Verified fault handling for modern board management controllers

Ben Fiedler[1][0000−0002−7215−9147], Zikai Liu[1][0009−0000−5411−9785], David Cock[1][0000−0003−2997−6560], and Timothy Roscoe[1][0000−0002−8298−1126]

ETH Zürich, Zürich, Switzerland

Abstract. Fault handling is the timely and crash-free response to critical changes in a system's operating characteristics, such as rapid temperature increases, or electrical shorts. In a typical computer system, it is the board management controller's job to correctly respond to such anomalous situations.
We develop an Isabelle/HOL model of a state machine for fault handling and define semantics for correctness of this procedure. Additionally, we formalize a notion of refinement that allows us to prove the correctness of implementations of this state machine. We also provide the first verified implementation of a C-based fault handler for board management controllers. Our implementation and the accompanying proofs are open-sourced and available online.
Furthermore, we successfully deploy our verified fault handler on top of the seL4 microkernel and alongside a production-grade, open source software stack widely deployed today, applying the cyber-retrofit approach to securing board management controllers in practice. The implementation and proof effort required is moderate, and our efforts indicate that already a small team of a handful of people can significantly raise the level of assurance of a modern, highly privileged software system.

Keywords: fault handling · applied formal methods · low-level systems verification · board management controllers

1 Introduction

Board management controllers (BMCs) are the hidden centerpieces of modern computers, and present a highly privileged interface to the lowest-level bits of hardware, such as clock, power, and fan control. Software running on the BMC is responsible for the safe operation of the platform, and often also for remote management. BMCs are critical pieces of hardware that *must* run correctly, otherwise the hardware can be irreversibly damaged.

Despite this, BMCs today are not built to be trustworthy and present a good candidate for the cyber-retrofit approach [24] to building high-assurance systems, by identifying critical functionality that can be extracted from a legacy system and verified or otherwise improved, while leaving the rest of the functionality untouched.

We apply the cyber-retrofit approach to the fault handling procedure of a BMC firmware image based on OpenBMC and Linux. *Fault handling* is the process of interpreting and responding to faults: hardware-generated events corresponding to a change

in the external system: over-temperature, over-current, under-voltage, and so on. This is a critical function of the BMC, and it must always run reliably. We verify the fault handling implementation written in C against an abstract specification written in Isabelle/HOL, proving that our implementation is functionally correct.

Our definition of functional correctness states that any received fault will be handled in bounded time and, furthermore, that occurrence of a *critical* fault will eventually lead to a system shutdown. Our proof uses established tools for verification of C programs in Isabelle/HOL, developed for the verification of the seL4 microkernel, but applies them in a new context.

Furthermore, we have deployed our implementation on a real hardware platform: a server-class research computer we describe in section 7. The fault handler runs as a component on the seL4 [23] formally verified microkernel, while the remaining legacy OpenBMC/Linux firmware runs isolated within a virtual machine (VM). Our source code and the associated proofs are available online on Zenodo [16].

2 Background

Most of the integrated circuits (ICs) in a modern server are connected by one or more I^2C buses [35], which provide a simple, two-wire protocol for transferring data between ICs and *bus controllers*. The System Management Bus (SMBus) [33] and Power Management Bus (PMBus) [38] standards define data transfer operations and semantics on top of the I^2C protocol, along with a procedure to report faults to bus controllers. The ICs which are networked with I^2C include the voltage regulators and clock distributors which provide power and clock to the rest of the platform.

The I^2C bus controllers are typically part of the BMC, a small computer which thus controls all aspects of the rest of the machine, including CPU and RAM. BMCs are also often network-facing, since remote management capabilities like console access and reset are indispensable for modern platform management.

A fault event is raised by an IC to signal a (possibly critical) change in operating characteristics, e.g. exceeding specified temperature, current, or voltage limits, and is signaled to the BMC generating an interrupt. The BMC must then identify which component faulted, what occurred, and what action must be taken. Figure 1 shows a simplified example of a server design, representative of the machine in section 7, with I^2C buses and the fault-handling signal path.

The faulting IC may itself take action. An over-voltage fault on a voltage controller, for example, might cause it to disable its output automatically, since I^2C communication between the IC and the BMC is too slow to safely react to these real-time events. Once a fault happens, however, the BMC must take appropriate action, for example increasing fan speed in response to a temperature fault. Ultimately, after a fault the BMC must decide whether to shut down the platform to prevent damage to critical components.

Ensuring BMCs are actually *trustworthy* is thus of critical importance. However, they are not built for high assurance [36]. BMCs are typically provided by the motherboard manufacturer as proprietary, closed-source components, making them difficult to inspect or modify. As a result, new exploits appear each year [9,10,11,12,13,14]. State-of-the-art BMCs today use open-source software like u-bmc [5] or OpenBMC

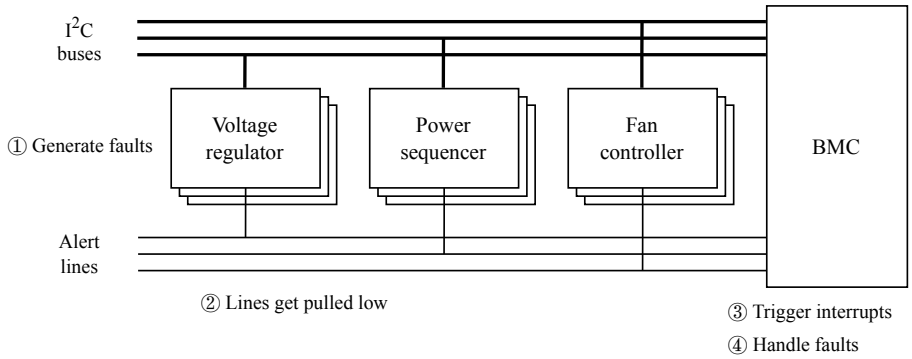

Fig. 1: Overview of the fault handling path on a typical server. I^2C-connected ICs generate faults by pulling the SMBus alert lines connected to the BMC, raising an interrupt.

