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.

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

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

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

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

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

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

Notice how in the second rule, we read an arbitrary integer x, and record it in the trace [**Rk A.1.1**]. The `Store` rule would be similar.

We now have two options when it comes to modeling the memory m:

1. Include the MMIO addresses into the map. Notice that, for any address in MMIO, the specific value held by the map m for that address is now irrelevant. Indeed, m is never read nor modified for addresses in MMIO (possible exception: see [**Rk A.1.2**]). One could choose to enforce that a particular dummy value is stored in m for these addresses or to leave them unconstrained.

2. Have m model the physical RAM addresses only. This model is closer to the conceptual model of state that we wish to implement, since it does not require us to talk about the values in the heap that MMIO locations are associated with. It however probably requires threading through the operational semantics the invariant that m is disjoint from MMIO.

Both approaches require us to make sure that no points-to chunks $l \mapsto \_$ with $l \in \mathsf{MMIO}$ can be created. In the next section, we assume the second approach [**Rk A.1.2**].

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 [**Rk A.2.1**]:

$$\text{state\_interp } (r, m, t) := \text{gen\_heap\_ctx } r * \text{gen\_heap\_ctx } m * \text{dom}(m) \cap \mathsf{MMIO} = \emptyset * \boxed{\bullet\, t}^{\gamma_{\mathrm{T}}}$$

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 $\boxed{\circ t +\!\!+ (\mathrm{IORead}, a, x)}^{\gamma_T}$ in the post-condition.

The corresponding resource algebra is Auth(Ex(Trace)) [**Rk A.2.2**]. A few relevant rules are:

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

To simplify notation, let us introduce the following definitions for user-level ownership over the trace, and preservation of an invariant on the trace, respectively:

$$\begin{array}{rcl}\mathrm{tr}(t) & \triangleq & \boxed{\circ t}^{\gamma_T} \\ \mathrm{trInv}^{\iota}(P) & \triangleq & \boxed{\exists t.\ \mathrm{tr}(t) * \ulcorner P(t) \urcorner}^{\iota}\end{array}$$

[Rk A.2.3] **[Rk A.2.4]**

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:

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

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 $C[\cdot] = \mathsf{assert}\ \mathsf{false}$), 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 $\mathsf{lift}\, \mathsf{lowloc}\ x$, 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 $\mathsf{print}\ \mathsf{Out}\ v$ 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:

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

The theorem then becomes:

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

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):

$$\mathsf{lift}\ \Psi\ \mathsf{Out} \quad \triangleq \quad \mathrm{trInv}^{\iota}(\lambda\ \_.\ \mathsf{True})$$

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:

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

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. [**Rk A.3.1**]

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.

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

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 $(\text{RX}, g, b, e, a_i)$ then points to the ith entry point of the code we want to gradually verify. **[Rk A.3.2]**

The capability (E, g, b, e, a_i) then represents a closure for this same piece of code. Adversarial code should only get access to the closure (as an enter capability), in order to prevent it from executing or reading arbitrary lines of code within the trusted module.

Notice that we also need to provide a world to instantiate the expression relation, here, we universally quantify over it... **[Rk A.3.3]**

The theorem above is *incomplete*. In particular, 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 now try to develop a theorem statement that correctly constrains the adversary.

A first thing that is lacking is a way to constrain the memory and the registers of the adversary once it gains control. We have to add a condition (on the state of the reachable 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. A possible way of syntactically expressing this condition (and in the process overestimating it), is to define the predicate nonCap() below. More specifically, we can implement the syntactic check as requiring registers and memory (except perhaps driver memory) to only contain non-capability values:

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

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

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

[Rk A.3.4]

This condition can also easily be specified in terms of the value relation $\mathcal{V}^{\overline{\text{MMIO}}}$, by requiring the registers and part of the PC register to be valid, as will be demonstrated below. This more semantic specification is easily implied by the syntactic one above, and is additionally the one we will need to apply the ftlr to the adversary code, hence proving that it executes securely.

Another change we add is that rather than having execution start off with the adversary, we model the entire boot process of a capability machine, and allow so-called *boot code* to run before the adversary does. Because of this set-up, rather than having a theorem that assumes unknown adversary code to

start executing immediately, we provide an end-to-end theorem that assumes the existence of this piece of boot code, that gets to run first when the capability machine starts up, and makes sure the necessary conditions for the ftlr indeed hold 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. We call this bootcode specification BootSpec and will demonstrate it below. 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 from address b to e
- 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). For simplicity, we will assume a contiguous region for the adversary, from b_{adv} to e_{adv}.

