

# Respond to:#104, EuroSys 2021, "FirmGuide: Re-hosting the Linux Kernel of Embedded Firmware through Model-Guided Kernel Execution"

## Contents

<a href="#">1 Introduction</a>	<a href="#">1</a>
<a href="#">2 Reviewer B</a>	<a href="#">1</a>
<a href="#">3 Reviewer F</a>	<a href="#">3</a>
<a href="#">A An example of model templates</a>	<a href="#">4</a>

## 1 Introduction

Dear Reviewers,

We would like to sincerely thank you for providing us positive and inspiring comments that will significantly improve the paper's quality. In this extended response letter, we will answer the concerns that are not detailed in the short response, especially technical details. Reviewers' comments are in the numbered italic face, and our responses are in the following. The responses to Reviewer A, C, D, and E are absent because we think our short response letter has already covered their questions.

## 2 Reviewer B

**Comment 1:** *Section 2.1: "Whereas the processor can access RAM directly through the system bus, I/O peripherals are accessed ... through the Memory Mapped I/O (MMIO) mechanism." – As far as I'm aware, all MMIO reads and writes go directly through the system bus just like access RAM (which just sits at a different part of the address space).*

**Response:** From software's perspective, accessing MMIO regions are similar as accessing memorys. However, from QEMU's perspective, it needs to call the read/write callbacks to emulate the hardware's logic.

**Comment 2:** *Section 3.3: This motivating example is confusing. In Figure 3a, the IRQ mask function takes in the IRQ number as a parameter to the function. However, you treat this as a fixed value (3) in your example. Why can't this be a symbolic value itself?*

**Response:** We enumerate all possible interrupt numbers (we can get the interrupt number range from device tree) during the symbolic execution to get the patterns of the kernel-hardware interactions (read/write sequence). This means that by matching different read/write sequence, FirmGuide emulated interrupt controller could recognize IRQ number kernel wants to mask, i.e. which argument value is passed to the `irq_mask` function. Standing at kernel’s perspective, the symbol could only be introduced by MMIO read, i.e. value comes from the hardware.

**Comment 3:** *Also, these #define’d values (e.g., `INTC_REG_MASK`) are not usually the MMIO register physical addresses themselves, but rather an index into the device’s MMIO address region as specified in the device tree (which are then used as an offset for the `ioremap()`’d region).*

**Response:** We use the physical address in this example for simplicity. In practice, we hooked the MMIO read/write functions to capture the real physical address kernel accesses for a specific device. Therefore, it doesn’t matter whether the macro is defined as offsets or addresses.

**Comment 4:** *Section 4.1: Could you talk a little bit more of the format for the template? I know in the evaluation you cite some number of lines of C code for the two template models (interrupt controller and timer), but I have no sense for what these might actually look like. Are the transition condition functions linked in later based on the model parameters?*

**Response:** The template contains the state machine of INTC or TIMER. It is designed as a general framework and cannot run directly without configuration. To emulate a specific device, it needs to be configured with specific configurations, which are automatically extracted from the device driver’s code and used as transition conditions. In current implementation, the transition condition (configuration) is generated as c code. It is a set of static arrays which could be directly linked with the framework code and be compiled together to work as a specific INTC or TIMER device. We provide an example of the template in Appendix A.

**Comment 5:** *Section 4.2: Under “Parameter Inference”, what happens if there is no prior value for the `<MMIOR, addr, USE_LAST_VALUE>` R/W seq node? Is this going to use some fixed initial value or?*

**Response:** In this case, we use fixed initial value 0.

**Comment 6:** *Section 4: For the timer, is the notion of passing time captured in your manually-generated model?*

**Response:** Yes. The kernel will always first generate a time value in the unit of `cycle_t`. Then it will convert that value to a MMIO value (usually some bit shift operations like `y = x << 2`). FirmGuide first collects the conversion expression during the symbolic execution, and then tries to understand when the kernel wants the next time interrupt by converting the MMIO value back to `cycle_t` according to the inverse function of the expression (or constraint solver).

**Comment 7:** *Table 1: You note that the timer is “Not Necessary” for the first two subtargets, because it is actually already supported in QEMU. Why did you not generate the device model anyway and use it instead of the one provided by QEMU?*

**Response:** In the two “Not Necessary” cases, the QEMU MIPS CPU emulation code al-

ready contains these device's implementation and we cannot replace or disable them easily. The large difference between paths and solutions is caused by the difference in SoC's kernel code. One path means a possible boot path of an SoC and usually corresponds to one solution.

