Capabilities, MMIO and Robust Safety

March 10, 2020

1 Memory Mapped I/O: Operational Semantics

The proposal is to simply represent read and writes to memory-mapped IO addresses as events in a trace. This says nothing a priori about devices that might be connected to these IO regions. In particular, without additional assumptions, reading a byte from a memory-mapped region just returns an arbitrary value.

If at some point we want to reason under the assumption that we are connected to a specific device that reacts in a specific way, we can express that as an extra Separation Logic assertion, that we assume as a pre-condition, and that restricts the set of different traces that we might observe.

$$\begin{array}{lcl} \text{EventTy} & := & \text{IOWrite} \mid \text{IORead} \\ \text{Event} & := & \text{EventTy} \times \text{Addr} \times \mathbb{Z} \\ \text{Trace} & := & \text{list Event} \\ \text{State} & := & \underbrace{\text{Reg} \times \text{Mem}}_{\text{old state}} \times \text{Trace} \end{array}$$

Values of type State represent the state of a configuration in the small-step operational semantics. In this setup, we assume the whole semantics to be parameterized by the range of memory-mapped addresses: MMIO.

$$\mathsf{MMIO} \;:=\; [\mathsf{MMIO}_{\mathrm{b}}, \mathsf{MMIO}_{\mathrm{e}})$$

In the (current) operational semantics without MMIO, the operational semantics of the `Load` instruction is:

$$\textsc{Load}\quad \frac{r[\text{src}] = (p, g, b, e, a) \qquad \text{readAllowed}\; p \qquad a \in [b, e)}{(r, m) \xrightarrow{\texttt{Load dst src}} (r[\text{dst} := m[a]], m)}$$

With MMIO, we obtain two rules for `Load`:

$$\frac{r[\text{src}] = (p,g,b,e,a) \qquad \text{readAllowed}\ p \qquad a \in [b,e) \qquad a \notin \mathsf{MMIO}}{(r,m,t) \xrightarrow{\texttt{Load dst src}} (r[\text{dst} := m[a]], m, t)}\ \textsc{MemLoad}$$

$$\frac{r[\text{src}] = (p,g,b,e,a) \qquad \text{readAllowed}\ p \qquad a \in [b,e) \qquad a \in \mathsf{MMIO}}{(r,m,t) \xrightarrow{\texttt{Load dst src}} (r[\text{dst} := x], m, t \mathbin{+\!\!+} (\text{IORead}, a, x))}\ \textsc{IOLoad}$$

Notice how in the second rule, we read an arbitrary integer x, and record it in the trace.

The `Store` rule would be similar.

Notice that, for any address in MMIO, which value is held by the map m as that address is now irrelevant. Indeed, m is never read nor modified for addresses in MMIO. One could choose to enforce that a particular dummy value is stored in m for these addresses. Instead, we choose to leave them unconstrained.

Remark: Thomas: Additionally, we might want to disallow in the semantics an execution with a PC register pointing to an MMIO location (I think this makes more sense than reading a random instruction through I/O). Armaël: but it makes the semantics of all instructions more complicated.

Remark: With this presentation, the specification of the "machine" is very much decoupled from the model of the devices it might be communicating with. This is a good thing, I think.

Nevertheless, one could consider an alternative presentation where the model of the devices is more tightly integrated with the semantics of the machine. For instance, one could make the operational semantics parameterized with the devices' model, where each device is modeled as having some internal state, the ability to react on reads or writes, or to perform an internal step. Then, the operational semantics would either step the usual way, or whenever a device steps.

I believe this would be somewhat similar to the semantics of I/O system calls through a foreign function interfaces as formalized in CakeML [FPK$^+$18], and also in Perennial [CTKZ].

Remark: Alix says that the proposed style of operational semantics (using a trace) is close to the semantics of `volatile` as in CompCert—which is also how memory mapped addresses seem to be exposed to C compilers in practice. So this is probably a good sign.