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

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

However, this predicate is not defined in terms of Iris resources, i.e. we cannot use it in our Coq development yet. We will redefine it below, in the process making it more semantic and hence general. The above predicate, however, demonstrates the state we would like our capability machine to be in when the adverary starts executing. Additionally, we will need to state that any registers pointing to the driver are secure; this is still lacking from the formalization.

The general semantic precondition on adversary execution would then contain the following elements:

- All registers except PC are in the value relation with respect to a world W, meant to contain a single standard region for all of the adversary's memory. The PC register satisfies both the read- and write conditions

(note the parallel with the fundamental theorem) with respect to this same world. If we ever want to allow more than just enter capabilities to point to the driver's memory, we have to make sure that some region governing the driver is part of the world W. It is not entirely clear yet how to do this in our current development.

- We have possession of the predicates `region` and `sts_full_world` for the world we picked in the previous bullet. we will need these to apply the fundamental theorem.

We can now take a closer look at the scenario 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.

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):

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

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 capability machine to initially start execution in the boot code at address l_{boot}.

Then, the semantic specification for the boot code that we have to prove is stated as follows:

$$\text{bootSpec}(P, r_0, m_0, \mathit{bootCont}, \Phi) \triangleq$$

$$\left\{ \begin{array}{l} \text{PC} \mapsto_r (\text{RWX}, \text{G}, 0, \text{MEM}_{\text{MAX}}, l_{\text{boot}}) * \\ \quad \mathop{\Large *}_{r \in \text{Reg} \setminus \text{PC}} r \mapsto_r r_0[r] \; * \mathop{\Large *}_{m \in \text{Mem} \setminus \mathsf{MMIO}} m \mapsto_a m_0[m] \; * \; \text{trInv}^{\iota}(P) * \\ \triangleright \, \mathit{bootCont} \end{array} \right\}$$

$$\texttt{Seq}\,(\texttt{Instr Executable})$$

$$\{\Phi\}$$

$$\text{bootCont}_{\text{sem}}(P, \Phi, \text{W}_{\text{init}}) \triangleq \forall b_{\text{adv}}\; e_{\text{adv}}\; a_{\text{adv}}\; \overline{w_r}\; (W' \sqsupseteq^{\text{priv}} \text{W}_{\text{init}}).$$

$$(\text{PC} \mapsto_r (\text{RWX}, \text{G}, b_{\text{adv}}, e_{\text{adv}}, a_{\text{adv}}) * \mathcal{V}^{\overline{\mathsf{MMIO}}}(W')(\text{RW}, \text{G}, b_{\text{adv}}, e_{\text{adv}}, a_{\text{adv}}) *$$

$$\mathop{\Large *}_{r \in Reg \setminus \{\text{PC}\}} (r \mapsto_r w_r * \mathcal{V}^{\overline{\mathsf{MMIO}}}(W')(w_r)) \; *$$

$$\text{region}(W') * \text{sts\_full\_world sts\_std } W')$$

$$\mathrel{-\!\!*} \; \texttt{wp Seq}\,(\texttt{Instr Executable})\,\{\Phi\}$$

$$\text{bootSpec}_{\text{sem}}(P, r_0, m_0) \triangleq$$

$$\forall \Phi.\, \text{bootSpec}(P, r_0, m_0, \text{bootCont}_{\text{sem}}(P, \Phi, \emptyset), \Phi)$$

To prove that any closures the boot code might create in the above actually constitute valid register values, we would need to prove the following, with A_{entry} the set of entry points to the driver:

$$\forall a_i \in A_{\text{entry}}.\; \forall W \sqsupseteq^{\text{priv}} \text{W}_{\text{init}}.\; \mathcal{E}^{\overline{\mathsf{MMIO}}}(W)(\text{RX}, g, b, e, a_i)$$

To actually be able to construct this proof, we will need the following 2 elements (they both need to be persistent, so that we can use them within the execution condition):

$$\text{trInv}^{\iota}(P) * \text{code}(b, e, \overline{w_l})$$

Here we defined code to be a description of the exact driver code, still in terms of points-to chunks. We can write this invariant as $\text{code}(b, e, \overline{w_l}) = \boxed{\mathop{\Large *}_{l \in [b,e)} l \mapsto_a w_l}^{\iota}$, with w_l some hard-coded driver word (an instruction or a capability) at location l.

