program counter in the kernel state, so it can be restored. On a switch to another application, we also want to ensure the new application resumes executing at its last state. We also want to ensure that regardless of the outcome, the invariants are still preserved. Assuming the invariants hold when the system call is triggered we need to prove the following theorems:

- *Yield-1:* We store the current application's register state and program counter in the kernel internal application state. (for simplicity we do this regardless of whether we actually switch or not).
- *Yield-2:* If we switch to another application, we restore it's register state and program counter as well as memory segment from the kernel state. Otherwise the registers, program counter, and memory segment remain unmodified and the application continues executing.
- *Yield-3:* The two invariants are still valid afterwards.
- *Yield-4:* None of the other application's state is modified.
- *Yield-5:* The kernel only modified its own writable memory.

`sbrk` syscall The sbrk system call should extend or shrink the application's memory segment by the specified amount if this does not lead to overlapping with other applications. So if the segment can be modified both the kernel internal state for the application, and the CPU state should reflect the new segment limit. Otherwise the correct error code should be returned. So again assuming the two invariants from above hold before the system call we have to prove the following theorems:

- *Sbrk-1:* If the modification of the segment limit does not lead to overlap or a negative segment size, we extend the application's limit. Thus the kernel internal application state is updated accordingly, and the CPU bound register is also updated. Otherwise the right error code is returned.
- *Sbrk-2:* The register state (besides the error code) and PC are unmodified.
- *Sbrk-3:* The two invariants are still valid afterwards.

```
#define PCB_VALID 0x1
struct pcb {
    uint64_t sp;
    uint64_t regs[30];
    uint64_t pc;
    uint64_t mem_base;
    uint64_t mem_bound;
    uint64_t flags;
};
struct pcb pcbs[CONFIG_MAX_APPS];
uint64_t cur_proc;
```

Figure 6: Run-time kernel state: `pcbs` is an array of the process control blocks for each application, and `cur_proc` contains the process id of the currently running process (index into the array).

- *Sbrk-4:* None of the other application's state is modified.
- *Sbrk-5:* The kernel only modified its own writable memory.

Exceptions If an application causes an exception it is terminated, and another application is resumed, or if there is no other application the system is stopped. The theorems here are analogous to the yield system call, with the only difference being that the current application will be marked as inactive.

3.3 Implementation

The kernel was implemented in a mix of around 200 lines of C code, and around 150 lines of assembly. Most of the assembly parts are concerned with context switching, either saving registers and other application state to the internal kernel state, or restoring application state from internal kernel state. Figure 6 shows the definition of the internal kernel state. It consists of an array of process control blocks, one for each application, and a variable for holding the index of the currently executing application. The process control block stores the register state, program counter, memory segment, and a flag that indicates whether an application is valid or not. For simplifying context switching we also store a pointer to the current process control block in the CPU `mscratch` register (only accessible to the kernel).

3.4 Verification

Because of time constraints we have only formally verified theorems Init-1 through Init-4 so far. Below we describe the techniques we used for these. But we do believe the our current framework is sufficient for verifying most of the theorems for the system calls and exceptions. Proving Init-5/Yield-5/Sbrk-5 will possibly require some extensions to our model to be verifiable in reasonable time.

For reasoning about our kernel, we load the kernel ELF file into our model of the machine state. Depending on whether we are reasoning about induction steps (system calls or exceptions) or initialization we either skip the writable segments or load them. At initialization the kernel starts out with all it's memory as it was loaded by the boot loader, corresponding to the ELF file. But when processing a system call, previously executed kernel code could have modified the writable parts, e.g. the kernel stack, so we leave those parts abstract. In order to be able to reason about code that actually accesses and modifies the writable state in system call or exception handlers e.g. to access the process control blocks, we need preconditions that constrain the initial state.

Expression the theorems mentioned above that reason about the internal kernel state requires parsing the actual machine state, i.e. the bytes from memory. Figure 7 shows our python code for parsing the process control blocks. Based on those parsed values we can then write propositions to be verified by Z3.

For initialization we start with an abstract machine state, load the ELF file, and set parts of the CPU state as it is to be expected at boot, i.e. running in machine mode and executing at the reset program counter. We also write an abstract kernel configuration to memory. Then we have our model run up to 1024 instructions or until the CPU switches to user space. If it conclusively reaches user space we use the resulting state to check our Init theorems on. The init theorems check that the CPU state and parsed kernel state, as described above match up with the abstract kernel configuration written to memory. Verifying those theorems currently takes around 1 minute for a kernel supporting up to 8 applications.

```python
def load_state(ms):
    st = {}
    st['cur_proc'] = ms.mem.
        read_double(mach_bv_val(
        _cur_proc_addr))
    apps = []
    addr = _pcbs_addr
    for i in range(0, _MAX_APPS):
        app = {}
        app['regs'] = []
        for ro in _PCB_OFF_REGS:
            app['regs'].append(ms.mem.
                read_double(
                mach_bv_val(addr + ro)
                ))
        app['pc'] = ms.mem.read_double
            (mach_bv_val(addr +
            _PCB_OFF_PC))
        app['base'] = ms.mem.
            read_double(mach_bv_val(
            addr + _PCB_OFF_BASE))
        app['bound'] = ms.mem.
            read_double(mach_bv_val(
            addr + _PCB_OFF_BOUND))
        app['flags'] = ms.mem.
            read_double(mach_bv_val(
            addr + _PCB_OFF_FLAGS))

        apps.append(app)
        addr += _PCB_SIZE
    st['apps'] = apps
    return st
```

Figure 7: Parsing in-memory kernel state for use in theorems

4 Future Work

Below a list of options for continuing this project and suggestions for other future work:

- The most obvious next step is to complete the verification for the kernel as described. While there are possibly some issues with verification performance that might need to be addressed, we anticipate that our framework should be sufficient for proving the outlined properties.

- Extending the kernel to use additional hardware features, such as virtual memory instead of base-and-bounds memory protection, adapting the kernel specification and verifying the resulting kernel is the next obvious target and would be required for the kernel to be practically useful.

- Our model currently does not model interrupts, but the mechanisms for traps are implemented so only interrupt sources are missing. We expect that it is feasible to just turn off interrupts in the kernel, so this would not have a significant impact on verification.

- Another interesting option that could be achieved with the current framework is proving more abstract properties based on the specification. E.g. Currently we only verify that the kernel sets the base and bounds registers correctly, but it would be desirable to show that this actually implies memory isolation.

- Extending this approach to other somewhat more complicated architectures is another interesting direction, and would also likely be required for practical applications. ARM would be an interesting target, X86 would likely be much harder.

- Reasoning about the kernel state requires interpreting the memory contents and connecting them to the abstract state that theorems refer to. Currently we are manually parsing the memory contents, which is less than satisfying. It would be great to have some tool support to reason about the state at a higher level possibly based on the kernel source code and generating the code for parsing memory automatically.

- Finally, the set of test cases for RISC-V is somewhat limited. So to get additional confidence that the model is correct, it would be interesting to automatically generate test cases by generating instruction sequences and running them through both the model and a RISC-V simulator and then comparing the results. This would improve confidence in the correctness of the model.

References

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[3] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, Nov. 2015.

[4] RISC-V Foundation. RISC-V: The free and open RISC instruction set architecture. `http://riscv.org/`.