[1], based on an embedded Linux kernel. These systems provide no rigorous guarantees, and the fact that the software must be customized for each board further reduces reliability and trustworthiness.

We improve on the state of the art by isolating a critical component, namely fault handling, from our system, providing an implementation with provably correct operation, and sandboxing the rest of the legacy system.

3 Related work

The cyber-retrofit approach [24] to incremental verification of mixed-criticality high-assurance systems was described by the seL4 [23] team in the context of the SMACCM component [7] of the larger DARPA HACMS project. The initial application was the isolation of an unmanned drone's mission-critical real-time flight-control software from a larger, untrusted Linux virtual machine providing lower-criticality functions. The specific approach has since been replicated in other aerial platforms [15], and falls within the wider category of static partitioning approaches [28] to building mixed-criticality systems [4], and has been cited as a reference point for future high-assurance systems for spaceflight [8]. We apply the technique to high-assurance firmware development in response to the shared characteristic of a mixed-criticality composition of safety-critical functions within a larger existing low-assurance code base.

Our proofs cover a subset of the guarantees established as part of the DARPA HACMS project. Compared to the work done by the seL4 team, our verification is limited to one component, thus we are not concerned with properties such as information flow control. We construct specifications for calls to inter-component interfaces, however we do not prove any properties of their implementations.

BMC software stacks have not been widely studied in the literature, and indeed it has been only a few years since open-source BMC software has reached significant adoption [17]. Other approaches for verifying low-level C code bases have been tried before, for example in sensor networks [3] using bounded model checking, or industrial

```
theorem faults_are_eventually_handled:
  defines tr :: "c_state stream"
  shows "alw step ⟶
           weak_scheduling_fairness ⟶
             alw (pending_fault f ⟶ ev (not (pending_fault f))) tr"

theorem critical_faults_eventually_lead_to_power_down:
  defines f :: c_fault and tr :: "c_state stream"
  assumes "is_critical_fault f"
  shows "alw step ⟶
           weak_scheduling_fairness ⟶
             alw (pending_fault f ⟶ ev powered_down) tr"
```

Listing 1: Top level Isabelle/HOL fault handler correctness theorems.

control applications [19] using deductive verification tools. Verification is usually done in full on the annotated source code, in contrast to our incremental approach. The scope of verification also varies greatly, from crash-freedom to functional verification.

4 The top-level result

We prove three core results of the overall fault-handling implementation. First, to discharge the well-formedness assumptions of the StrictC dialect we need to show that the C implementation is crash-free, terminates, and has the same core memory-safety and well-defined-behavior properties as the seL4 kernel itself. We then extend this with the two application-specific properties in listing 1.

These express that first, any fault that is signaled will eventually be handled and that second, any critical fault will eventually cause the system to be shut down. These are expressed as LTL (linear temporal logic) [32] formulae over the top-level state machine model of the system. The syntax is that of the existing LTL formalization available in Isabelle's HOL Library [30]. They correspond to the following formulae in standard LTL:

$$wsf \implies \qquad \Box(pending\_fault\ f \implies \Diamond(\neg pending\_fault\ f))$$

$$wsf \implies is\_critical\_fault\ f \implies \qquad \Box(pending\_fault\ f \implies \Diamond\, powered\_down)$$

"Assuming that weak scheduling fairness holds, it is always the case that if fault f is pending now, eventually it is not (and has thus been handled), and that if the fault is critical the system is eventually in a powered-down state."

Both statements make an explicit *weak fairness* assumption *wsf*. Lamport's definition of weak fairness for some transition T states that

$$weak\_fairness(T) \equiv \Box \Diamond \neg enabled(T) \vee \Box \Diamond\, step(T)$$

The transitions of the fault handling state machine are always enabled, and thus the weak fairness assumption simplifies to:

$$wsf \equiv \Box \Diamond\, consume\_faults \wedge \Box \Diamond\, check\_shutdown$$

```
definition enable_source where "enable_source s t = ..."
definition disable_source where "disable_source s t = ..."
definition receive_faults where "receive_faults s t = ..."

definition consume_faults :: "('s, 'f) state rel" where
  "consume_faults s t ≡ t = s⦇
      faults := [],
      shutdown_triggered := shutdown_triggered s
          ∨ (list_ex is_critical_fault (faults s))
   ⦈"

definition check_shutdown where "check_shutdown = ..."

definition step :: "('s, 'f) state rel" where
  "step s t ≡
     s = t ∨ enable_source s t ∨ disable_source s t
           ∨ receive_faults s t ∨ consume_faults s t
           ∨ check_shutdown s t"
```

Listing 2: The abstract state machine.

"At any point in time, both the fault and shutdown handlers will be called (strictly, be the current task) in a finite number of steps."

This liveness assumption is carried explicitly in both top-level correctness statements and expresses a requirement on the scheduler configuration in any deployment. This may be discharged by, for example, employing the strict real-time MCS scheduler [27] now undergoing verification.

By showing that the C implementation both refines the abstract state-machine model and separately preserves these liveness properties, we establish that they hold for the final deployed fault handler. We integrate this fault handler with the production-grade OpenBMC [1] software stack, a Linux distribution widely used for commercial BMCs. The fault handler comprises a set of native seL4 tasks running alongside the rest of the OpenBMC stack which is isolated within a virtual machine.