---

2 Separation Logic Resources

We added a trace as part of the state, so we wish to also expose it as a Separation Logic assertion. Recall the current definition of the state interpretation:

$$\text{state\_interp } (r, m) := \text{gen\_heap\_ctx } r * \text{gen\_heap\_ctx } m$$

The simplest way to account for the trace is to directly expose it in a monolithic fashion. In that case, the state interpretation additionally holds a resource for the trace.

$$\text{state\_interp } (r, m, t) := \text{gen\_heap\_ctx } r * \text{gen\_heap\_ctx } (m \setminus \mathsf{MMIO}) * \boxed{\bullet\, t}^{\gamma_T}$$

Note that we restrict the usual "points-to" assertions to be used only for non-MMIO addresses. Instead, to assert ownership of the MMIO region, the user works with assertions of the form $\boxed{\circ\, t'}^{\gamma_T}$. Such an assertion is not duplicable, and grants full ownership over the trace. In particular, it allows one to *update* the trace by emitting events, i.e. by performing I/O operations.

The wp-rules for `Load` and `Store` need to be updated consequently. For instance, the rule for a `Load` reading the integer x on a MMIO address a now requires $\boxed{\circ\, t}^{\gamma_T}$ in the pre-condition (for some t), and provides $\boxed{\circ\, (\text{IORead}, a, x) :: t}^{\gamma_T}$ in the post-condition.

The corresponding resource algebra is $\textsc{Auth}(\textsc{Ex}(\text{Trace}))$. A few relevant rules are:

$$\begin{array}{llll} \boxed{\bullet\, t}^{\gamma_T} * \boxed{\circ\, t'}^{\gamma_T} & \mathrel{-\!\!*} & \ulcorner t = t' \urcorner & \\ \boxed{\circ\, t}^{\gamma_T} * \boxed{\circ\, t'}^{\gamma_T} & \mathrel{-\!\!*} & \mathsf{False} & \\ \boxed{\bullet\, t}^{\gamma_T} * \boxed{\circ\, t}^{\gamma_T} & \mathrel{\Rrightarrow\!\!*} & \boxed{\bullet\, t'}^{\gamma_T} * \boxed{\circ\, t'}^{\gamma_T} & \end{array}$$

Remark: This is very coarse-grained: either one has the full ownership for performing I/O and reasoning about it, or one cannot know anything about it.

A first extension could be to allow observing prefixes of the trace (since events can only be appended to the trace). The observation that the trace has a given prefix would be duplicable. This again seems to be an application of monotonicity, that could be realized by having a duplicable AtLeast part as part of $\boxed{\circ\, t'}^{\gamma_T}$, similarly to how we currently model Monotone References.

Another extension, that seems very useful for modular reasoning, would be to allow splitting the trace along separate range of addresses. Concretely, a trace containing events about the range of MMIO addresses $[a, c)$ could be split into two traces, granting ownership over events on addresses $[a, b)$ and $[b, c)$ respectively. Note that recombining these two traces would only yield some unspecified interleaving of the events from the two traces, and not necessarily yield the original trace, since in our model, we cannot know the exact interleaving of IO operations issued by different parties.

One application of this second extension could be the verification of an example involving "multiplexed" I/O, where two separate parts of the code are granted separate ownership over separate MMIO addresses. These two separate pieces of code would be verified separately; then, in the end, one could prove that one gets *some* interleaving of all the events emitted by both components.

Remark: As an alternative to having the set of MMIO addresses (MMIO) as a global parameter, we might want to make it part of the state in the operational semantics. In that case, we would need another resource algebra to model the MMIO locations, i.e. add do the state interpretation we had before:

$$\text{state\_interp}\ (r, m, t, \mathsf{MMIO}) := \ldots * \boxed{\bullet\,(\text{MMIO})}^{\gamma_{\text{MMIO}}}$$

(where $\text{AUTH}(\text{EX}(\text{Trace}))$ is the resource algebra).