To prove the PC register valid, we will often be able to use the fact that the memory left by the boot code only contains integers z, i.e. that the nonCap predicate holds for memory.

Note that the above specification does not allow for a scenario where we have multiple set-up programs in a row, i.e. multiple boot or set-up processes that each need some form of access to MMIO, that each want to set up their own

closures and that we want to verify modularly. The trust model of these set-up procedures goes bottom-up, i.e. any set-up procedure only trusts the set-up procedures that ran before it to ensure that its preconditions are satisfied when it assumes control. To allow for such a scenario, the execution of the bootcode has to be split up into different phases, where it holds that the postondition (precondition of the continuation) of boot program i implies the precondition of boot program $i+1$. Additionally, the last bootloader has to make sure that the registers it delivers are safe with respect to $\mathcal{V}^{\overline{\mathsf{MMIO}}}$, such that the ftlr (which will only hold for adversaries that have no access to MMIO whatsoever) can be applied.

In general, each set-up process i will require the following as a precondition to safely execute.

- Safe register values, with respect to some argument World W (whose regions are standard - this will mostly be used for proving the enter capabilities to e.g the driver secure, and I expect this world to be empty at the moment) and a set of MMIO locations MMIO_i, for which it holds that $\mathsf{MMIO}_i \subseteq \mathsf{MMIO}$. This allows the process to perform some MMIO, but only to locations in $\mathsf{MMIO} \setminus \mathsf{MMIO}_i$. If needed, more strict functional requirements on registers can be made.
- Its code and data stored somewhere accessible in memory (using regular points-to chunks). Generally, also point-to chunks for memory that will be used in the future. This is the region, R_{adv}, currently represented as a contiguous range $[b_{\text{adv}}, e_{\text{adv}}[$ for simplicity. Note that it is possible to write drivers that use relative addressing in their spec to not make their precondition unnecessarily strict.
- The PC register points to (a subset of) R_{adv}, so that it is actually possible for our process to access this non-standard memory. Optionally, part of the memory the PC points to can be in W, but then the precondition needs to contain a proof that this part of the PC is RW-valid, i.e. $\in \mathcal{V}(\text{RW}, \ldots)$, so that we can actually obtain the necessary points-to chunks from the region predicate.
- An invariant $\text{trInv}^{\iota}(P)$ on the trace, so that the process can perform IO itself or create more closures to do so.

Each set-up process will deliver the following postcondition (precondition of the continuation) to the next set-up process or to adversary code, starting from the precondition: (this also means that the proof of the next process has to go through, given this postcondition, i.e. it has to be sufficiently specific)

- Safe register values in any ($\forall$) private future world of W, with respect to the relation $\mathcal{V}^{\mathsf{MMIO}_{i+1}}$, where it has to hold that $\mathsf{MMIO}_i \subseteq \mathsf{MMIO}_{i+1}$ i.e. the permissions to perform IO are monotonely decreasing as set-up progresses. This constitutes no problem for instances of $\mathcal{V}$,$\mathcal{E}$ and $\mathcal{R}$

proven in earlier set-up programs, since they all respect the following contravariant monotonicity property: $S_1 \subseteq S_2 \Rightarrow \mathfrak{X}^{\overline{S_2}} \subseteq \mathfrak{X}^{\overline{S_1}}$.

- Some description of the layout of memory and concrete register values, that should be sufficiently detailed to be able to prove the precondition of the next set-up program.
- The trace invariant $\mathrm{trInv}^{\iota}(P)$ (so that the next process can perform its own MMIO). In the future, we can allow strengthening of P throughout set-up by creating a monotone resource algebra for predicates, similarly to what we did for OCPL.