This is an application of the cyber-retrofit approach [24]. This technique increases the assurance of a software system by extracting a small, trusted part, comprehensively verifying it, and recombining it with the unmodified base system without compromising either the new formal guarantees or the functionality of the unverified component. Despite the presence of untrusted and unverified code in OpenBMC, we can appeal to the verified security properties of the seL4 microkernel to establish that these verified fault-handling guarantees hold even if the network-accessible OpenBMC component is completely compromised.

```
definition weak_scheduling_fairness :: "('s, 'f) state trace ⇒ bool" where
  "weak_scheduling_fairness ≡ alw (ev consume_faults) ∧
                              alw (ev check_shutdown)"
```

Listing 3: Isabelle/HOL definition of the weak scheduling fairness assumption; it is a straightforward translation of the previous LTL formula defining *wsf*.

```
record ('s, 'f) state =
  enabled_sources :: "'s set"    faults :: "'f list"
  shutdown_triggered :: "bool"   power_up :: "bool"

locale fault_handling_assumptions = fixes
  is_critical_fault :: "'f ⇒ bool" and
  max_faults        :: "nat"
```

Listing 4: Fault handler state and ancillary assumptions.

5 The abstract specification

Our top-level model is the nondeterministic state machine defined in listing 2 with next-step relation `step`. This expresses the nondeterministic composition of steps of the handler (`enable_source`, `disable_source`, `consume_faults`, `check_shutdown`), steps of the environment (`receive_faults`), and idle transitions (`s = t`). The definition of `consume_faults` is expanded to illustrate the abstract implementation of a component. Here the list of active faults in the prior state is cleared (consumed), and the shutdown flag is set if at least one of those faults was critical or it was already set.

While the model defines a set of possible next states (the image of the relation), any trace of the implementation is a definite sequence of states lying pairwise in the step relation.

Which trace actually executes is a property of the OS scheduler and any other nondeterminism refined by the final implementation. The OS scheduler and environmental nondeterminism determine the order in which steps are executed to form the final trace. This is abstracted in the model and the only property of the trace we rely on is expressed in the Isabelle/HOL formulation of the weak scheduling fairness assumption in listing 3.

The abstract state over which the machine operates is defined in listing 4. The types `'s` and `'f` are parameters and represent a set of *fault sources* and *fault types*, respectively. Fault sources are disabled, in which case the associated fault values will not be signaled, if they are not in the set of `enabled_sources`. The list of `faults` records all as-yet-unhandled faults in order of occurrence. The flag `shutdown_triggered` records whether a shutdown has been initiated (but not necessarily completed), and `power_up` whether the system is currently powered. Once the shutdown completes, `power_up` becomes false.

A *locale* is an Isabelle mechanism to express a named bundle of definitions and assumptions [22]. The `fault_handling_assumptions` locale assumes the existence

```
definition refines ::
  "('c ⇒ 'a) ⇒ 'a rel ⇒ 'c rel ⇒ bool" where
  "refines lift an cn ≡
     (∀s t. cn s t ⟶ an (lift s) (lift t))"
```

Listing 5: Refinement with respect to a lifting function

```
definition sufficiently_live ::
  "('c ⇒ 'a) ⇒ ('c stream ⇒ bool) ⇒ bool"
where
  "sufficiently_live lift LP ≡
     ∀tr. LP tr ⟶ weak_scheduling_fairness (smap lift tr)"

definition liveness_prop ::
  "(('s,'f) model stream ⇒ bool) ⇒ bool"
where
  "∀tr. (alw step' ⟶ weak_scheduling_fairness ⟶ P) tr"

lemma transfer_liveness_prop:
  assumes
    "refines lift step cn"
    "sufficiently_live lift LP"
    "liveness_prop P"
  shows
    "(alw (lift cn) ⟶ LP ⟶ P o (smap lift)) tr"
```

Listing 6: Liveness preservation across refinement

of an otherwise-undefined predicate classifying some faults as critical (requiring shutdown), and that there exists some bound (`max_faults`) on the number of faults that can occur before the handler executes.

We show that the C implementation of the fault handler, as imported into Isabelle using the StrictC tool chain, is a *refinement* [26] of this abstract state machine. This relation is defined precisely in listing 5. The *concrete* system `cn` is a refinement of the *abstract* system `an` if, given a lifting function from concrete to abstract states, wherever a concrete transition between two states exists, an abstract transition exists between the corresponding lifted states. Informally, this expresses that "Every behavior of the (concrete) implementation is permitted by the (abstract) specification.".

Refinement does not generally preserve liveness. An implementation which does nothing at all trivially refines any specification. More generally it only requires that the steps of the implementation are some subset of those of the specification, but not that any particular step actually happens. We thus extend our definition as shown in listing 6 to express that a class of liveness properties (those that depend on weak scheduling fairness) are in fact preserved by our refinement.

A *sufficiently-live* property on the implementation is one which if it holds for a concrete trace, weak scheduling fairness holds for the corresponding lifted abstract trace. A *liveness property* (in this context) is one which holds in any (abstract) state where weak scheduling fairness holds.

The *transfer lemma* establishes that given a sufficiently-live property for the concrete system, any liveness property of the abstract system also holds for the concrete when interpreted through the lifting function. The property may be *transferred* from the abstract states to their pre-images under the lifting function. Both of the top-level formulae of listing 1 satisfy this definition of liveness, and are thus preserved by the refinement.

6 The proof

Having defined an abstract model for fault handling, we next implement a fault handler in C as a component on top of the seL4 microkernel [23] and its component framework CAmkES [25]. We provide a behavioral proof of each of the fault handler functions in the form of Hoare triples, as well as total correctness (i.e. termination) proofs for invariant preservation. A brief schematic of the whole code-to-proof pipeline from the C source to the top-level result is shown in fig. 2.

