Mechanised Robust Safety for Virtual Machines Communicating above FF-A

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, Lars Birkedal
Aarhus University

Abstract—Thin hypervisors make it possible to isolate key security components like keychains, fingerprint readers, and digital wallets from the easily-compromised operating system. To work together, virtual machines running on top of the hypervisor can make hypercalls to the hypervisor to share pages between each other in a controlled way. However, the design of such a hypercall ABI remains a delicate balancing task between conflicting needs for expressivity, performance, and security. In particular, it raises the question of what makes the specification of a hypervisor, and of its hypercall ABIs, good enough for the virtual machines. In this paper, we validate the expressivity/security balance of the design of the hypercall ABIs of Arm’s FF-A. We formalise a substantial fragment of FF-A as a machine with a simplified ISA in which hypercalls are steps of the machine. We then develop a separation logic, which we prove sound with respect to the machine execution model, and use it to reason modularly about virtual machines which communicate through the hypercall ABIs, demonstrating the hypercall ABIs’ expressivity. Moreover, we use the logic to prove robust safety of communicating virtual machines, that is, that even if some of the virtual machines are compromised and execute unknown code, then they cannot break the safety properties of other virtual machines running known code. This demonstrates the intended security guarantees of the hypercall ABIs. All the results in the paper have been formalised in Coq using the Iris framework.

1. Introduction

A verification effort can only ever be as good as the specification it relies on. This is especially true for key security components like hypervisors, where a single error in design can void all security guarantees. Specifications for real-world programs are sizeable programs themselves, and thus commonly suffer from bugs themselves; and while some are found during the verification effort [1] §VI, this is not always the case [2]. Moreover, the verification effort does not necessarily validate the expressivity of the specification either. To address this, specifications themselves need to be validated and tested, in particular by exercising them to verify client code. In the terminology of DeepSpec, we need to make sure that specifications are ‘live’ [3], in that they are “connected via machine-checkable proofs to [not just] the implementation [but also to] client code”.

In this paper, we formalise and validate a substantial fragment of the hypercall (aka ‘hypervisor call’, HVC) ABI of FF-A, the Arm Firmware Framework for Arm A-profile [4], as implemented by Google’s Hafnium hypervisor [5]. The hypercall ABI allows virtual machines (VMs) running atop of a hypervisor to communicate and share data, e.g., by sending messages or by controlled sharing of memory pages, and to pass control to others. Our formalisation simplifies the ABI compared to the informal FF-A specifications, but still captures the essence (see Simplification paragraph in Section [2] for details). We then validate it by exercising it to verify key scenarios of VMs using the ABI for controlled sharing of memory in the presence of adversarial, unknown code. Controlled sharing is essential for communication between VMs in real use cases, but makes the security analysis of hypervisors much challenging.

Our running example is that of Figure [1] where the ‘primary’ VM (typically, Linux) is privileged, and can ask the hypervisor to schedule other, ‘secondary’ VMs (typically, the keychain, or DRMs). Here, we have two secondary VMs, one running known code, VM1, and one adversarial, running unknown code, VM2; each VM has its own pages, disjoint from that of the others. The primary VM, VM0, first asks the hypervisor to share one of its pages with VM1; then asks the hypervisor to run the adversarial VM2; and, when given back control, asks the hypervisor to run the known VM1.

Dealing with the HVC ABI and its underlying use of virtual memory adds many components to the machine state: page tables, in-flight memory sharing transactions between VMs, etc. Managing the size and details of such a machine state poses a significant proof engineering challenge. For reasoning to be tractable, we need to be able to reason about known VMs individually: we should only need to consider the relevant parts of the machine state, and only need to take interference into account at interaction points, not at every step of the program. To this end, we develop VMSL, a novel higher-order separation logic that supports formal modular reasoning about the execution of communicating VMs.

One key intuitive desired security guarantee is robust safety: no matter what HVCs the adversarial VM2 may invoke, it will not be able to affect the private pages of VM0 and VM1, nor the page shared between only VM0 and VM1. It requires carefully a designed ABI, posing constraints to each HVC, making sure the desired guarantee is not breakable in any case, which results in a sophisticated and lengthy informal FF-A specification [4]. In this paper, we describe how to capture robust safety formally, even in the presence of in-flight transactions between VMs, and how to prove that the ABI specification enforces robust safety.

We highlight the following features of our VMSL logic:
VMSL is factored into two parts: a general part that handles issues that arise for any low-level model with scheduling, and a specific part that deals with the HVC ABI of FF-A.

VMSL supports modular reasoning in the sense that each VM can be verified individually. This is crucial for formal verification to work at scale.

VMSL features two compatible resource sharing mechanisms to support reasoning about communication among VMs: (1) standard separation logic invariants and (2) resumption conditions, a novel sharing mechanism specialised to our setting that offers more convenience than standard invariants.

VMSL is sufficiently expressive to support not only formal reasoning about concrete known programs but also the definition of so-called logical relations which can be used to reason about robust safety. We use logical relations to reason about scenarios like that of Figure 1 where some VMs run known code and others run unknown possibly adversarial code.

Contributions.

We formalise a substantial fragment of the Arm’s FF-A ABI, as implemented by Hafnium, as an operational semantics in which HVCs are primitive steps (Section 2).

We develop and prove soundness of VMSL, a novel separation logic for modular reasoning about communicating VMs (Section 3).

We show how we capture the desired security guarantees using logical relations and how we apply them to reason about robust safety (Section 4).

All of our results are mechanised in Coq using the Iris program logic framework [6] and the Iris Proof Mode [7]. Reviewers can find the anonymized Coq code at https://www.dropbox.com/s/yzervl3hgmaqojc/snp.zip?dl=0.

Non-goals. We focus on exercising the HVC ABI, and thus do not address other key complementary aspects, which we discuss in Section 5. In particular: (1) We are not verifying a hypervisor, but rather making sure that the hypervisor specification that we are providing is adequate. (2) We focus on the HVC ABI, and our operational semantics is a minimalistic instruction set: it has the right shape, but it is far from a full-scale ISA. (3) Our operational semantics assumes that there is no concurrency, as characterising the semantics of virtual memory in a concurrent setting is work in progress [8].

Threat model. We only consider integrity, not secrecy. Our attacker model is that of adversary VMs running unknown code; we do not consider side-channels. To reason about adversarial VMs running unknown code, we only assume knowledge of initially accessible pages and transactions related to adversaries; both the content of memory and registers of adversaries are unspecified. Adversaries therefore could perform attacks by executing malicious code stored in its memory. For instance, adversaries could invoke arbitrary HVCs to try to interfere in-flight transactions between trusted VMs, or read/write memory of other VMs. With this model, we show that adversaries cannot break the integrity of memory under protection of hardware and the hypervisor.