In the case of the last set-up process n, we will also need the following, to be able to prove the semantic bootspec's postondition and in the end apply the ftlr, thereby proving the rest of execution safe:

- We need that $\mathsf{MMIO}_n = \mathsf{MMIO}$, such that we can apply the ftlr (otherwise some valid registers might grant the adverary access to MMIO memory, and we cannot consider it an adversarial context in the robust safety sense)
- For similar reasons, the PC should now point to valid memory only, i.e. the condition $\mathcal{V}^{\overline{\mathsf{MMIO}}}(W)(\mathrm{RW}, \mathrm{G}, b_{\mathrm{adv}}, e_{\mathrm{adv}}, a_{\mathrm{adv}})$ should hold.
- The trace invariant $\mathrm{trInv}^{\iota}(P)$ is no longer required, since the untrusted adversary code will not be able to perform MMIO anyway. Leaving it there is not a technical problem either, however, given that the registers and memory will be valid with respect to $\mathcal{V}^{\mathsf{MMIO}}$.

Using the notation we introduced, we must prove the following for each

set-up program (TODO: write an attempt at a more modular spec here):

$$\begin{aligned}
&\mathrm{bootSpec}(P, g, b, e, \mathit{bootCont}, \Phi) \triangleq \\
&\quad \left\{ \begin{array}{l} \mathrm{PC} \mapsto_r (\mathrm{RWX}, \mathrm{G}, 0, \mathrm{MEM_{MAX}}, l_{\mathrm{boot}}) * \\ \quad \mathop{\ast}_{r \in \mathrm{Reg} \setminus \mathrm{PC}} r \mapsto_r r_0[r] \ * \mathop{\ast}_{m \in \mathrm{Mem} \setminus \mathsf{MMIO}} m \mapsto_a m_0[m] \ * \ \mathrm{trInv}^{\iota}(P) * \\ \quad \triangleright \mathit{bootCont} \end{array} \right\} \\
&\qquad \texttt{Seq}\ (\texttt{Instr}\ \texttt{Executable}) \\
&\quad \{\Phi\} \\
&\mathrm{bootCont_{synt}}(P, g, b, e, A_{\mathrm{entry}}, \Phi) \triangleq \\
&\ (\exists b_{adv}\ e_{adv}\ t. \\
&\qquad \mathrm{PC} \mapsto_r (\mathrm{RWX}, \mathrm{G}, b_{adv}, e_{adv}, b_{adv}) * \\
&\qquad \mathop{\ast}_{a_i \in A_{\mathrm{entry}}} r_{a_i} \mapsto_r (\mathrm{E}, g, b, e, a_i) \ * \mathop{\ast}_{r \in Reg \setminus \{\mathrm{PC}, A_{\mathrm{entry}}\}} r \mapsto_r \mathrm{inl}\ _ \ * \\
&\qquad \mathop{\ast}_{m \in [b_{adv}, e_{adv})} m \mapsto_a \mathrm{inl}\ _ \ * \mathop{\ast}_{l \in [b,e)} m \mapsto_a w_l \ * \ \boxed{\circ\, t}^{\gamma_{\mathrm{T}}} * \ulcorner P(t) \urcorner \\
&\qquad \forall a_i \in A_{\mathrm{entry}}. \left\{ \begin{array}{l} \mathrm{trInv}^{\iota}(P) * \mathrm{code}(b, e, \overline{w_l}) \vdash \\ \forall W.\ \mathcal{E}^{\overline{\mathsf{MMIO}}}(W)(\mathrm{RX}, g, b, e, a_i) \end{array} \right.) \\
&\quad \twoheadrightarrow \ \texttt{wp Seq}\ (\texttt{Instr}\ \texttt{Executable})\ \{\Phi\}
\end{aligned}$$

Remark: We will have to allocate the above predicates $\mathrm{region}(\mathrm{W_{init}}) * \mathrm{sts\_full\_world}\ \mathrm{sts\_std}\ \mathrm{W_{init}}$ from our Coq development under a different ghost name γ than we usually do. I do not know how exactly that would work on a technical level, but it should somehow. We should probably have $\mathrm{region}(\emptyset) * \mathrm{sts\_full\_world}\ \mathrm{sts\_std}\ \emptyset$ in the precondition of the hoare triple, and then externally have a universal quantification over the concrete `traceG` typeclass used, similar to how OCPL does this.