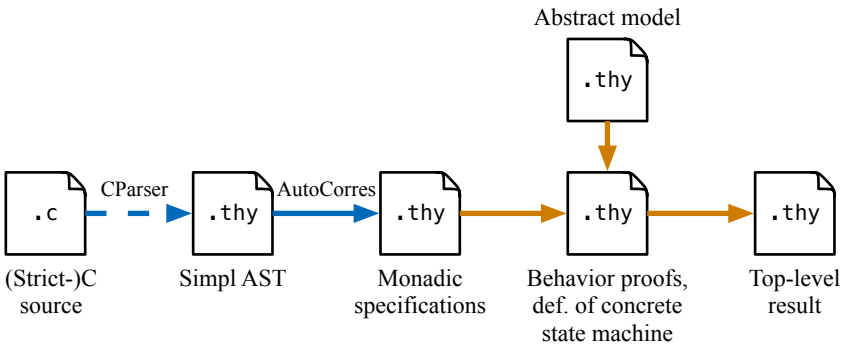

Fig. 2: Overview of the proof pipeline, from C code to final result. The dotted arrow denotes an unverified step, while the solid arrows are formally proven in Isabelle/HOL. Blue and orange distinguish automatic and manual proofs, respectively.

Our code is written in StrictC, a dialect of the C language developed by Schirmer [34]. As the name suggests, StrictC disallows use of some C constructs, including but not limited to: `goto` statements, assignments within expressions, fall-through cases, type unions, and C99's `bool`. We translate the StrictC program to the Simpl language using an existing, ML-based parser, initially developed for use in the verification of the seL4 microkernel. Simpl is a sequential, imperative programming language with formal semantics defined in Isabelle/HOL. Note that this translation is unverified; there is no

```
lemma inv_no_fail:   "{inv} m {inv}!"
lemma executable:    "executable_specification m"
lemma refines_model: "{s} m {model.step (lift_state s) (lift_state s')}"
```

Listing 7: Proof obligations for an AutoCorres-generated monadic specification *m*

formal proof that the semantics of the generated Simpl program matches the semantics of the StrictC program.

The Simpl translation of our C code is then fed into AutoCorres, which was developed by Greenaway et al. [18]. AutoCorres translates the Simpl programs to a monadic specification over a non-deterministic state monad with failure, including an automatically derived proof that these specifications refined the Simpl programs they were generated from.

The monadic specifications generated by AutoCorres contain statements that relate to the well-formedness of the underlying StrictC program: accessed pointers are always valid, satisfy bounds checks when indexing arrays, and so on. Otherwise, the program fails, and the resulting state will have a failure flag set. Showing that the monadic programs do not fail is a significant part of our work. On the other hand, a monadic specification could also be *trivial*, returning an empty set of resulting states. This would imply that said specification cannot be executed. Fortunately, we can automatically prove non-triviality for almost all of our specifications, and complete the remaining cases by hand.

Our main proof effort pertains to proving Hoare triples about the behavior of these monadic specifications. From these behavioral predicates, we construct a state machine, where the state chosen corresponds to the state of the monadic translation of StrictC program. We prove that each monadic specification, translated from its original function, refines a corresponding transition in the abstract state machine. Finally, we assume the existence of a weakly fair scheduler for some transitions, in order to meet the liveness assumptions made about our state machine.

Plugging our refinement results, proven on the abstract machine, together with the concrete behavior proved on our monadic translations, allows us to prove that the concrete machine satisfies the conditions for fault handling identified in section 4. Adding the refinement result generated by AutoCorres relates the concrete state machine's behavior down to the StrictC code.

6.1 Reasoning about the generated monadic specifications

Recall that the monadic specifications generated by AutoCorres are non-deterministic and allow failure, thus we have to prove that for each specification, it never fails *and* always produces at least one resulting state. Thus, after translating a C function f to a monadic specification m, we need to prove three things of m: it terminates and does not fail, it is executable, and it refines the behavior of an abstract transition. We prove each of these statements individually, later combining them to a full correctness statement. A brief overview of the proof goals can be seen in listing 7.

In order to prove that the specification does not fail, we need to show that any pointers and arrays accessed/indexed by m are valid. To this end, we define a global

```
definition "valid_num_faults g ≡ num_faults g ≤ max_faults"
definition "valid_ptrs g ≡
              is_valid_w8 addr_ptr ∧
              is_valid_w16 status_word_ptr ∧
              (* ... *)"
definition "inv s ≡ valid_ptrs s ∧ valid_num_faults s"
definition "init s ≡ valid_ptrs s ∧ num_faults s = 0 ∧ ..."
lemma init_impl_invariant: "init s ==> inv s" (* ... *)
```

Listing 8: The main fault handler invariant for the concrete implementation.

invariant `inv` that is preserved by every monadic program, and it does not cause the program to fail. Listing 8 shows an excerpt of our invariant definition.

Proving that the pointer invariants are preserved is straightforward, since our pointers are valid initially, and are never modified. Proving that the number of faults received stays valid is slightly more involved, since we use `num_faults` as an index when modifying the `max_faults`-sized buffer of faults. We utilize lemmas for reasoning about word-sized arithmetic in these cases. Most of the proofs can be discharged by the `wp` tool.

Next, in order to prove that a monadic specification is non-trivial, we have to prove that the set of result states it produces is non-empty. However, doing so would require us to duplicate all proofs we have already done about total correctness of the invariant. Therefore, we prove an equivalent statement, which allows us to re-use our invariant proofs: we prove that each monadic specification also fails when it produces an empty set of output states. Since we have already proven non-failure under the invariant, this is sufficient to prove executability as well. This property can be derived automatically in almost all cases.