2. Formalising a substantial fragment of the HVC ABI

As we focus on the HVC ABI, we use a simplified subset of the Arm-A instruction set, with only one unusual feature: the hvc instruction. Figure 2 shows the running example of Figure 1 more precisely in our language.

We partially specify the hardware behaviours of virtualisation, including page table lookup and context switching, plus the following HVCs of FF-A that are supposed to be provided by hypervisors to VMs used in ‘no-secure world’: (1) for memory sharing: Donate, Lend, Share, Retrieve, Relinquish, and Reclaim; (2) for scheduling: Run, Yield, and Wait; and (3) for messaging: (asynchronous) Send and Poll. We omit the the synchronous variant of send, which requires extra machinery without increasing expressivity, and the new messaging HVC, notify, that was introduced after this work started. The ‘secure world’ part of FF-A is out of the scope of the paper.

Simplification. We make two main simplifications in our model of FF-A: (1) We only formalise the ownership and access fields of the page table entries, and only consider read-write-execute permissions. (2) We only model 1-to-1 sharing (as implemented by Hafnium) instead of 1-to-\(n\), and accordingly simplify the format of the transaction descriptors. These help keep the size of our model manageable, but do not significantly impact expressivity, and we believe the model can be easily scaled to 1-to-\(n\).

2.1. Formalising HVCs

Informally, a hypervisor provides the illusion to VMs that they are running on a machine in which the whole HVC
Figure 2: Code of the two known VMs in Figure 1. $x$ is a constant standing for the data stored in $R0$ that VM0 will share with VM1; $p$ is the page that VM0 will share (represented with the base address of the page); $px$ is the base addresses of the TX page of VM0, and $prx$ is the RX page of VM1. We assume the two programs live at the start of two separate pages, $pp$ and $pp'$. is just a step of the machine; the hypervisor itself is invisible. Accordingly, in our model, an HVC is a primitive step of the operational semantics. The reduction rule for a Share in Figure 3 is a representative example (we explain this rule in the following).

Memory Access. On a concrete machine, an hvc causes a jump to a higher exception level and the execution of hypervisor code. The hypervisor code operates on its private data in physical memory; in our model, the private state of the hypervisor is represented abstractly, separate from the physical memory that the VMs operate on, which we model as a partial function from memory addresses to machine words (both are represented by our type of machine words, Word).

In particular, on a concrete machine, the page tables are in-memory data structures that are edited by the hypervisor and looked up by the hardware; in our model, the page tables are merged into one partial (mathematical) function that is updated by memory-sharing HVCs. The partial function maps a page identifier (page base address, which is sufficient, given that we assume identity address mappings) to a page status, which is composed of an optional page owner, a bit indicating whether it is exclusively owned (can only be accessed) by one VM, and the set of VMIDs of the VMs that have access to the page. For instance, the status of page $p$ in the example of Figure 2 is initially $(\text{Some}(0), \text{True}, \{\})$, since VM0 has exclusive ownership on the page; and it is updated to $(\text{Some}(0), \text{False}, \{0\})$ after the page is shared with VM1.

When a VM with VMID $i$ tries to perform a memory access at an address $a$, e.g. $\text{str}$ at line 3 storing the value in $R5$ to address $p$, the page status of the page $p$ is looked up in the page table, and checked to determine whether the VM is allowed to access $p$ (which it can in this case, since 0 is an element of the ‘accessible’ set $\{0\}$). If the access is not allowed by the page table, a page fault is raised. In our setup, this terminates the execution (of all VMs, because there is no concurrency) with the execution mode PageFault, and therefore a page fault is safe.

Configuration. A state of our operational semantics is composed of the aforementioned components for modeling memory access, plus those for HVCs:

\[
\text{State} \overset{\text{def}}{=} \{ \begin{array}{l}
\text{mem} : \text{Word} \to \text{Word} ; \\
\text{pgt} : \text{PageID} \to \text{PageStatus} ; \\
\text{regs} : \text{VMID} \to \text{RegisterFile} ; \\
\text{curr} : \text{VMID} ; \\
\text{trans} : \text{Transactions} ; \\
\text{mb} : \text{VMID} \to \text{Mailbox} ; \\
\end{array} \}
\]

A configuration is a pair of a state together with an execution mode. We have three execution modes: Normal, PageFault, and Halted.

The machine can only take a further step to execute the next instruction if it is in Normal mode. Halted is the mode reached by ‘normal’ termination via the halt instruction, and, as mentioned above, PageFault is used for page faults.

Transactions. On a concrete machine, to support memory sharing transactions between VMs, the hypervisor needs to maintain some metadata in its private memory; in our model, we keep a partial mapping from transaction handles (machine words) to abstract transactions, which are composed of the sender, the receiver, the set of pages being sent, the type of the transaction, and the state of it (a bit indicating whether the receiver has retrieved the access to the pages). For instance, the hvc at line 13 of VM0 invokes a sharing transaction of page $p$ to VM1, which is represented as $(\text{Some}(0, \{p\}, \text{Share}, \text{False}))$ (see the last line of antecedents in the rule in Figure 3).

A VM is allowed to send pages to other VMs via transactions. To do so, the sending VM first has to prepare

\[
\sigma.\text{curr} = i \quad \text{valid instr}(\sigma, i) = \text{Some}(\text{hvc}, a) \\
\text{valid share}(\sigma, i) = \text{Some}(i, s, h) \\
\text{mem} = \sigma.\text{mem} ; \quad \text{curr} = \sigma.\text{curr} ; \quad \text{mb} = \sigma.\text{mb} ;
\]

\[
\begin{array}{l}
\text{pgt} = \sigma.\text{pgt} [p \mapsto (\text{Some}(i, \{i\}, \text{False})] ; \\
\end{array}
\]

\[
\begin{array}{l}
\sigma' = \\
\begin{cases}
\text{Normal} & \to \text{Normal} ; \\
\text{Halted} & \to \text{Halted} ; \\
\text{PageFault} & \to \text{PageFault} ; \\
\text{Share} & \to \text{Share} \end{cases}
\]

Figure 3: Reduction rule for Share.
a transaction descriptor specifying the receiver and the page IDs of the pages in its TX page (lines 4–9 in the example). Next, the sending VM invokes a memory sending HVC, asking the hypervisor to create a transaction of a type with the given descriptor. The type of transaction (Donation, Sharing, or Lending) determines the effect of the HVC on the status of the pages being sent, as per Figure 4. The sharing of page p in the example corresponds to edges (1) and (3). In all cases, the hypervisor checks that the pages are owned and exclusively accessible by the sender before creating the transaction (e.g. done by valid_share in Figure 5). If the checking fails, the hypervisor returns an error code to the VM and resumes its execution. If it succeeds, the hypervisor then returns a fresh handle h (initially mapped to None, meaning that it is not bound to any transaction) referring to the newly created transaction to the sender, and remembers the transaction in its meta data (trans).