Remark: As Frank mentioned, if we do not enforce any validity constraints on the part of the adversary's memory that is not reachable from the valid registers themselves, the driver cannot read from this memory in a useful way during execution (since the memory might contain any value whatsoever, ánd we cannot obtain a points-to chunk to it). There is currently no concrete use case for this scenario yet, but it is good to be aware of. In the general case, we would have to make sure that the parts of the adversary code that are not pointed to by registers but are needed by the driver for some reason are present in the world (otherwise, we have no way to obtain the points-to chunk either), or present in an invariant.

Intuitively, establishing this specification ensures that, after running the boot code, we arrive in a state that satisfies init_{OK}.

The resulting boot-code formulation of the robust safety theorem is now

$$\frac{\text{bootSpec}^{\text{sem}}(P, r_0, m_0) \qquad (r_0, m_0, \emptyset) \longrightarrow^* (r, m, t)}{P(t)}$$

Note that we will need a slightly more general adequacy theorem than is suggested by the above robust safety theorem, 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 if the boot-code performs I/O, then it is required to preserve 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 true of the suffix of the trace for events that occur *after* the boot code has run.

Lastly, note how the trace invariant that appears in the boot specification is not an atomic invariant, even though atomic invariants might be involved in the verification of the driver if it has state (see e.g. the counter example below). We are obliged to have a regular invariant for the predicate over the trace, since otherwise adequacy breaks; even for infinite executions, we have to be able to show that any output values satisfy P at *any* point during execution.

Remark: Dominique made the remark that we might want to allow capabilities pointing into the driver code that are not enter capabilities, but can just generally be proven safe wrt. the driver code. This scenario (which we currently do not support out of simplicity considerations) would force us to add the driver code as a static region to the world. While this would work on paper, it does not in the current Coq formalization, since all regions that standard regions point to are forced to be standard as well (through the conditions in the different cases of the `interp1` predicate). Essentially, this behavior is disallowed because the Coq development merged the notion of read- and write condition, and there is hence no room to define a static region which defines a stricter reading policy, but hence disallowes writing. More generally, it is currently impossible to plug non-standard regions into the world, and allow standard regions to depend on them in any way whatsoever. This was not a problem up until now, since our examples never required this sort of dependency.

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 and hence the above 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.

4 Potential driver clients/properties

The concrete driver that we described in Section 3.3.3 does not yet enforce any useful safety properties on the traces that it outputs. This section describes a couple such properties that we might want to enforce, modifications to the driver that they require and example clients that make use of our modified driver.

4.1 Stateless properties

Printing values ≤ 1000 To only print values ≤ 1000, we need to wrap the driver's read method in a function that checks this condition, before passing control to the actual read method. On an abstract level, given a function f, we would have the following wrapper:

$$\text{wrap}_{\leq 1000}\, f \triangleq \lambda x.\, \text{assert}(x \leq 1000); f\ x$$

, and the closure we share with the environment is hence $\text{wrap}_{\leq 1000}(\text{read})$ instead of just read.

A trivial misbehaving client could look as follows: (TODO: ask Dominique for Lau's code on environments)

```
move r1 1001
jmp r2
```

It simply loads an argument to write and jumps to the write routine.

A well-behaved client might look as follows:

```
move r1 0
jmp r2
```

Modelling channel-specific safety properties Here, we could encode different channels into our integers (in the spirit of what Frank mentioned), and enforce different safety properties on the different channels (by ensuring that we can split P). On a conceptual level, this does not require any changes whatsoever.

4.2 Stateful properties

Performing ≤ 1000 I/O operations To ensure that no more than 1000 values will be read or sent, the driver needs to keep some internal state, e.g. a single integer value n (at location l_n) representing the number of times it has been called.

To accommodate this local state, the initial world will need to contain one extra region for location l_n, that allows any integer values ≥ 0. Since this region only has a single state, we can again short-circuit the world and enforce the constraints we want on n through an invariant directly. We now discuss the concrete constraint that we are looking for.