### 3 Reviewer F

**Comment 1:** *"During this process, for each MMIO read operation, ...": this really needs an illustration to make sense.*

**Response:** During the simplified boot process, we hook MMIO read/write functions. For each MMIO read, we tell KLEE to introduce a new read-from symbol. That's because the MMIO read operation is indeed a volatile memory operation that has no assumption to keep its value all the time.

**Comment 2:** *"After the execution ...": are you talking about concrete or symbolic execution here? Also, what is it you're executing here?*

**Response:** This execution means symbolic execution. We compile and link kernel's code to an LLVM IR file. Then we use KLEE to do symbolic execution on this IR file. We also add our main function into this IR file to control the code KLEE executes. KLEE will execute the kernel's boot process (`start_kernel` function). And during the KLEE's execution, the only place introduces new symbols is the MMIO read operation.

**Comment 3:** *"we may get several paths that can successfully finish the whole booting process": how do you measure success here?*

**Response:** A successful path must reach the end of the booting process without any error or early exit during the symbolic execution.

**Comment 4:** *"... are modeled as dummy MMIO memory space that accepts read and write operations.": What does accepting read/write operations mean?*

**Response:** It means that Type-II device's MMIO regions are emulated as RAM memory regions. No hardware specific logic is emulated and kernel could read/write it as normal memory.

**Comment 5:** *"Kernel first determines a value in cycle\_t ... Then it aligns the unit with the specific timer device.": What does "aligning the unit" mean?*

**Response:** Linux kernel and a real timer device use different units for time. Linux kernel uses the second(s) of which we are aware. The timer device use their smallest cycle (`cycle_t`) as unit, for example,  $1\text{M cycle\_t} = 1\text{s}$ . The Linux kernel accepts a command from us, say setting a timer in 1 second, then the Linux kernel will first convert (align) 1 s to 100M `cycle_t`. Once kernel wants to set a countdown value to a specific Timer device, it will always first generate a time value in the unit of `cycle_t`. Then kernel needs to convert that value to the value set to Timer's MMIO region which has equivalent meaning (usually the conversion are some bit shift operations like  $y = x \ll 2$ ). For example, if kernel wants to set 1s (100 `cycle_t`), it will set 25 (according to the example  $y = x \ll 2$ ) to the Timer. The "aligning the unit" represents for this unit conversion process.

## A An example of model templates

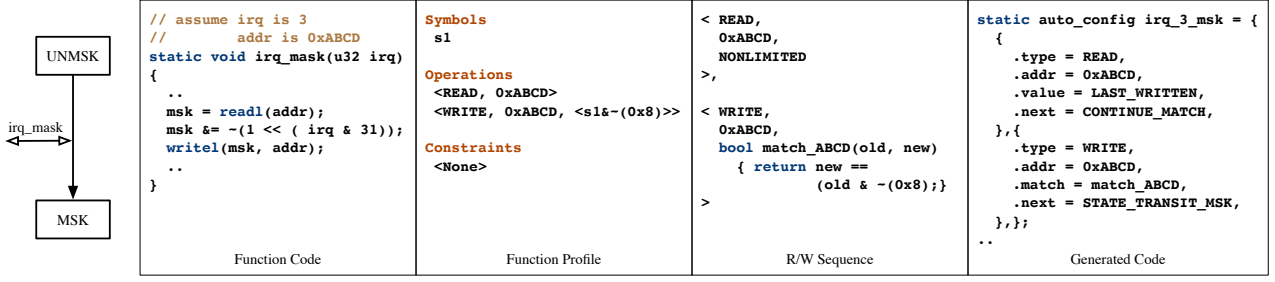


Figure 1: An example of using the static analysis to generate the transition condition. The figure takes the interrupt mask function as an example.

As we said in the response, the model template defines the states and the transition table of the device’s state machine. States and the transition table are manually built and fixed for one type of device, i.e. Interrupt Controller in this figure.

Figure 1 depicts the process of our model parameter extraction. We first use symbolic execution to explore the path of `irq_mask`. The `irq` argument is determined by the Timer’s interrupt source number. Next we extract the function profile recorded by modified KLEE, transform it to MMIO read/write sequence. And finally we generate the model parameters based on the sequence. The parameters are configuration arrays in C language. The combination of the parameters of the model template could be directly put into QEMU and emulate the corresponding devices.