Remark: In the future, we might be interested in the dynamic allocation of MMIO memory. The question is what this would mean, though, since the set of available devices allowing for MMIO access and the concrete buffers they provide will most likely still be modeled as fixed at runtime. Rather, this would allow us to take (un)mapped MMIO locations, connected to the different devices, and (un)map them anywhere in main memory. This would require a different way of modeling MMIO, where we need both a notion of the total pool of possible MMIO locations, and a description of the currently mapped locations. We could model this using an authoritative RA.

---

3 Toplevel Theorem (*à la* OCPL)

Let us start with a brief recap of the toplevel "Robust Safety" theorem for OCPL itself (and its key ingredients), then the extension (by Thomas) of OCPL to include a `print` capability, and finally move to the capability machine setting.

3.1 OCPL

The ROBUSTSAFETY theorem of OCPL is as follows:

$$\frac{e \text{ closed} \qquad \{\mathsf{True}\}\, e\, \{x.\ \mathsf{lowval}\ x\} \qquad \begin{array}{c} C \in AdvCtx \\ (C[e]); (\emptyset, \mathsf{OK}) \longrightarrow^* T'; (h', g') \end{array}}{g' = \mathsf{OK}}$$

For any closed expression e, if e has been verified to only return low values, then running e wrapped in an adversarial context C from an initial state, then we can observe that every reachable state is good ($g' = \mathsf{OK}$ means that no assertion has failed in e).

$C \in \mathit{AdvCtx}$ means that C cannot contain assertions (otherwise one could trivially contradict the theorem by taking $C[\cdot] = \mathsf{assert\ false}$), and cannot contain references to raw memory locations (otherwise one could directly access e's private state, invalidating the local state encapsulation mechanisms).

$\mathsf{lowval}\ x$ intuitively means that x cannot be used to directly access private (or "high") locations. It is formally defined using a logical relation "$\mathsf{lift}\ \Psi\ v$", which more generally asserts that the value v only gives direct access to locations that satisfy the predicate Ψ.

$$\begin{array}{lcl}
\mathsf{lift}\ \Psi\ (\mathsf{rec}\ f\ x.\ e) & \triangleq & \triangleright \forall v.\ \{\mathsf{lift}\ \Psi\ v\}\ e[v/x, \mathsf{rec}\ f\ x.\ e/f]\ \{y.\ \mathsf{lift}\ \Psi\ y\} \\
\mathsf{lift}\ \Psi\ (v_1, v_2) & \triangleq & \triangleright(\mathsf{lift}\ \Psi\ v_1, \mathsf{lift}\ \Psi\ v_2) \\
\mathsf{lift}\ \Psi\ \ell & \triangleq & \Psi\ \ell \\
\ldots & &
\end{array}$$

Then, $\mathsf{lowval}\ x$ is defined as $\mathsf{lift\ lowloc}\ x$, where lowloc is a predicate characterizing the region of memory containing "low locations" (distinct from the other region containing "high locations").

3.2 OCPL with `print`

In Thomas' extension of OCPL, a new value Out is added, denoting an "output object capability", as well as a print primitive, where $\mathsf{print\ Out}\ v$ effectively "prints" the value v, i.e. adds it to the trace of printed values.

One then wants to be able to encapsulate the use of Out, for instance by defining object capabilities that enforce some invariants on the values being printed. In that setting, Out is considered as a "high" value:

$$\mathsf{lift}\ \Psi\ \mathsf{Out} \quad \triangleq \quad \mathsf{False}$$

The theorem then becomes:

$$\frac{C \in \mathit{AdvCtx} \qquad e \text{ closed} \qquad \boxed{\exists t.\ \left[\circ\, t\right]^{\gamma_{\mathrm{T}}} * \ulcorner P(t) \urcorner}^{\iota} \vdash \{\mathsf{True}\}\ e\ \{x.\ \mathsf{lowval}\ x\} \qquad (C[e]); (\emptyset, \mathsf{OK}); \emptyset \longrightarrow^{*} T'; (h', g'); t}{g' = \mathsf{OK} \wedge P(t)}$$

That is, if e has been verified under the assumption that the predicate P holds as a trace invariant, then executing e in an adversarial context yields a trace that does satisfy P.