Finally, to prove that our specification refines our abstract model, we define a lifting function, which lifts our concrete state to the abstract model state and is shown in listing 9. The `lift_state` function is responsible for translating the data representation from C arrays and integers to the richer type system of our state machine using richer types such as lists and sets. Concretely, the `faults` list of our abstracted state is represented as two variables in our C code: a statically-sized array of faults, and an integer recording the number of faults currently buffered. The set of enabled sources is related to the non-zero entries of underlying C array. The boolean field holding whether a shutdown was triggered is represented as an integer in the underlying C code[1], with zero corresponding to *false*, and any non-zero value corresponding to *true*. Finally, the abstract machine is considered "powered up" when the main power supply is switched on. Other powered components, such as the passive power rail that powers the BMC itself are irrelevant.

To complete our refinement proof, we need to prove that each monadic specification modifies the state in a way consistent with the abstract model. These proofs boil down to proving that some fields have been modified in a specific way, while leaving most of

[1] This is due to the StrictC parser being unable to parse the `stdbool.h` header.

```
definition
  lift_state :: "c_state ⇒ (fault_line, fault) Model.state"
where "lift_state g = (|
    enabled_sources    = {s. faults_enabled g.[unat s] ≠ 0},
    faults             = let n = unat (num_faults g)
                         in take n (list_array (faults g)),
    shutdown_triggered = shutdown_triggered g ≠ 0,
    power_up           = psu_up (power_state g)
 |)"
```

Listing 9: Definition of our lifting function from `c_state` to abstract model state

the other fields untouched. Take the `receive_fault` transition, for example: we prove that most of the global state remains unchanged, while the number of received faults is increased, and new faults are correctly buffered, keeping the old buffer intact. Thus, we can decompose our proofs about the whole state by considering each field separately, leveraging `wp`'s proof automation to combine the sub-proofs for us.

An example of the individual proof goals for a function receiving faults is shown in listing 10. Note that in a system with multiple fault sources, there might multiple functions that receive faults, and each of them have to be proven correct separately. Generally, we divide the proofs in two parts, depending on whether we aim to prove a *framing condition* about a particular field, or whether we prove something about the correct modification of some state.

Many of our proof obligations are, in fact, such framing conditions. Those can often be proven automatically using some existing tooling that "crunches" through these (often repetitive) proofs. It was even able to deal with simple loops, which alleviated a non-trivial amount of proof effort. In more complicated cases, however, we had to write proofs by hand.

The behavioral proofs seldom worked out-of-the-box. In order to get those proofs down to a combination of applying the `wp` tactic and appropriate simplifier invocations, we introduce and prove other Hoare triples or simplification rules. Sometimes, the form of required rules could be straightforwardly derived from the proof state, indicating where the weakest-precondition tactic was "stuck". However, in other cases the proof state was unhelpful in determining a statement that could be useful to the automated tactics, and required us to carefully construct a proof by hand.

As already mentioned, we rely heavily on the weakest precondition tactic `wp`, as distributed with the nondeterministic state monad, for verification condition generation. In practice, we found it often not necessary to find the weakest precondition, only a sufficiently weak one. Supplying the tool with theorems that are not in weakest precondition form still guarantees soundness, but completeness is lost: it might be the case that some valid goals are not provable, and in those cases we had to re-visit some intermediate proofs we had done. We used this fact in some of our proofs: by supplying non-weakest preconditions, we were able to reduce the precondition's complexity and simplify the remaining proof. This was especially useful in statements that involved branching.

```
(* Framing conditions *)
lemma receive_fault_lift_enabled:
  "{%s. P (lift_enabled s)} receive_fault {%s. P (lift_enabled s)}"
lemma receive_fault_lift_shutdown:
  "{%s. P (lift_shutdown s)} receive_fault {%s. P (lift_shutdown s)}"
lemma receive_fault_lift_power_up:
  "{%s. P (lift_power_up s)} receive_fault {%s. P (lift_power_up s)}"

(* Precise behavior *)
lemma receive_fault_lift_faults_prefix:
  "{%s. prefix a (lift_faults s)}
     receive_fault
   {%s. prefix a (lift_faults s)}"
lemma receive_fault_line_disabled:
  "{%s. P (lift_faults s) ∧ l ∉ lift_enabled s}
     receive_fault
   {%s. P (lift_faults s))}"

lemma receive_fault_combined:
  "{%s'. s = s'}
      receive_fault
   {%s'. model.step (lift_state s) (lift_state s')}"
```

Listing 10: Proof obligations split into framing and non-framing conditions.

Finally, we construct our state machine by defining a suitable initial state predicate and transition relation, and prove that, together with the lifting function defined earlier, this concrete state machine is a refinement of the abstract state machine introduced in section 5, which concludes our proof.

From here on, state the liveness assumptions on the concrete state machine like we did on the abstract state machine, and prove that it is indeed sufficient to transfer the properties proved on our abstract state machine. This is shown in listing 11. Together with the previously proven facts about the abstract, we achieve the main result shown in section 4.

7 Running on real hardware

Enzian [6] is a server-class research computer built at ETH Zurich with two main chips, a CPU and a large FPGA as a second NUMA node. The BMC is a Zynq UltraScale+ MPSoC [2], initially running Linaro Linux and a custom distribution of OpenBMC [1] for board management.