VMs can invoke other HVCs with the same handle to refer to the transaction. For instance, with the hvc at line 10, VM1 Retreives access to the page, flipping the retrieved bit to True. In case of donation, this HVC also transfers ownership of the pages to the receiver and finishes the transaction (and frees the handle). In case of sharing or lending, the receiver could Relinquish access to the pages afterwards, flipping the bit back. The sender can Reclaim the exclusive access to the pages if the access has not been retrieved, or has been relinquished by the receiver (in either case, the retrieved bit is False), which is the second way of ending the transaction.

**Scheduling.** On a concrete machine, to support switching between VMs, the hypervisor needs to save registers by spilling them in its private memory, and restore them upon context switching; in our model, we keep a total mapping (regs) from VMIDs to register files (a register file is itself a map from register names to words), and a VMID (curr) to remember which VM is currently running.

By duplicating RegisterFile and picking the right one to update according to curr when registers are modified, we avoid modelling register saving and restoring at context switching. For instance, mov at line 1 of VM0 only updates R5 of VM0 since curr is 0. As a consequence, the switching HVCs only need to change curr.

FF-A allows putting the responsibility of scheduling VMs either on the hypervisor, or delegates it to VM0, the ‘primary’ VM. Typically, thin hypervisors like Hafnium choose the latter, for instance letting the thread scheduler of Linux make scheduling decisions.

Therefore, it grants the primary VM the privilege to Run other so-called secondary VMs. Secondary VMs are only allowed to return control back to the primary; either explicitly with Yield, or as the consequence of an HVC, for example to wait for a message with Wait.

**Messaging.** To support messaging between VMs, on a concrete machine, the hypervisor needs to maintain two dedicated memory pages, named TX and RX, as the message buffers for each VM, and remembering the state of all RX buffers (e.g. whether the buffer is full); in our model, we keep a total mapping (mb) from VMID to Mailbox, which consists of two buffers.

The TX and RX buffers are respectively write-only and read-only, and are used for sending and receiving messages between two VMs, or a VM and the hypervisor. Line 21 of VM0 sends the handle referring to the sharing transaction to VM1. The hypervisor copies the handle from the TX page of VM0, pastes it to the RX page of VM1, and remembers the length and the sender in its private state as Some(1, 0). In the case when the sender is a secondary VM, the control is yielded to the primary immediately, notifying it that a message has just been sent to the receiver, so that the primary could schedule the receiver to run next to actually receive the message. The receiver, like VM1, can ask for the length and the sender of the message with Poll (line 5 of VM1), which also notifies the hypervisor that it is ready to take the next message (updates the RX buffer to None).

**Calling Convention.** The calling convention that we have used in the example above works in general as follows: to invoke a specific HVC, a VM executes the hvc instruction with the identifier of the HVC in R0, and other arguments saved in successive general-purpose registers (for example, the identifier of the VM to Run in R1), or in the TX buffer (for “large” arguments like transaction descriptors), as appropriate. Return values, including whether the HVC is successful and possible error codes, are passed back to the VM via return registers, like in Figure 3 or RX buffers (depending on the HVC).

### 2.2. Conformance

As with any formal modeling activity, there is an unavoidable gap between the informal FF-A specification and our formal specification. We have, of course, tried to follow the intent of the informal specification when designing our formal model, and we have also studied the Hafnium implementation of the informal FF-A spec to gain more confidence in our formal model. Future work includes showing that some of the Hafnium hypercall implementations refine our formal specification.

### 3. Reasoning about communicating VMs

To validate our model of the FF-A HVC ABI, we develop a program logic, called VMSL, to reason about
key scenarios of virtual machines communicating using this ABI. We start this section by discussing two of the key challenges involved in developing a program logic for communicating VMs.

The programs running on VMs are imperative and operate on mutable shared data and thus we will base VMSL on separation logic \[9\]. In particular, this will allow us to support local reasoning via the celebrated frame rule of separation logic, as will be exemplified later.

The first challenge is that we wish to reason about a low-level language model where instructions are stored in the memory, which complicates the formulation of a sequential composition proof rule, which allows one to reason about each instruction at a time. We address this challenge by developing a novel form of specification (a novel kind of weakest precondition) with built-in support for reasoning about each instruction at a time. We discuss how our approach relates to previous work on program logics for assembly in Section 5.

The second key challenge is that we wish to support ‘VM-local’ reasoning: it should be possible to verify each VM individually. This is analogous to ‘thread-local’ reasoning in concurrent separation logic and is crucial for formal verification to work at scale. Of course, we can treat each VM in a manner similar to how a thread is treated in concurrent separation logic and then use concurrent separation logic style invariants to reason about sharing of data among different VMs. However, such invariants for concurrency are somewhat complicated to use: since they have to be maintained at every step of execution, one can only get access to the shared resources in the invariant for an atomic step and one has to reestablish the invariant after each atomic step. However, this poses an undue burden in our setting where VMs are executed sequentially and not concurrently. Thus we introduce resumption conditions, an alternative mechanism to share resources among VMs, which allows a VM to use shared resources freely during its execution until control is transferred to another VM. We explain these on our example in Section 3.2 and describe them in more detail in Section 5.5.

We have proven soundness of VMSL with respect to the operational semantics of the machine model. Thus we have proven that all of our proof rules are sound with respect to our definition of weakest precondition and we have proven an adequacy theorem which intuitively says that if a weakest precondition holds in the VMSL, then it really means that it is safe to execute the program on the machine. We refer the reader to our Coq formalisation for a precise formal statement of the soundness and adequacy theorems and the proofs thereof.

3.1. VMSL

In this section we introduce VMSL by explaining how it is used to specify and reason about VMs executing known code. Concretely we use a simplified variant of Figure 2 without the unknown VM2 (that is, with lines 22–25 of VM0 removed) as a running example.

**Informal specification.** In this example, the primary VM writes the content \( x \) of register R0 to the first location of page \( p \), shares the page with VM1, then schedules VM1. VM1 retrieves access to the page \( p \), increments the first location of \( p \) by two, then yields. The primary VM then reads from \( p \) into R0, and halts. We want to show that it reads \( x + 2 \).

**Register state.** To state this formally, we introduce the register ‘points-to’ assertion, \( r \mapsto v \), which captures the fact that register \( r \) contains the value \( v \); because our registers are banked, we specify which VM the register belongs to via its VMID, \( i \). As usual in separation logic, our assertion also captures ownership of register \( r \) of VMID \( i \), so that this assertion is exclusive. In Figure 6 we present a collection of similar points-to predicates of VMSL, together with their intuitive meaning. Most of them will be introduced gradually along with our explanation of how we use VMSL to reason about the example.