We have to tie this integer n to the number of I/O events present in the trace t, otherwise it will prove impossible to verify our driver. Concretely, the invariant we will verify our driver under changes from

$\mathrm{trInv}^\iota(|t| \leq 1000)$ to $\mathrm{trInv}^\iota_{\mathrm{np}}(\exists n.\, \ulcorner |t| = n \wedge n \leq 1000 \urcorner * \boxed{\bullet\, n}^{\gamma_n}) * \mathrm{NaInv}^\iota(\exists n.\, l_n \mapsto n * \boxed{\circ\, n}^{\gamma_n})$, where we defined

$$\mathrm{trInv}^\iota_{\mathrm{np}}(P_{\mathrm{np}}) \quad \triangleq \quad \boxed{\exists t.\ \mathrm{tr}(t) * P_{\mathrm{np}}(t)}^\iota$$

(i.e., the predicate over the trace does not need to be pure any longer). Concretely, this means that instead of having a pure predicate P in our robust safety theorem, we should also allow non-pure predicates P_{np} that entail P, i.e. $\forall t.\, P_{\mathrm{np}}(t) \twoheadrightarrow \ulcorner P(t) \urcorner$, since we might have to enforce other invariants relating to local state over the trace as well.

Note that the following two formalizations do not work, since the trace and counter value are not updated at the same time in the code:

$$\mathrm{trInv}^\iota_{\mathrm{np}}(\ulcorner |t| \leq 1000 \urcorner * \exists n.\, l_n \mapsto n * n = |t|)$$
$$\mathrm{trInv}^\iota_{\mathrm{np}}(\ulcorner |t| \leq 1000 \urcorner * \exists n. \boxed{\bullet\, n}^{\gamma} * n = |t|) * \boxed{\exists n. \boxed{\circ\, n}^{\gamma} * l_n \mapsto n}^\iota$$

Using a non-atomic invariant in the first of these two formalizations would not work either, as it would break the adequacy (as stated earlier); we would not know that the safety property inside P_{np} was upheld at *any* point during execution.

Remark: I swept this under the rug in the above exposition, but if we allow boot code to run first, we have to make sure that the value n indeed corresponds to the number of I/O events that have occurred when control is passed to the adversary.

To make sure we can prove the driver specification, we will have to update the robust safety theorem slightly to incorporate these changes.

Given a macro instruction to perform loops, a well/mis-behaving client might look as follows: (TODO: write this out, and come up with some macro formulation)

Never performing I/O again after some stop token has been read on in Again, this requires no extra conceptual changes.

Verifying OCP's As Armaël noted at some point, it should be possible to simulate the proof-style we see in OCPL by emitting values onto the trace, instead of making assertions over them. This should give us the same proving power as the OCPL paper, and allow us to emulate their examples.

5 Proof sketch

This section describes the steps involved in proving the robust safety theorem stated above in some more detail.

5.1 Adapting WP and the FTLR

The `Load` and `Store` cases will require new WP rules, that do not mention points-to chunks but mention the trace resource instead (and specifically, we will need rules in the presence of invariants such as $\mathrm{trInv}^{\iota}(P)$ above), when interactions at MMIO locations happen. The general WP spec will need to distinguish between the MMIO and non-MMIO cases (alternatively, if both cases are too distinct, the ftlr can make this distinction again).

The relations $\mathcal{V}, \mathcal{E}, \mathcal{R}$ need to be updated in terms of $\overline{\mathsf{MMIO}}$ and the FTLR will need to be re-stated in terms of these updated relations. This should again only require some changes for the `Load` and `Store` cases, depending on how we choose to formalize the WP cases above.

5.2 Proving the driver safe

Specifically, we need to prove the condition

$$\forall a_i \in A_{\mathrm{entry}}. \begin{cases} \mathrm{trInv}^{\iota}(P) * \mathrm{code}(b, e, \overline{w_l}) \vdash \\ \forall W \sqsupseteq^{\mathrm{priv}} \mathrm{W}_{\mathrm{init}}.\ \mathcal{E}^{\overline{\mathsf{MMIO}}}(W)(\mathrm{RX}, g, b, e, a_i) \end{cases}$$

mentioned above, for the set A_{entry} of entry points that the boot code will actually hand closures out for. This will require reductive reasoning over the concrete code of the driver, but should not be too hard in the absence of local return pointers and local driver state.

5.3 Syntactic boot spec

Proving the syntactic boot specification given above, again through reasoning on the concrete boot code that we have written down in our example. This should be relatively easy, as our boot code erases adversary memory before handing

over control. The proof includes verifying any IO that the boot code might need to do, using the new WP rules mentioned above.