The relevant system architecture is shown in fig. 3. Enzian uses multiple PMBuses equipped with alert signals to communicate with the peripheral devices. We focus on two interesting voltage regulator ICs, the IR3581 [20] and the MAX15301 [29], in this presentation. Applying the cyber-retrofit approach, we encapsulate the Linux-based

```
definition step :: "(c_state, unit) nondet_monad" where
 "step ≡ skip <|> receive_fault <|> enable_faults <|> disable_faults
             (* other fault lines omitted *)
             <|> consume_faults <|> condition lift_shutdown shutdown skip"

theorem step_refines_model:
 "model.refines lift_state model.step (run_monad step)"

definition weak_liveness :: "c_state trace ⇒ bool" where
  "weak_liveness ≡ alw (ev (run_monad consume_faults')) aand
                   alw (ev (run_monad check_shutdown'))"

lemma sufficiently_live_assumption: "sufficiently_live weak_liveness"
```

Listing 11: The main refinement theorem of our abstracted code.

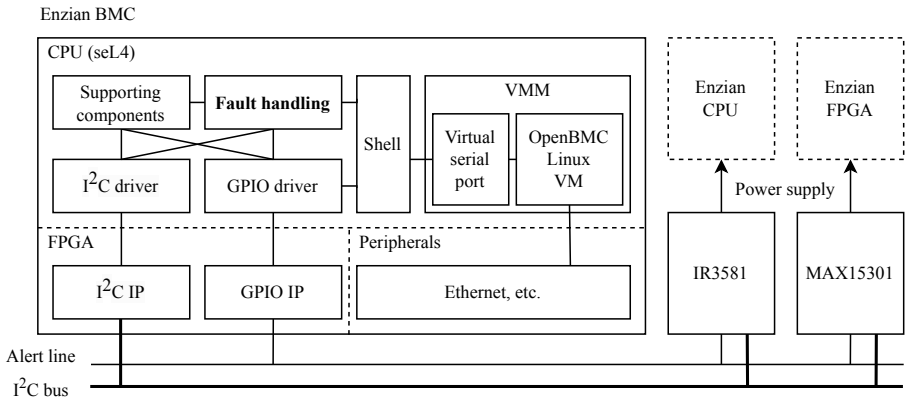

Fig. 3: System architecture.

OpenBMC in a VM over seL4 (with device pass-through) and implement the fault handling code in a native seL4 process using the CAmkES component framework.

The code requires device drivers to communicate with the regulators. Unlike Linux, limited drivers are available on seL4. For ease of implementation, we use Xilinx driver IPs on the MPSoC FPGA [2] and ported the Xilinx bare-metal drivers [40] to seL4. Currently these components are trusted, and verifying their correctness is beyond the scope of the present work. The fault handler runs in the background; we also implemented an interactive shell for testing. We modify the original OpenBMC to redirect commands to those two regulators to a virtual serial port connected to the native seL4 system. The supporting OpenBMC components include a time server and a board resetting server.

The IR3581 and MAX15301 raise alerts when temperatures exceed set values. To simulate over-temperature alerts reliably, we set the threshold temperatures to *below* room temperature. These alerts are critical faults that should lead to system shutdown.

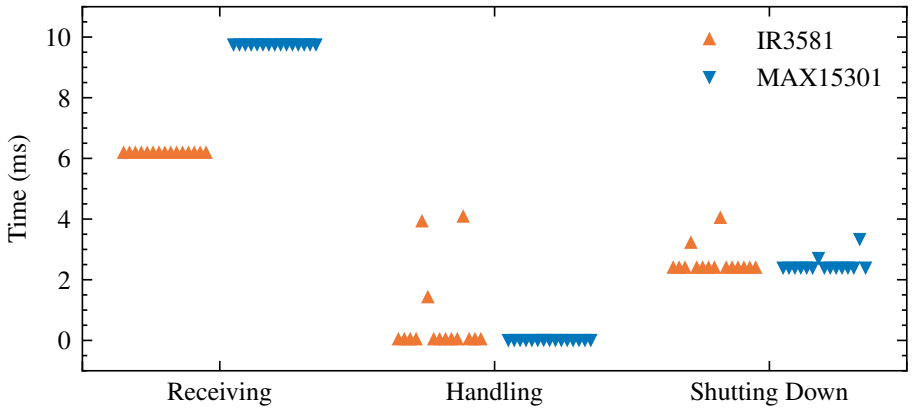

Fig. 4: Time spent in steps of fault handling.

The evaluation process thus consists of the following steps, performed through the interactive shell:

1. Power up the system.
2. Read back voltage levels to ensure the system is indeed powered up.
3. Trigger alerts by lowering the threshold temperature. The fault handling code reports time spent in each step to the shell.
4. Read back voltage levels to check if the system is powered down.

In step 3, the SMBus protocol [33] requires the BMC to send out a response-address request on the I^2C bus to identify the faulting device. The device then responds with its address. This step, when receiving faults, involves relatively slow I^2C communication. The same applies to the step of shutting down the system. Time spent during I^2C communication is not a focus of this work. Nevertheless, we show the time of those two steps to briefly demonstrate the end-to-end functionality of the code, while focusing on the step of processing fault events.

For each device, we repeat the procedure 15 times. Lowering the threshold temperatures results in alerts raised by the IR3581 [20] and MAX15301 [29], respectively. Read-back voltages are machine-checked, showing that the fault handling *functions as expected*. The time spent in each fault handling step is reported in fig. 4.

The fault handling time (middle) is relatively small. Handling one alert from the MAX15301 takes only 25.6 μs on average with a standard deviation of 0.7 μs. For the IR3581, most of the time, handling two alerts takes less than 26.5 μs. However, we observe a few outliers that take up to 4.1 μs. We believe this is due to seL4 switching context between the two alerts. The fault handling and shutting down processes are connected by seL4 signals [39]. When the first alert gets handled, a signal is emitted, which may trigger the seL4 scheduler to deschedule the fault handling code.