**Formal specification.** Returning to the example, starting from a state where \( R0 \mapsto x \), with some side conditions we introduce below, we want to show that, when the machine terminates, VM0 reaches a halted state (indicating success), and moreover we have \( R0 \mapsto x + 2 \). We phrase this in VMSL by using a weakest precondition predicate which expresses the partial correctness of the program, i.e., we execute the program and if it terminates, then the postcondition holds:

\[
\begin{array}{c}
R0 \mapsto x \ldots \text{(other resources)} \\
\vdash WP \text{ Normal } 0 \{m.m = \text{Halted} \land R0 \mapsto x + 2\}
\end{array}
\]

3.2. Proving the specification

**First instruction.** To safely execute the first instruction of VM0, \texttt{mov R5 \#p} (where \( p \) is an immediate), we need, as captured in our \texttt{SS-mov} proof rule for an immediate-to-register mov:

\[\begin{array}{l}
(1) \text{The value } a \text{ of the program counter, which indicates the location of the current instruction in the memory, as captured by the register points-to for registers } \text{pc} \mapsto a \text{ (here, } \text{pc} \mapsto \text{pp}).

(2) \text{Knowledge that the page at address } a \text{ (here, } pp) \text{ is in the accessible set of } PageIDs...

(3) \ldots \text{that are mapped for the current VM, as captured by ownership of the page tables points-to assertion, } \text{Pgt} \mapsto s \text{ (here, } \text{Pgt} \mapsto s).

(4) \text{Ownership of the memory points-to-resource for that memory location, } a \mapsto w \text{ (here, } pp \mapsto w) \text{, which contains a word } w \text{ that is the encoding of an immediate-to-register mov instruction (here, mov } R0 \#p).\n
(5) \text{Ownership of the register points-to resource for the affected register (here, } R5 \mapsto \text{False}); \text{ we do not need}
\end{array}\]

Thus \( R0 \mapsto v \mapsto v \mapsto \text{False} \), where \( * \) is the separating conjunction of separation logic.
### SS-mov

$\text{SSWP Normal } @ i \begin{cases} (\text{False,Normal}) \end{cases}$

$$\begin{align*}
(1) & \ \text{pc} \circ @ i \xrightarrow{\text{reg}} a \ast \ast (a \in_p s \ast \ast (3) \ Pgt \circ @ i \xrightarrow{\text{acc}} s \ast \ast (4) \ a \xrightarrow{\text{mem}} \text{encode(mov } r \ n) \ast \ast (5) \ r \circ @ i \xrightarrow{\text{reg}} - \end{align*}$$

Figure 5: The proof rule for an immediate-to-register mov instruction. The updated resources are highlighted. For simplicity, we use $\text{IsInstr}\circ @ i(s,a,mov r n)$ to represent that mov instruction is stored at address $a$ which belongs to the page that is one of VM$i$’s accessible pages $s$ (1) to (4) in later rules.

#### Predicate

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r \circ @ i \xrightarrow{\text{reg}} w$</td>
<td>register $r$ of VM$i$ contains word $w$</td>
</tr>
<tr>
<td>$a \xrightarrow{\text{mem}} w$</td>
<td>value at $a$ in memory is $w$</td>
</tr>
<tr>
<td>$\text{MemPage}(p,ws)$</td>
<td>content of page $p$ in memory is $ws$</td>
</tr>
<tr>
<td>$\text{Pgt} \circ @ i \xrightarrow{\text{acc}} s$</td>
<td>VM$i$ has access to pages $s$</td>
</tr>
<tr>
<td>$\text{Pgt} \circ @ i \xrightarrow{\text{own}} i$</td>
<td>VM$i$ owns page $p$</td>
</tr>
<tr>
<td>$\text{Pgt} \circ @ i \xrightarrow{\text{excl}} i$</td>
<td>VM$i$’s access to page $p$ is exclusive</td>
</tr>
<tr>
<td>$\text{Tran} \circ @ h \xrightarrow{\text{tran}} t$</td>
<td>transaction $t$ is bound to handle $h$</td>
</tr>
<tr>
<td>$\text{Tran} \circ @ h \xrightarrow{\text{status}} b$</td>
<td>status of transaction bound to $h$ is $b$</td>
</tr>
<tr>
<td>$\text{FreshHandles}(hs)$</td>
<td>handles $hs$ are fresh</td>
</tr>
<tr>
<td>$\text{Mb} \circ @ i \xrightarrow{\text{tx}} p$</td>
<td>VM$i$’s RX page is $p$</td>
</tr>
<tr>
<td>$\text{Mb} \circ @ i \xrightarrow{\text{tx}} p$</td>
<td>VM$i$’s TX page is $p$</td>
</tr>
</tbody>
</table>

Figure 6: Selected collection of resources of VMSL

to know what it contains (as signified by the use of $\neg$), but we must have the right to update it.

After the mov, the VM does not lose the control (so the switching bit is $\text{False}$), and the execution mode is still $\text{Normal}$. We get the updated resources back in our context; in particular, the program counter has been incremented, $\text{pc} \circ @ i \xrightarrow{\text{reg}} p + 1$, and the register now contains the immediate, $\text{R5}(0) \xrightarrow{\text{reg}} p$; the page tables and the instruction have not been affected, so we get their assertions back unchanged.

The proof rule requires exactly the resources needed to safely execute the instruction; other resources are implicitly kept unchanged via framing, which is a key feature of separation logic that saves us from maintaining global resources all the time, and helps reduce the proof effort.

The SS-mov rule, and all other single-instruction proof rules, use SSWP, our single-step variant of weakest preconditions. Informally, SSWP is like a weakest precondition that only specifies the behaviour of a single step (an instruction); applying a proof rule using SSWP takes resources specified in the premise, and returns resources stated in the postcondition, with the resulting execution mode and a bit indicating whether the instruction would cause the VM to lose control of the machine (the hypervisor switching to another VM to execute). SSWP underpins our approach to reason about one instruction at a time. We show its definition in the Appendix, and how to formally apply it to a weakest precondition in Section 3.3.

### Sharing

The following instructions prepare the descriptor and arguments for the Share HVC at line 12. They only involve register manipulations, which can be reasoned about in a similar way to the previous one, and memory access. To reason about memory access instructions, including ldr and str, we need memory points-to predicates, with side conditions checking whether the VM has the permission to access the address, similar to (2) of the SSS-mov rule.

Before reasoning about this specific Share, let us first consider the expected behaviour of a general Share hypercall, specified by the SSS-share rule.