$C \in \mathit{AdvCtx}$ also has to be extended to forbid C from containing Out.

To be slightly more general, we could actually allow sharing Out in case the predicate P does not place any constraints on the output, i.e. it is the True

predicate. Technically, we could hence have the following alternative definition for lift (although admittedly, it does look a bit hard-coded):

$$\mathsf{lift}\ \Psi\ \mathsf{Out} \quad \triangleq \quad \exists \iota.\ \boxed{\exists t.\ \boxed{\circ\, t}^{\gamma_{\mathrm{T}}} * \ulcorner (\lambda\, \_.\ \mathsf{True})\ t \urcorner}^{\iota}$$

3.3 Capability Machine with MMIO

3.3.1 Logical relation

Similarly to lift, we can generalize the existing logical relation to thread a constraint on directly accessible memory locations. One would parameterize the value, expression and register relations ($\mathcal{V}$, $\mathcal{E}$ and $\mathcal{R}$) with a predicate Ψ on addresses.

Then, for any permission p that includes either the R or W bit, $\mathcal{V}$ is extended as follows:

$$\mathcal{V}^{\Psi}(p, g, b, e, a) \triangleq \underbrace{\dots}_{\text{as before}} * \ulcorner \forall a' \in [b, e).\ \Psi(a') \urcorner$$

Then, $\mathcal{E}^{\Psi}$ and $\mathcal{R}^{\Psi}$ are simplify defined by threading the extra Ψ parameter through the existing definition.

Finally, for our relation to characterize "low values" that do not give direct access to MMIO addresses, one would instantiate Ψ with a predicate $\overline{\mathsf{MMIO}}$ that excludes memory mapped addresses:

$$\overline{\mathsf{MMIO}}(a) \triangleq a \notin \mathsf{MMIO}$$

Then, $\mathcal{V}^{\overline{\mathsf{MMIO}}}$ is similar to the lowval predicate of OCPL. Intuitively, a value in the relation does not directly point to memory-mapped locations, nor can it gain access to memory-mapped locations indirectly, similarly to how lowvals cannot grant access to high locations directly. The relation $\mathcal{E}^{\overline{\mathsf{MMIO}}}$ specifies that, if we fill the registers with values in $\mathcal{V}^{\overline{\mathsf{MMIO}}}$, then the invariants in a private future world of our starting world will be satisfied at the end of execution.

Remark: As it is defined above, the "generalized" value relation is in fact not very useful for instantiation of Ψ other than $\overline{\mathsf{MMIO}}$. Indeed, even if Ψ *does* allow referring to addresses in the MMIO region, the value relation does not grant any corresponding resources. The fix would be to use the generalized trace resources mentioned previously, and grant ownership for the part of the trace corresponding to the addresses in $[b, e) \cap \mathsf{MMIO}_{\mathrm{pub}}$ with $\mathsf{MMIO}_{\mathrm{pub}}$ the shareable subset of addresses in MMIO.

3.3.2 Robust safety theorem

A first attempt at adapting the OCPL robust safety theorem to the capability machine setting might look something like the following **(incomplete!)** statement.

$$\frac{\begin{array}{c}\forall a_i \in A_{\text{entry}}.\ \boxed{\exists t.\ \boxed{\circ\, t}^{\gamma_{\mathrm{T}}} * P(t)}^{\iota} \vdash \forall W.\ (\mathcal{E}^{\overline{\mathsf{MMIO}}}(\mathrm{RX}, g, b, e, a_i))\, W \\ (r[\overline{r_{a_i} := (\mathrm{E}, g, b, e, a_i)}], m, \emptyset) \longrightarrow^{*} (r', m', t)\end{array}}{P(t)}$$

The code that is verified (similar to the expression e in OCPL) is here given as a set of *entrypoints* (addresses) A_{entry}. The capability $(\mathrm{RX}, g, b, e, a_i)$ then points to the ith entry point of the code we want to gradually verify.