5.4 Semantic boot spec

The semantic boot specification should be derivable from the syntactic one, combined with the proof of safety of the driver (so that we know that enter capabilities to the driver are safe). The different elements of the semantic bootSpec continuation can be proven as follows:

- First off, we can prove

  $$\exists \overline{\gamma}.\, \mathrm{region}(\emptyset) * \mathrm{sts\_full\_world}\ \mathrm{sts\_std}\ \emptyset$$

  for *some* ghost names $\overline{\gamma}$ (we left this existential out for simplicity in our spec). This will suffice, since all our other proofs are parametric in the concrete ghost names, and we can use these specific ghost names to construct an instance of STSG. We can then populate the world with a standard, permanent region for the adversary's memory, using the chunks $\ast_{m\in[b_{adv},e_{adv})} m \mapsto_a \mathrm{inl}\ \_$ from the syntactic boot spec. This allows us to prove

  $$\exists \overline{\gamma}.\, \mathrm{region}(\mathrm{W}_{\mathrm{init}}) * \mathrm{sts\_full\_world}\ \mathrm{sts\_std}\ \mathrm{W}_{\mathrm{init}}$$

  in the semantic bootspec. We could generalize the semantic bootspec slightly, by only requiring $[b_{adv}, e_{adv})$ to be a subset of adversary memory, but it is not immediately clear what we gain from that.

- $\mathrm{trInv}^{\iota}(P)$ and $\mathrm{code}(b, e, \overline{w_l})$ are trivially proven by allocating invariants for $\boxed{\circ\, t}^{\gamma_{\mathrm{T}}} * \ulcorner P(t) \urcorner$ and $\ast_{l\in[b,e)} m \mapsto_a w_l$ respectively.

- In proving that

  $$\ast_{r\in Reg\setminus\{\mathrm{PC}\}} (r \mapsto_r w_r * \mathcal{V}^{\overline{\mathsf{MMIO}}}(\mathrm{W}_{\mathrm{init}})(w_r))$$

  holds, we can distinguish two different cases:

  - For registers a_i containing closures to the driver, we use the chunks $\ast_{a_i\in A_{\mathrm{entry}}} r_{a_i} \mapsto_r (\mathrm{E}, g, b, e, a_i)$ from the syntactic bootspec, together with the proof of driver safety. To be able to apply the latter, we have to use the $\mathrm{code}(b, e, \overline{w_l})$ and $\mathrm{trInv}^{\iota}(P)$ invariants we allocated earlier.
  - All other registers will contain integers $\mathrm{inl}\ z$ and will hence trivially be in $\mathcal{V}$. We can just use the chunks $\ast_{r\in Reg\setminus\{\mathrm{PC}, A_{\mathrm{entry}}\}} r \mapsto_r \mathrm{inl}\ \_$ to satisfy the proof obligation.

- The proof of

  $$\mathrm{PC} \mapsto_r (\mathrm{RWX}, \mathrm{G}, b_{\mathrm{adv}}, e_{\mathrm{adv}}, b_{\mathrm{adv}}) * \overline{\mathcal{V}^{\mathsf{MMIO}}}(\mathrm{W}_{\mathrm{init}})(\mathrm{RW}, \mathrm{G}, b_{\mathrm{adv}}, e_{\mathrm{adv}}, a_{\mathrm{adv}}))$$

  follows directly from $\mathrm{PC} \mapsto_r (\mathrm{RWX}, \mathrm{G}, b_{adv}, e_{adv}, b_{adv})$ in the syntactic boot spec, combined with the shape of $\mathrm{W}_{\mathrm{init}}$ (i.e. the PC points to a standard, permanent region).

- The execution condition for the PC with respect to $\mathrm{W}_{\mathrm{init}}$ can be proven using the fundamental theorem, since we know from the previous bullet that both the read- and write-conditions must be satisfied for the PC register. This completes the proof of $\overline{\mathcal{V}^{\mathsf{MMIO}}}(\mathrm{W}_{\mathrm{init}})(\mathrm{RWX}, \mathrm{G}, b_{\mathrm{adv}}, e_{\mathrm{adv}}, a_{\mathrm{adv}}))$.

5.5 Proving the robust safety theorem

Given a proof of the semantic boot spec, it should be fairly simple to prove that the robust safety theorem holds.