To share pages represented by a set of PageIDs $ps$, VM$i$ invokes a Share hypercall with a descriptor in its TX page describing information about the transaction. Therefore, the proof rule requires (4) that the TX page is $ptx$, (5) ownership of that page, and knowledge that the contents of that page encode the descriptor $\text{memtx}$, which is expressed as memory points-to predicates for all of the words of that page, connected by $\ast$, and (1) knowledge that the descriptor in $\text{memtx}$ is valid. In addition, after validating the descriptor, the page table will be examined to check whether VM$i$ is allowed to share those pages.

Therefore, (6) requires page ownership $\text{Pgt} \circ @ p \xrightarrow{\text{own}} i$ and exclusiveness $\text{Pgt} \circ @ p \xrightarrow{\text{excl}} i$ True to VM$i$ of each page $p$ in $ps$. The side condition (2) plus the resource for page access further ensure that VM$i$ has access to those pages. This information combined ensure that VM$i$ is allowed to share pages $ps$. To initiate a transaction, the hypervisor has to allocate a fresh transaction handle $h$, which is ensured by (7) remembering the set $hs$ of available handles, and (3) requiring it not be empty. The hypervisor further binds $h$ to the meta-information and the state of the transaction that are also represented as resources as in the post condition.

In our example, VM0 shares a single page $p$ to VM1, so we let $i$, $j$, and $ps$ be 0, 1, and $\{p\}$ respectively. (1) is justified by the previous instructions constructing the descriptor correctly; (2) is justified as we assumed $s$ to be $\{pp; p; ps\}$. (3) can be justified if we assume a non-empty $hs$ in the specification. After applying the proof rule, we get $\text{Tran} \circ @ h \xrightarrow{\text{tran}} (0,1,p,\text{Share})$ and $\text{Tran} \circ @ h \xrightarrow{\text{status}} \text{False}$, stating that the requested transaction has been initiated, and is bound to $h$, which is also returned to VM0 so that it can refer to the transaction.

#### Messaging

To retrieve access to the shared page $p$, VM1 has to refer to the transaction with the handle $h$. To let VM1 do so, VM0 passes $h$ to it by messaging at lines 14–21. Messaging essentially copies from the sender’s TX page and pastes to the receiver’s RX page; therefore, the proof rule for messaging requires the resources for the two pages and associated memory. We capture the state of the VM1’s
SS-SHARE
(1) \( \text{ValidDesc}(\text{mentx}, i, j, ps) \land (2) \text{ps} \subseteq s \land (3) \text{hs} \neq \emptyset \land \text{IsInstr}(i, s, a, hv) \ast \)

\[
R_{0@i} \xrightarrow{reg} \text{encode(Share)} = R_{1@i} \xrightarrow{reg} l \ast R_{2@i} \xrightarrow{reg} R_{1@i} \xrightarrow{reg} l \ast \ast (4) \text{Mb}_0 \xrightarrow{tx} ptx \ast
\]

(5) \( \text{MemPage}(\text{ptx}, \text{mentx}) \ast \) (6) \( \bigwedge_{p \in ps} (P_{gtx@p} \xrightarrow{own} i \ast P_{gtx@p} \xrightarrow{excl} \text{False}) \ast \) (7) \( \text{FreshHandles}(\text{hs}) \)

\[
\text{SSWP Normal } @ i \left\{ \begin{array}{l}
\text{SSWP Normal } \left\{ \begin{array}{l}
\text{False, Normal).}
\end{array}
\right.
\end{array}
\right.
\]

\[\begin{aligned}
\text{SS-RUN} \\
(1) i \neq 0 \land \text{IsInstr}(0, s, a, hv) \ast R_{0@0} \xrightarrow{reg} \text{encode(Run)} \ast R_{1@0} \xrightarrow{reg} i \ast (2) R_{C_{1/2@i}} \left\{ \Psi_i \right\} \ast (3) R_{C_{1@0}} (\ast) \ast
\end{aligned}\]

\[\begin{aligned}
(4) \left( \begin{array}{l}
\text{pc}_0 \xrightarrow{reg} a + 1 \ast a \ \text{mem} \ \text{encode(hvc)} \ast P_{gtx@0} \xrightarrow{acc} s \ast
\end{array}
\right)
\end{aligned}\]

\[\begin{aligned}
R_{0@0} \xrightarrow{reg} \text{encode(Run)} \ast R_{1@0} \xrightarrow{reg} i \ast \Phi_{othr} \ast R_{C_{1@0}} \left\{ \Psi_0 \right\} \\
\rightarrow \Psi_i \ast \Phi_{rest}
\end{aligned}\]

(5) \( \Phi_{othr} \ast
\]

\[
\text{SSWP Normal } @ 0 \left\{ \text{True, Normal, } R_{C_{1/2@0}} \left\{ \Psi_0 \right\} \ast \Phi_{rest} \}
\]

\[
\text{WP-SSWP} \ \\
WP \ m @ i \ \{ \Phi \} \vdash \text{SSWP } m @ i \ \{ \{b, m'\}, \{b \land \text{RCHold}(\text{@i}) \lor \lnot b\} \rightarrow \text{WP } \ m' @ i \ \{ \Phi \} \}
\]

RC-HOLD
\[
R_{\text{Hold}}@i \ast R_{C_{1/2@i}} \left\{ \Psi \right\} \vdash \Psi \ast R_{C_{1@i}} \left\{ \Psi \right\}
\]

Figure 7: Selected rules of VMSL

RX page with a resource RXState@1 \( \rightarrow \) Some(1, 0) in the example, expressing that VM0 has passed one word to VM1.

**Scheduling.** At line 25, VM0 runs VM1 to allow VM1 to receive the handle and retrieve page \( p \). To reason about such scheduling, we introduce a resumption condition for VM1. A resumption condition for a VMi, denoted as \( R_{C_{1@i}} \left\{ \Psi \right\} \), captures the resources \( \Psi \) that need to be handed over to VMi to resume its execution. We use resumption conditions to express communication protocols (reminiscent of session types [10], [11]) between VMs, and to transfer resources between VMs along the scheduling control flow. Accordingly, the proof rule for Run, **SS-RUN**, uses a resumption condition. Concretely, we have to show the following to apply **SS-RUN** when the primary VM, VM0, is about to run VMi:

1. The VM being run is not the primary VM itself.
2. VM0 has to satisfy the resumption condition of VMi, \( \Psi_i \). The fraction \( 1/2 \) indicates that the resumption condition is split into two halves, and only one half is required. We elaborate on this point later.
3. We may pick the resumption condition of VM0, \( \Psi_0 \), that VMi will have to satisfy to yield back to VM0.
4. The magic wand \( P \rightarrow Q \) is separation logic’s resource-aware implication. It is used here to express that with resources required by the rule (the first line) and (5), we can show \( \Psi_i \), intuitively the resources transferred to VMi, and the left over \( \Phi_{rest} \), i.e. the resources that are required by the rule, but not needed to show \( \Psi_i \), that are still owned by VM0 afterwards.
5. Other resources required to justify (4).