The capability $(\mathrm{E}, g, b, e, a_i)$ then represents a closure for this same piece of code. Adversarial code only gets access to the closure (as an enter capability), in order to avoid them from executing or reading arbitrary lines of code within the trusted module.

Notice the universal quantification over worlds in the precondition; this needs to be there, since we do not know in what world our code will be called.

Remark: We could simplify the theorem statement by only considering the case of a single entrypoint. This is equivalent: a piece of code that wants to expose two distinct "methods" can implement a single toplevel entrypoint that immediately returns two pointers that can then be used to invoke the two methods. This would make the toplevel theorem simpler, at the cost of somewhat more contrieved calling convention between trusted and untrusted code.

Remark: The universal quantification on worlds was not explicitly present in the OCPL formalization. The reason for that is that (I think so at least, do not quote me on this - Thomas) they used the built-in Iris worlds within their definition of weakest precondition, by leveraging the state interpretation, which is universally quantified over in the definition of weakest precondition. As we discussed before, if we do away with local capabilities, it should be possible to fall back onto Iris' notion of worlds, which would allow us to remove the universal quantification in the above definition too because it would be implicitly present in the WP inside $\mathcal{E}$.

It would be interesting to have a discussion about how to do away with the worlds if we remove local capabilities, regardless of whether we will in the end, since that might shed some light on why the worlds look the way they do right now.

Remark: Note that we could have written $\ldots \vdash \forall W.\ (\mathcal{V}^{\overline{\mathsf{MMIO}}}(\mathrm{E}, \ldots))\ W$ in the above theorem, instead of using the expression relation. This is equivalent, and easily unfolds to the former statement through the definition of the value relation. I wrote $\mathcal{E}$ to place the focus on the execution of the driver being safe. I consider this a matter of taste.

The theorem above is *incomplete*. In other words, it is not provable as is, because it includes no notion of adversarial context (similar to OCPL's *AdvCtx*), and therefore imposes no restriction on the adversarial code (i.e., the contents of the r and m other than our verified code). As such, the statement can be trivially falsified by taking an attacker with direct read or write access to any location in MMIO, or with direct access to the trusted code's internal state.

We move to a theorem statement that correctly constrains the adversary in two steps:

1. We add a syntactic condition (on the state of memory and registers) to the tentative statement of our robust safety theorem, ensuring that the adversary code cannot gain access to any non-E capabilities pointing into the driver code or any capabilities pointing into MMIO directly.

2. Then, we provide an end-to-end statement that assumes the existence of a piece of boot code, that gets to run first when the capability machine starts up, and makes sure the above condition indeed holds before passing control to the adversary.

   The boot code is part of the trusted code base, and must satisfy a given specification, as a precondition of the toplevel theorem.

   The particular implementation of the boot code depends on the model of the initial state of the machine. If the memory can contain any capability at startup, then the boot code has to erase the memory before passing control. If we assume that the initial memory cannot contain any capabilities whatsoever, then the boot code does not need to do any cleanup.

In this setting, we can subdivide the memory m into 3 relevant subsections; $m = m_{\mathsf{MMIO}} \uplus m_{\text{driver}} \uplus m_{\text{adv}}$ with the following meaning:

- m_{MMIO} contains the memory-mapped locations
- m_{driver} contains the driver code
- m_{adv} contains the adversary's code and data (this will also include the boot code in the general setting, since it does not contain any inherently dangerous capabilities itself).