Nevertheless, the fault handling code functions correctly on the evaluation platform and in a timely fashion. The outliers for the IR3581 demonstrate how the OS scheduler

may affect the liveness of fault handling, motivating a verified mixed-criticality scheduler or careful design in the implementation to better match seL4's default fixed-priority scheduler.

8 Experience

We now briefly discuss our experience replacing the OpenBMC fault handler with our verified implementation.

The entire project, from developing the initial implementation, specification, integrating our code with virtualized OpenBMC, and proving the implementation correct took us about 20 person-months (shared across several people). A detailed breakdown of the individual tasks can be seen in table 1.

Task	**Effort**
Initial seL4 port	6 mo
verified C	2 mo
Abstract model	1 mo
Full verification	7 mo
Virtualizing OpenBMC and integration	4 mo
Total time spent	20 mo

Verified Component	**LoP**	**%**
Top-level statement	167	7.0
Abstract model	354	14.8
Extracting monadic specification from C	43	1.8
Proofs about monadic specifications	1'733	72.4
Miscellaneous	96	4.0
Total	2'393	100.0

Table 1: Overview of time and lines of proof required for the project. We do not include the existing seL4 components we use.

In total, our efforts clock in at about 7.4k lines of C code (excluding generated files), and 2.4k lines of Isabelle/HOL. Of those 7.5k lines of C, the verification covers about 1k lines, of which 700 lines are source code and 300 lines are headers. Unsurprisingly, the majority of the effort is spent verifying the C code, both in time spent and the proof length.

Verification strongly influenced the implementation during development, for a variety of reasons. On the one hand, writing code that we later verify forces us to be precise with respect to our intended semantics, such as the consideration of what happens when the fault buffer is full. On the other hand, the restrictions placed on us by StrictC disallows some constructs that could come in handy at certain points when interacting with hardware, such as type unions. Finally, it causes us to use language features that alter control flow (such as `continue` or `break` within loops) only very sparingly. It is possible to reason about the specifications for such code, however we found it is often easier to restructure the control flow.

Our proofs themselves are designed to be modular. Changes to the implementation of one function generally have no effect on other functions, assuming we can re-use the

same top-level specification. We took care to design our proofs such that some assumptions can easily be changed, such as the definition of a critical fault or the maximum number of buffered faults.

Now that we have verified one component of a larger system, the question arises how to compose our proof with other correctness proofs that we might be interested in developing.

Due to limitations in the StrictC tool, we were forced to specify some C functions, such as those that interact directly with the I^2C hardware through registers. Since we have already come up with these specifications, further proof efforts should be able to leverage these.

However, composing our proofs with correctness proofs of other components that would concurrently execute within the same system is much more difficult, as there is no general method known to compose two unrelated, concurrent specifications.

9 Conclusion

We have shown that the cyber-retrofit strategy can be applied to new domains in order to verify large-scale system. We were able to apply existing verification tooling and techniques to develop and prove correct the fault handling implementation of a modern BMC, while deploying the resulting artifact in production and successfully handling faults.

The required work for writing and proving correct code is sufficiently small to allow even small teams of students to verify non-trivial functionality, thanks to the verification tools developed by the seL4 team [23].

There are open questions for scaling the work in order to verify larger systems. Cross-CAmkES component verification was shown to be possible, and new approaches [31] are being tested to verify sequential code across process boundaries. It is, however, unclear how concurrency fits into the picture. Approaches for verifying concurrent code are being tried right now [21,37], and it remains to be seen how the seL4 team itself will tackle these. We assume a sequential schedule for now.

Future work on our end includes extracting and verifying other parts of the OpenBMC stack, and ensuring the specifications we developed for inter-component function calls. Additionally, we are looking forward to a version of seL4 with a verified mixed-criticality scheduler in order to formalize our liveness assumptions.

Acknowledgments. We thank our anonymous reviewers for their valuable feedback, and helping us to improve the presentation of this paper.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. OpenBMC, https://www.openbmc.org/