By picking the right \( \Psi_i \) and \( \Psi_0 \), we describe the protocol according to which shared resources are transferred between the VMs. In our example, we know that to run VM1, VM0 has to have written \( x \) to the page \( p \), shared the page, sent the handle, and run VM1. We express this in \( \Psi_i \) as follows:

\[
\Psi_1 \overset{\text{def}}{=} p \xrightarrow{\text{mem}} x \ast \text{Trans@h} \xrightarrow{\text{tran}} (0, 1, \{p\}, \text{Share} \ast \text{Trans@h} \xrightarrow{\text{rty}} \text{False} \ast \text{Mb}_0 \xrightarrow{\text{tx}} \text{prx} \ast \text{RXState}@1 \xrightarrow{\text{Some}(1, 0) \lor \text{prx} \xrightarrow{\text{mem}} \text{h} \ast \text{Run} \ast R_{0@0} \xrightarrow{\text{reg}} 1 \ast R_{C_{1/2@0}} \left\{ \Psi_0 \right\}
\]

Note that when VM1 yields back control to VM0, it needs to have established VM0’s resumption condition, so we also include \( R_{C_{1/2@0}} \left\{ \Psi_0 \right\} \) in \( \Psi_1 \). VM1 thus can refer to \( \Psi_0 \) and show it when yielding. In our example, we want to show that VM1 has incremented \( x \) by 2 and yielded. We express this in \( \Psi_0 \):

\[
\Psi_0 \overset{\text{def}}{=} p \xrightarrow{\text{mem}} x + 2 \ast R_{0@0} \xrightarrow{\text{reg}} \text{Yield} \ast R_{1@0} \xrightarrow{\text{reg}} 1
\]
To justify (4), we let \( \Phi_{\text{other}} \) be the first three lines of \( \Psi_i \), and \( \Phi_{\text{rest}} \) naturally be the resources that are in premise but not required by \( \Phi_i \). We get \( \Phi_{\text{rest}} \) and \( RC_{i/0} \{ \Psi_0 \} \) after applying the rule.

To explain how to get resources stated in \( \Psi_0 \) out, we first introduce \( RCHolds@i \). It assumes the resumption of \( VM_i \) and can interact with the resumption condition of \( VM_i \) by \( \text{RC-Hold} \). Intuitively speaking, the rule says that if we know the resumption condition of a VM, and the VM is indeed resumed, then the condition holds. \( \triangleright \Psi \) means that \( \Psi \) holds later, i.e., after taking a step in the underlying model (this is used to break circularity of definitions [12], [13]).

Back to the example, we already get \( RC_{i/0} \{ \Psi_0 \} \) in the postcondition, so we would be able to apply this rule and proceed the proof with the transferred-back resources in \( \Psi_0 \) if we have \( RCHolds@0 \) as well. For now, readers only need to know that we can actually get it for free, because we have baked it into the definition of weakest preconditions in a way that we can get it out when a switching just happened.

**Halting and suspension.** After loading the word \( x + 2 \) at \( p \) to \( R0 \), the execution of VM0 is terminated by a halt.

The proof rule updates the execution mode from Normal to Halted, and thus we obtain the postcondition of our initial specification, \( m = \text{Halted} \land R0 \triangleright \text{reg} \to x + 2 \), and conclude the proof.

Observe the 'VM-modularity': the proof of VM0 was done without considering the code of VM1; all we needed was an abstract characterization of the protocol governing the interaction between VM0 and VM1, as captured by the resumption conditions.

The proof of VM1 is similarly done without considering the code of VM0. The proof for VM1 concludes in a different way, as it does not terminate, but instead suspends via the Yield at line 18. Because our protocol specifies it will not be scheduled again, it suffices to show that when we resume it, we get an immediate contradiction.

### 3.3. More on SSWP and RC

The example above shows how single-step weakest preconditions and resumption conditions are two key components that make reasoning with VMSL manageable. We now discuss these features in more detail.

**Single-step weakest preconditions.** We develop single-step weakest preconditions, which allow us to reason about a single instruction at a time. Rule \( \text{WP-SSWP} \) shows the relation between weakest preconditions and single-step weakest preconditions: informally, it says that (setting aside the antecedent of the separating implication in the postcondition) to reason about a list of instructions, we can reason about the first one, and then the rest. This gives us, for our assembly language, the type of sequential composition we expect from higher-level languages. We can always apply \( \text{WP-SSWP} \) to transform a goal formulated in terms of weakest preconditions into one formulated in terms of single-step weakest precondition, so that we can apply a proof rules for an individual instruction, and then proceed with the reasoning of the remaining instructions.

**Resumption conditions.** We achieve modular reasoning between VMs through RCs. To ensure that the entire logic integrates with resumption conditions, we bake \( RCHolds \) into the definition of weakest preconditions, so that we have to prove \( RCHolds \) when giving away the control, and in exchange we can assume it when getting the control back as in the postcondition of \( \text{WP-SSWP} \). Doing so allows us to write specifications for individual VMs, and prove them separately without needing to reason about other VMs’ private state, and only having to reason about the private resources of the current VM and the shared resources that are transferred according to the communication upon scheduling. If a yielding (or scheduling) just happened, we immediately get to assume \( RCHolds \), and we can obtain ownerships of the transferred resources stated in the resumption condition by \( \text{RC-Hold} \) to continue the reasoning.

Then, to combine the proofs of the local specifications, we have to make sure that the resumption conditions are consistent and compatible, i.e. combined together, they form a unified global protocol, and therefore the combined global specification is valid. To do so, we use the fractional permissions of separation logic [14], [15]: we split the RC of a secondary VM in two halves, and let the primary VM and that secondary VM own one half each. Owning half is safe for both VMs, since \( \text{SS-run} \) requires merely half to run the secondary, and \( \text{RC-Hold} \) requires merely half to obtain ownerships of the resources in the RC.

In the example, the protocol is specified by the RC of VM1 with RC of VM0 embedded into it. The RC of VM1 is split into two fractions owned by the two VMs so that the two conform to the same protocol.