We can then implement the syntactic check as requiring registers and memory to only contain non-capability values:

$$\text{nonCap}(\text{inl } z) \triangleq \mathsf{True} \qquad\qquad \text{nonCap}(\text{inr } (p, g, b, e, a)) \triangleq \mathsf{False}$$

$$\text{nonCap}(\mathit{reg}) \triangleq \forall r \in \text{dom}(\mathit{reg}).\ \text{nonCap}(\mathit{reg}(r))$$

$$\text{nonCap}(\mathit{mem}) \triangleq \forall l \in \text{dom}(\mathit{mem}).\ \text{nonCap}(\mathit{mem}(l))$$

Given this definition, we can now formalize the initialization constraints on registers r and memory m, given the region MMIO and the driver's address space $[b, e)$ as follows:

$$\frac{\begin{array}{c} m = m_{\mathsf{MMIO}} \uplus m_{\text{driver}} \uplus m_{\text{adv}} \\ \text{dom}(m_{\mathsf{MMIO}}) = \mathsf{MMIO} \qquad \text{dom}(m_{\text{driver}}) = [b, e) \qquad \text{dom}(m_{\text{adv}}) = [b_{\text{adv}}, e_{\text{adv}}) \\ \text{nonCap}(r) \qquad \text{nonCap}(m_{\text{adv}}) \qquad r[\text{PC}] = (\text{RWX}, \text{G}, b_{\text{adv}}, e_{\text{adv}}, b_{\text{adv}}) \end{array}}{\text{init}_{\text{OK}}([b, e), r, m)}$$

and subsequently formulate the robust safety theorem as follows:

$$\frac{\begin{array}{c} \forall a_i \in A_{\text{entry}}.\ \boxed{\exists t.\ \boxed{\circ\, t}^{\gamma_{\text{T}}} * P(t)}^{\iota} \vdash \forall W.\ (\mathcal{E}^{\overline{\mathsf{MMIO}}}(\text{RX}, g, b, e, a_i))\, W \\ \text{init}_{\text{OK}}([b, e), r, m) \qquad (r[\overline{r_{a_i} := (\text{E}, g, b, e, a_i)}], m, \emptyset) \longrightarrow^* (r', m', t) \end{array}}{P(t)}$$

where we have set the PC up to be a sensible capability (pointing to b_{adv} without loss of generality).

Notice how the syntactic check for the registers is only performed on the set of registers r, allowing the closures in $\overline{r_{a_i}}$ to point into to the driver address space.

Remark: If useful, the syntactic check can be relaxed to allow any capabilities that do not point into the MMIO region or the driver's memory. One would define the predicate $\rightarrow^{\overline{R}}(w)$, that takes a word, memory region or register file and checks if it points into the forbidden region R as follows:

$$\begin{array}{lcl} \rightarrow^{\overline{R}}(\text{inl } z) & \triangleq & \mathsf{True} \\ \rightarrow^{\overline{R}}(\text{inr } (p, g, b, e, a)) & \triangleq & R \cap [b, e) \ = \emptyset \\ \rightarrow^{\overline{R}}(\mathit{reg}) & \triangleq & \forall r \in \text{dom}(\mathit{reg}).\ \rightarrow^{\overline{R}}(\mathit{reg}(r)) \\ \rightarrow^{\overline{R}}(\mathit{mem}) & \triangleq & \forall l \in \text{dom}(\mathit{mem}).\ \rightarrow^{\overline{R}}(\mathit{mem}(l)) \end{array}$$

Then, one would use $\rightarrow^{\overline{\mathsf{MMIO} \cup [b,e)}}()$ in place of nonCap().

Having this theorem in our toolbox, we can now take a closer look at scenario 2 above, where a piece of boot code is used to set the memory up correctly, obviating the need for any other assumptions than that the boot code behaves according to some spec. Concretely, we want the boot code to uphold a spec that allows us to prove the above theorem. (Alternatively, we could also just implement one specific instance of boot code, and prove that after it finishes executing, all preconditions to apply the above theorem are met.)