2. AMD: Zynq UltraScale+ Device Technical Reference Manual. Tech. Rep. 2.4 (2023), https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm
3. Bucur, D., Kwiatkowska, M.: On software verification for sensor nodes. Journal of Systems and Software **84**(10), 1693–1707 (2011)
4. Burns, A., Davis, R.I.: A survey of research into mixed criticality systems. ACM Computing Surveys **50**(6) (nov 2017). https://doi.org/10.1145/3131347, https://doi.org/10.1145/3131347
5. Christian Svensson et al: Open-source firmware for your baseboard management controller (BMC) (2018), https://github.com/u-root/u-bmc, computer Software
6. Cock, D., Ramdas, A., Schwyn, D., Giardino, M., Turowski, A., He, Z., Hossle, N., Korolija, D., Licciardello, M., Martsenko, K., Achermann, R., Alonso, G., Roscoe, T.: Enzian: an open, general, CPU/FPGA platform for systems software research. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 434–451. ASPLOS '22, Association for Computing Machinery, New York, NY, USA (Feb 2022). https://doi.org/10.1145/3503222.3507742, https://dl.acm.org/doi/10.1145/3503222.3507742
7. Cofer, D., Backes, J., Gacek, A., DaCosta, D., Whalen, M., Kuz, I., Klein, G., Heiser, G., Pike, L., Foltzer, A., Podhradsky, M., Stuart, D., Grahan, J., Wilson, B.: Secure mathematically-assured composition of control models. Tech. Rep. AFRL-RI-RS-TR-2017-176, Rockwell Collins, Cedar Rapids, United States (Sep 2017), https://apps.dtic.mil/sti/citations/AD1039782
8. Curbo, J., Falco, G.: A research agenda for space flight software security. In: 2023 IEEE 9th International Conference on Space Mission Challenges for Information Technology (SMC-IT). pp. 68–77 (2023). https://doi.org/10.1109/SMC-IT56444.2023.00016
9. Database, M.C.: CVE-2022-40242 (Dec 2022), https://www.cve.org/CVERecord?id=CVE-2022-40242
10. Database, M.C.: CVE-2022-40259 (Dec 2022), https://www.cve.org/CVERecord?id=CVE-2022-40259
11. Database, M.C.: CVE-2021-39295 (Apr 2023), https://www.cve.org/CVERecord?id=CVE-2021-39295
12. Database, M.C.: CVE-2022-26872 (Jan 2023), https://www.cve.org/CVERecord?id=CVE-2022-26872
13. Database, M.C.: CVE-2023-31008 (Jan 2024), https://www.cve.org/CVERecord?id=CVE-2023-31008
14. Database, M.C.: CVE-2023-31037 (Jan 2024), https://www.cve.org/CVERecord?id=CVE-2023-31037
15. Farrukh, A., West, R.: Flyos: Integrated modular avionics for autonomous multicopters. In: 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 68–81 (2022). https://doi.org/10.1109/RTAS54340.2022.00014
16. Fiedler, B., Liu, Z.: An sel4-based board management controller with verified fault handling for the enzian research computer. https://doi.org/10.5281/zenodo.12775411
17. Frazelle, J.: Opening up the baseboard management controller. Communications of the ACM **63**(2), 38–40 (2020)
18. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: formal verification of c code without the pain. SIGPLAN Not. **49**(6), 429–439 (jun 2014). https://doi.org/10.1145/2666356.2594296, https://doi.org/10.1145/2666356.2594296
19. Huisman, M., Monti, R.E.: On the industrial application of critical software verification with vercors. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9. pp. 273–292. Springer (2020)

20. International Rectifier: Dual Output Digital Multi-Phase Controller IR3581. Tech. rep. (Apr 2014)
21. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 637–650. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2676726.2676980, https://doi.org/10.1145/2676726.2676980
22. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - a sectioning concept for isabelle. p. 149–166. TPHOLs '99, Springer-Verlag, Berlin, Heidelberg (1999)
23. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems **32**(1), 2:1–2:70 (Feb 2014). https://doi.org/10.1145/2560537, https://dl.acm.org/doi/10.1145/2560537
24. Klein, G., Andronick, J., Kuz, I., Murray, T., Heiser, G., Fernandez, M.: Formally verified software in the real world. Communications of the ACM **61**, 68–77 (Oct 2018). https://doi.org/10.1145/3230627, https://trustworthy.systems/publications/full_text/Klein_AKMHF_18.pdf
25. Kuz, I., Liu, Y., Gorton, I., Heiser, G.: CAmkES: A component model for secure microkernel-based embedded systems. Journal of Systems and Software **80**(5), 687–699 (May 2007). https://doi.org/10.1016/j.jss.2006.08.039, https://www.sciencedirect.com/science/article/pii/S016412120600224X
26. Lamport, L.: Specifying concurrent program modules. ACM Transactions on Programming Languages and Systems pp. 190–222 (4 1983), https://www.microsoft.com/en-us/research/publication/specifying-concurrent-program-modules/
27. Lyons, A.: Mixed-Criticality Scheduling and Resource Sharing for High-Assurance Operating Systems. Phd thesis, University of New South Wales (Sep 2018), https://trustworthy.systems/publications/papers/Lyons%3Aphd.pdf
28. Martins, J., Pinto, S.: Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems. In: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 40–53 (2023). https://doi.org/10.1109/RTAS58335.2023.00011
29. Maxim Integrated Products, Inc.: MAX15301 InTune Automatically Compensated Digital PoL Controller with Driver and PMBus Telemetry. Tech. rep. (Nov 2013)
30. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer-Verlag, Berlin, Heidelberg (2002)
31. Paturel, M., Subasinghe, I., Heiser, G.: First steps in verifying the seL4 Core Platform. In: Asia-Pacific Workshop on Systems (APSys). ACM, Seoul, KR (Aug 2023). https://doi.org/10.1145/3609510.3609821
32. Pnueli, A.: The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science 1977 pp. 46–57 (1977), https://api.semanticscholar.org/CorpusID:117103037
33. SBS Implementers Forum: System Management Bus (SMBus) Specification (Version 2.0). Tech. rep. (Aug 2000)
34. Schirmer, N.: A sequential imperative programming language syntax, semantics, hoare logics and verification environment. Arch. Formal Proofs **2008** (2008), https://www.isa-afp.org/entries/Simpl.shtml
35. Semiconductors, N.: I2c-bus specification and user manual. Tech. rep. (Oct 2021), https://www.nxp.com/docs/en/user-guide/UM10204.pdf
36. Steven Vaughan-Nichols: MINIX: Intel's hidden in-chip operating system, https://www.zdnet.com/article/minix-intels-hidden-in-chip-operating-system/

37. Sudvarg, M., Gill, C.: A concurrency framework for priority-aware intercomponent requests in camkes on sel4. In: 2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 1–10 (2022). https://doi.org/10.1109/RTCSA55878.2022.00007
38. System Management Interface Forum, Inc.: PMBus Power System Management Protocol Specification Part I – General Requirements, Transport And Electrical Interface (Revision 1.3.1). Tech. rep. (Mar 2015)
39. The seL4 community: Notifications and shared memory (Nov 2020), https://docs.sel4.systems/Tutorials/notifications.html, accessed on 2024-05-28
40. Xilinx: Zynq UltraScale+ MPSoC Software Developer Guide. Tech. Rep. UG1137 (v2022.2) (Nov 2022)