**Recursive resumption condition.** We have shown in the example above how we can embed one resumption condition into another to construct a run-and-yield protocol between two VMs. In fact, our logic more generally supports recursively defined resumption conditions, which are useful for reasoning about examples where the number of switchings is unknown or infinite. Consider a ‘ping-pong’ example, in which a primary VM and a secondary VM do nothing but keep running each other; we can model this protocol as follows:

\[
\Psi_i \overset{\text{def}}{=} R0 \triangleright \text{reg} \to \text{Run} \ast R1 \triangleright \text{reg} \ast i \ast

RC_{i/0} \{ R0 \triangleright \text{reg} \to \text{Yield} \ast R1 \triangleright \text{reg} \ast i \ast \}
\]

### 4. Reasoning in the presence of unknown VMs

In our full motivating example in Figure 1, VM0 runs an unknown VM2 before running VM1 to let it retrieve the shared page. We assume that page \( pp_2 \), a page that VM0 and VM1 have no access to, is the only page that VM2 has access to except for its mailbox pages. Since the hypervisor provides isolation between VMs, we would like to show that the effect of VM2 is contained, in the sense that it
cannot interfere with the sharing of the page \( p \), nor touch the contents of it. We capture this by showing that the same specification holds for VM0 as in the previous section.

This kind of scenario underpins many use cases of the kind of thin hypervisor we are modelling. For instance, if a secondary VM running some safety-critical service only interacts with the primary VM (running the operating system for scheduling and simple memory sharing), then other VMs cannot manipulate or break the secondary VM through malicious writes to memory.

We leverage the basic memory integrity mechanism of the machine to show robust safety for some key scenarios, that is, safety even in the presence of interactions with arbitrary unknown VMs trying to violate memory isolation, including by making hypercalls to attempt to get access to the private memory of other VMs. There are two overall shapes of scenarios: (1) When the primary VM is safe: because the primary VM is where the scheduler resides, and therefore it interacts with all the secondary VMs (at least for scheduling), strong properties hold for the whole system. (2) When the primary VM is compromised: weaker properties still hold for known secondary VMs.

**Proving robust safety.** Proving robust safety for a machine with only known VMs is straightforward, as the property is captured by VMSL: (1) For each known VM, we prove a weakest precondition. (2) We apply the adequacy theorem, which combines the proved weakestpre of all VMs together, to get a valid global execution of the whole machine. However, this approach does not work directly if an extra unknown VM is considered. To be able to apply the adequacy theorem, we first have to establish a weakestpre for that unknown VM under conditions that are compatible with the resources used for the other VMs. Because we do not have a concrete program, we do not know whether the program will behave properly, or try to maliciously write to a memory cell that exclusively belongs to another VM, or share memory with other VMs via hypercalls, or any combination of these. Therefore, the questions we face are how to obtain a weakestpre for an unknown VM, and whether we can use VMSL to establish one.

Inspired by models for capabilities [16], [17], [18], our answer is that we can do so using logical relations. We define two logical relations that are compatible with each other, one for each of the two scenarios. We introduce the logical relation for the first scenario and illustrate it on the example of Figure 1 in Section 4.1, and describe how the second logical relation is derived by extending the first in Section 4.2.

### 4.1. A logical relation for unknown secondary VMs

To prove examples like Figure 1, we define a unary logical relation \( \mathcal{R} \) whose fundamental theorem gives us a weakest precondition for any unknown secondary VM. Our logical relation states that, given the state of the page table and in-flight transactions that determine which memory pages VM\( i \) has or may get access to, as defined by \( \text{InterpAccess} \), the execution of VM\( i \) can be safely resumed, as defined by \( \text{InterpExecute} \):

\[
\mathcal{R}(i) \overset{\text{def}}{=} \text{InterpAccess}(i) \rightarrow \text{InterpExecute}(i)
\]

Then, the fundamental theorem of the logical relation (FTLR) just states that the logical relation holds for any VMID \( i \) except for 0:

\[
\forall i, i \neq 0 \rightarrow \mathcal{R}(i)
\]

From the perspective of proving the FTLR, \( \text{InterpAccess} \) can be regarded as a predicate specifying the exact resources we need to prove the execution of VM\( i \). We define \( \text{InterpExecute} \) in terms of a weakest precondition for \( \top \), to capture that if the execution of the VM is resumed, with the resources needed to resume it, then we can execute the VM until it stops or suspends again:

\[
\text{InterpExecute}(i) \overset{\text{def}}{=} \text{RCHolds}@i \rightarrow WP\text{ Normal} @ i \{ \top \}
\]

It is sufficient for the postcondition to be \( \top \), because we do not need to know what the state of the unknown VM is at the point of halting (in fact, we would not be able to specify it anyway).

**Defining \( \text{InterpAccess} \).** During the execution, VM\( i \) may execute any valid instruction, and so we cannot make assumptions about the contents of memory of VM\( i \) that would restrict its behaviours. Therefore, we have to reason about all possible cases of its execution in the proof of FTLR (which we do by using the proof rules of VMSL).

The definition of \( \text{InterpAccess} \) for a VM\( i \) follows two principles: (1) It must allow us to characterise the behaviour of VM\( i \) enough to prove our desired safety property, whatever instructions VM\( i \) executes. The way this manifests in the proof is that it must include enough resources for us to be able to apply our proof rules for any instruction. (2) It should not needlessly limit our ability to reason about other VMs. Giving to VM\( i \) resources that VM\( j \) could own means we might not have enough resources to reason about VM\( j \). Therefore, \( \text{InterpAccess}(i) \) should contain just enough resources to reason about VM\( i \). These two principles make \( \text{InterpAccess}(i) \) the footprint of running an arbitrary program on VM\( i \). Figure 8 shows the top level definition of \( \text{InterpAccess} \).

In general, \( \text{InterpAccess}(i) \) is parametrised by \( s_{acc} \), the set of pages that VM\( i \) has access to, and \( \tau \), the map from Word to Transaction representing all in-flight transactions. Intuitively, the behaviour of VM\( i \), in particular its interactions with other VMs, is (and can only be) restricted by information carried by these two variables. For instance, VM\( i \) cannot share a page whose PageID is not in \( s_{acc} \), nor retrieve pages shared with another VM according to \( \tau \). The main goals of \( \text{InterpAccess} \) is therefore to interpret these variables with resources, following the two principles above.