Let r_0 and m_0 be the respective initial state of registers and memory immediately after booting. We assume that the following two properties hold, where l_{boot} is some memory location (initially loaded in PC):

$$\mathrm{dom}(m_0) = [0, \mathrm{MEM}_{\mathrm{MAX}}) \qquad r_0[\mathrm{PC}] = (\mathrm{RWX}, \mathrm{G}, 0, \mathrm{MEM}_{\mathrm{MAX}}, l_{\mathrm{boot}})$$

Depending on the specifics of the operiational semantics, we might have more information about m_0 and r_0, that can then be used in the implementation of the boot code (e.g., m_0 might be initialized to only zeroes).

We also assume the boot-code to be initially loaded in memory at address l_{boot}. Then, the specification for the boot code that we have to prove is stated as follows:

$$\mathrm{bootSpec}(P, g, b, e, A_{\mathrm{entry}}) \triangleq$$

$$\left\{ \begin{array}{l} \mathrm{PC} \mapsto_r (\mathrm{RWX}, \mathrm{G}, 0, \mathrm{MEM}_{\mathrm{MAX}}, l_{\mathrm{boot}}) \ast \\ \mathop{\Huge\ast}\limits_{r \in \mathrm{Reg} \setminus \mathrm{PC}} r \mapsto_r \_ \ast \mathop{\Huge\ast}\limits_{m \in \mathrm{Mem} \setminus \mathsf{MMIO}} m \mapsto_a \_ \ast \boxed{\circ\, \emptyset}^{\gamma_{\mathrm{T}}} \end{array} \right\}$$

$$\texttt{Instr Executable}$$

$$\left\{ \begin{array}{l} \exists b_{adv}\ e_{adv}\ t. \\ \quad \mathrm{PC} \mapsto_r (\mathrm{RWX}, \mathrm{G}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \quad \mathop{\Huge\ast}\limits_{a_i \in A_{\mathrm{entry}}} a_i \mapsto_r (\mathrm{E}, g, b, e, a_i) \ast \mathop{\Huge\ast}\limits_{r \in Reg \setminus \{\mathrm{PC}, A_{\mathrm{entry}}\}} r \mapsto_r \mathrm{inl}\ \_ \ast \\ \quad \mathop{\Huge\ast}\limits_{m \in [b_{adv}, e_{adv})} m \mapsto_a \mathrm{inl}\ \_ \ast \boxed{\circ\, t}^{\gamma_{\mathrm{T}}} \ast \ulcorner P(t) \urcorner \end{array} \right\}$$

Intuitively, establishing this specification ensures that, after running the boot code, we arrive in a state that satisfies $\mathrm{init}_{\mathrm{OK}}$.

The resulting boot-code-base formulation of the robust safety theorem is now a corollary of the general robust safety theorem, and has the following statement:

$$\frac{\forall a_i \in A_{\text{entry}}.\ \boxed{\exists t.\ \boxed{\circ\, t}^{\gamma_{\mathrm{T}}} * P(t)}^{\iota} \vdash \forall W.\ (\mathcal{E}^{\overline{\mathsf{MMIO}}}(\mathrm{RX}, g, b, e, a_i))\, W \qquad \text{bootSpec}(P, g, b, e, A_{\text{entry}}) \qquad (r_0, m_0, \emptyset) \longrightarrow^{*} (r, m, t)}{P(t)}$$

Note that to link scenario 2 to scenario 1, we need a slightly adapted version of the robust safety theorem stated above where the initial trace is allowed to not be empty, but rather just satisfy $P(t)$, but that should not cause any problems.

Note as well that here, if the boot-code performs I/O, it is required to satisfy the invariant P on the trace. If needed, one could come up with a more general theorem statement, where the boot code is allowed to perform arbitrary I/O, and where P is thus only enforced on the suffix of the trace *after* the boot code has run...