To prove that our entire execution is safe, we will first of all need to prove that $\mathrm{bootCont}_{\mathrm{sem}}(P, g, b, e, A_{\mathrm{entry}}, \Phi, \mathrm{W}_{\mathrm{init}})$ holds. This should be simple, since part of the precondition is that the PC is in the execution relation, which is essentially what we need to prove (under sensible -i.e. safe- memory and registers).

When we know that our entire execution is safe according to the WP predicate, we can apply adequacy of the weakest precondition (and the fact that we have an invariant stating that our predicate P must hold), to prove $P(t)$ over the entire I/O trace t.

A Remarks

A.1 Operational Semantics

A.1.1 Non-determinism

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.

One possible downside is that this makes the operational semantics non-deterministic, which might or might not be an issue in the future.

There are several ways in which the operational semantics could be made "more deterministic".

One could additionally parameterize the operational semantics by an input-stream in. The updated IOLoad load rule might then look something like

this:

IOLoad

$$\frac{\text{readAllowed } p \qquad a \in [b, e) \qquad r[\text{src}] = (p, g, b, e, a) \qquad a \in \mathsf{MMIO} \qquad n = \#\{i \mid t[i] = (\text{IORead}, \_, \_)\}}{(r, m, t) \xrightarrow{\texttt{Load dst src}} (r[\text{dst} := \text{in}[n]], m, t +\!\!+ (\text{IORead}, a, \text{in}[n]))}$$

When parameterizing our development with multiple values, we could either write them out as different Coq `Parameters` where we need them, or store them as fields in a `DriverG` typeclass (the latter is the approach taken in e.g. the code-base of [SGDL19]).

One could also take as a parameter a predicate that restricts the set of traces that are allowed (that is what [SGDL19] do).

We could also express more precisely the devices' model, and equip each device with 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].

A.1.2 PC in MMIO

Additionally, we have the option of disallowing an execution with a PC register pointing to a MMIO location in the semantics. First off, consider that this is an edge case; since we always have points-to chunks describing the layout of code in memory (and a points-to chunk for location l doubles as a proof that $l \notin \mathsf{MMIO}$), it is trivial to prove that this case would not occur in any of our current examples. If we make no changes to the current code base, this behavior would be allowed, and the following would happen in both cases described above:

1. An unconstrained value is read from the location. Any instruction could be executed.

2. The predicate `MemLocate` in `lang.v` will make sure that the default value (e.g. 0) is read and decoded into an instruction when we try to execute an MMIO location. This would not cause a conflict with the current WP rules, since a points-to chunk is required in the precondition, ensuring that this case does not occur. Another alternative is to read the encoding of a NOP instruction, instead of just the value 0.

If we were to simply disallow execution when the PC points into MMIO, we would make the semantics of all instructions slightly more complicated, but in a uniform way. We could simply add $a \notin \mathsf{MMIO}$ to the predicate `isCorrectPC` (or add it as an additional predicate), which is used to define failing execution. It would again not be a drastic change, since a points-to for the address a the

PC points to is required by the WP rules regardless, from which we can prove that $a \notin \mathsf{MMIO}$.

A.2 Resources

A.2.1 MMIO overlapping with m

For the case where the MMIO locations are still part of the heap m, this would look as follows:

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

Sanity check: we never have allocation of new locations l in our capability machine, but if we did, that would work in both settings by requiring that $l \notin \mathsf{MMIO}$ in the operational semantics rule for Alloc.

A.2.2 Fine-grained trace resources

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 $\boxed{\circ\, t'}^{\gamma_{\mathrm{T}}}$, 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.

A.2.3 MMIO in the heap

As an alternative to having the set of MMIO addresses (MMIO) as a global parameter, we could 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:

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

(where Auth(Ex(Trace)) is the resource algebra).

A.2.4 Dynamic MMIO

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.

A.3 Toplevel theorem

A.3.1 $\mathcal{V}^{\Psi}$ with arbitrary Ψ

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 $[b, e) \cap \mathsf{MMIO}_{\text{pub}}$ with $\mathsf{MMIO}_{\text{pub}}$ the shareable subset of addresses in MMIO.

A.3.2 Driver with single entrypoint

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. Another alternative is to provide an opcode to specify the operation that needs to be performed. This is slightly less general, since it does not allow providing read access only to an adversary.

A.3.3 OCPL and explicit worlds

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.

A.3.4 Less restricted but still syntactic check for the adversary memory

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:

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

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

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.

[SGDL19] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.