Remark: On the level of the operational semantics, proving the boot specification corresponds to proving that the following holds, in order to use our concrete driver implementation with the robust safety theorem above (the driver still needs to be proven safe separately):

$$\begin{aligned}
&(\textsc{BootCodeOK})\\
&\exists n\ b\ e.\,(r_0, m_0, \emptyset) \longrightarrow^{n} (r'[\overline{r_{a_i} := (\mathrm{E}, g, b, e, a_i)}], m', t)\\
&\wedge P(t) \wedge \text{init}_{\text{OK}}([b, e), r', m')
\end{aligned}$$

3.3.3 Concrete boot and driver code

Now we have a closer look at the concrete example we would like to verify, including what its boot code looks like. Our goal is to prove that this concrete example satisfies bootSpec (alternatively, BootCodeOk) and this last robust safety theorem.

The code for for the driver scenario we want to gradually verify can be found in `driver_code.v`. This example file contains more details on the scenario in the comments. From a more high-level perspective, it contains a trusted part, consisting of a single MMIO location, boot code to properly set up the capability machine at start-up, and the code for the driver itself. On the other hand, the example also contains an untrusted part, consisting of a known, but untrusted code section and sandbox section (which we do not make assumptions over). The first address of the adversary code section is what the trusted part jumps to once it has finished setting up.

The boot code is where execution starts off when the capability machine is powered on. It contains an omnipotent RWX capability (i.e. ranging over all of memory and providing full permissions over it) starts off execution. (**Design Alternative:** in the long run, it might be worthwhile to provide the boot code with a RWLX capability. Since RWX cannot be upgraded to RWLX, it is impossible in the current setting to develop a secure stack calling convention á la Lau, where the stack capability has to be local-WL and (in this case) the driver code has to ensure that there is no other WL than the stack. However, having a global stack capability is currently not an issue, since the interface to our driver is first order, i.e. it will never create another adversary stack frame on top of itself (currently, it does not even use the stack in the first place!), and since we are working in a single-threaded setting (not very relevant, but I think the multi-threaded setting is an interesting thought experiment). I avoided unnecessary complications and hence kept the omnipotent capability as RWX. If the omnipotent capability were to become RWLX in the future, we would have to be careful using a stack, since if we want our secure calling convention with a higher order interface to our driver API, there cannot be any adversary-accessible global write-local memory, i.e. the reduced omnipotent capability cannot be allowed to be global if we pass it to the adversary. We would thus have to pass a stack capability that we explicitly make local, and a global RWX capability for the rest of memory)

The boot code has the following responsibilities:

- Generate the necessary closures (i.e. Enter capabilities) for the driver's read and write methods, so that the adversary can use safely use them to perform I/O, and any properties that the driver wants to uphold on the input-output stream can actually be enforced.

- Wipe all of the adversary's sandbox section, to make sure no remaining capabilities can be found there. (**Design Alternative:** maybe we should try to find out how CHERI handles this; is it possible that any lingering capabilities are left in RAM on machine start-up? In any case, we are already being slightly unrealistic by assuming the adversary code section is just *there*. We have the option to not do any erasure, and just make the assumption that adversary memory contains no capabilities pointing into MMIO or pointing into the driver.). It does not, however, wipe the untrusted, hard-coded adversary routine that reads N lines of code from a single (and the only) MMIO location in memory.

- Reduce the omnipotent capability to provide RWX access to all of the adversary's code, makes it point to the first element of the adversary's code section and jumps to it, in order to hand control to the adversary. Before the jump, it clears all registers except for the 2 containing the read and write driver closures, in order to make sure that no extra permissions leak to the adversary.

References

[CTKZ] Tej Chajed, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Perennial: language semantics. `https://github.com/mit-pdos/perennial/blob/d71e8fa9291ba51eb4b59915f5ac94b728578899/src/goose_lang/lang.v`.

[FPK+18] Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program verification in the presence of I/O: Semantics, verified library routines, and verified applications. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments (VSTTE)*, volume 11294 of *Lecture Notes in Computer Science*. Springer, 2018.