Among all the resources of \( \text{InterpAccess}(i) \), some are exclusively owned by VM\( i \), and some have to be shared between VM\( i \) and other VMs due to the communication allowed by HVCs. The shared part is transferred from the
\( \forall s_{acc}, \tau. \)
\[
Pgti \mapsto acc_{\text{acc}} s_{acc} * PgtOea(s_{oea}) * 
\]
\[
\text{MemPages}(s_{oea} \cup \text{exclusive}_{\text{pages}}(\tau)) * 
\]
\[
Pgt\text{TranP}(\tau) * \ldots 
\]
\[
RC_{1/3}@i \{ \Psi_i \} 
\]
\[
\text{where } \Psi_i \overset{\text{def}}{=} 
\]
\[
\exists \tau'. \tau \sim \tau' \land \text{TranHandles}(\tau') * Pgt\text{TranS}(\tau') * 
\]
\[
\text{MemPages}(\text{shared}_{\text{pages}}(\tau')) * \ldots 
\]
\[
RC_{1/3}@0 \{ \Psi_0 \} 
\]  

Figure 8: The shape of the definition of \( \text{InterpAccess}(i) \). All predicates are implicitly parametrised by \( i \) if \( i \) is mentioned in their definitions.

primary to VM\( i \) upon resumption (via \( \Psi_i \)) and is given back to the primary upon yielding (via \( \Psi_0 \)), using \( RCs. \) \( \Psi_i \) and \( \Psi_0 \) are parametrised by an extra \( \tau' \), to represent new transactions allocated or updated during the suspension of VM\( i \). The connection between \( \tau \) and \( \tau' \) is captured by the relation \( \tau \sim \tau' \), which captures how the transactions in which the VM is the sender or receiver in \( \tau \) cannot be touched by other VMs during its suspension, and therefore remain unchanged in \( \tau' \). This relation allows us to safely replace \( \tau \) with \( \tau' \) in \( \text{InterpAccess} \). We then only work with the single variable \( \tau' \) that includes all ongoing transactions when VM\( i \) is actually executed.

We present this definition by first considering the resources interpreting \( s_{acc} \) and \( \tau' \) as a whole, without distinguishing between exclusively owned and shared, to argue why the unknown VM needs them, and later argue why and how to divide it into owned and shared portions.

Interpreting \( s_{acc} \). The interpretation of \( s_{acc} \) is split as follows: First, \( Pgti \mapsto acc_{\text{acc}} \) on Line 2 states that these pages are accessible to VM\( i \), which is required by all the proof rules (e.g. (3) of \text{SS-mov}). Second, \( PgtOea(s_{oea}) \) on the same line provides page table resources for pages that VM\( i \) owns and has exclusive access to (denoted as \( s_{oea} \)). Those resources are required by the proof rules (e.g. (6) of \text{SS-share}) if VM\( i \) shares those pages. \( PgtOea(s_{oea}) \) is defined as \( *_{p \in s_{oea}} Pgt@p \overset{\text{own}}{\longrightarrow} i * Pgt@p \overset{\text{exc}}{\longrightarrow} \text{True} \) and denoted as \( PgtOE(s_{oea}, i, \text{True}) \).

These two components are exclusively owned by VM\( i \) since no other VMs may require them. Another necessary but shared component regarding pages is the memory of \( s_{acc} \), \( \text{MemPages}(s_{acc}) \), which is required by rules for memory access instructions. We divide \( s_{acc} \) and this predicate in two parts: the part that VM\( i \) has exclusive access to (Line 3), and the remainder that is shared with other VMs (Line 7). The former is captured by \( s_{oea} \) plus pages that are lent to VM\( i \), collected by \( \text{exclusive}_{\text{pages}}(\tau') \), and the latter is collected by \( \text{shared}_{\text{pages}}(\tau') \). We refer readers to the Appendix for definitions of \( s_{oea} \) and the two helper functions.

<table>
<thead>
<tr>
<th>Cases</th>
<th>( \text{tran} )</th>
<th>( \text{trr} )</th>
<th>( PgtOE )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i,j,\text{Share, True} )</td>
<td>( 1/3 + 1/3 )</td>
<td>( 1/2 )</td>
<td>( 1/3 + 2/3 )</td>
</tr>
<tr>
<td>( i,j,\text{Share, False} )</td>
<td>( 1/3 + 2/3 )</td>
<td>( 1 )</td>
<td>( 1/3 + 2/3 )</td>
</tr>
<tr>
<td>( i,j,\text{Donate, False} )</td>
<td>( 3 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( i,j,\text{Share, True} )</td>
<td>( 1/3 + 1/3 )</td>
<td>( 1/2 )</td>
<td>( 1/3 + 2/3 )</td>
</tr>
<tr>
<td>( j,k,\text{Share, False} )</td>
<td>( 1/3 )</td>
<td>( 1 )</td>
<td>( 2/3 )</td>
</tr>
<tr>
<td>( j,k,\text{Donate, False} )</td>
<td>( 1/3 )</td>
<td>( 1 )</td>
<td>( 2/3 )</td>
</tr>
</tbody>
</table>

Figure 9: The resources required by cases of transactions in \( \tau' \) (impossible cases are omitted). \( i \) is the VMID of unknown VM, \( j \) and \( k \) are that of two other VMs. The first column gives the type of transaction. Case “\( i,j,\text{Share, True} \)” means a transaction \( h \mapsto t \) where \( t = (i, j, s, \text{Share, True}) \) for some \( h \) and \( s \). Columns two to four give the required fractions of the three types of resources used for possible HVCs on the transaction. \( 1/3 + 2/3 \) under column \( \text{tran} \) means \( \text{tran}@h \mapsto \tau \) is required in total, with \( 1/3 \) of it owned by the unknown VM, and \( 2/3 \) shared.

Interpreting \( \tau' \). Figure 9 outlines how different cases of transactions in \( \tau' \) are interpreted with resources. In general, three types of resources could be required to allow VM\( i \) to perform memory sharing HVCs: \( \text{tran}@h \mapsto t \) is necessary to refer to transaction \( t \) for any sharing operations; \( \text{tran}@h \mapsto \text{trr} \) is necessary to retrieve transaction bound to handle \( h \); \( PgtOE(s_{oea}) \) is necessary to update the status of shared pages in \( s \). These resources are split into fractions; in particular, we split the points-to for transactions into three fractions that must agree on their values. One fraction in some cases is owned by the unknown VM, \( i \), and at least another one is shared in all cases. The owned and shared fractions are used to interpret \( \tau \) and \( \tau' \) respectively, and unified later by \( \tau \sim \tau' \). The points-to resources for the page table are split and unified in the same way, and the splitting is then lifted to \( PgtOE \). At least two fractions of \( PgtOE \) that interpret \( \tau' \) are shared, which allows us to derive the fact that pages shared by two transactions are disjoint by leveraging the exclusivity of \( PgtOE_{\text{trans}} \) that is derived from that of the underlying page table points-tos.

Now let us zoom in on several representative cases to see why those resources are distributed like this. In case “\( i,j,\text{Share, False} \)” VM\( i \) is the sender, and therefore the owner of the pages \( s \). All fractions of the three resources are required as the sender could Reclaim access, recycling the two transaction points-tos and updating \( PgtOE \) by the proof rule \( \text{tran}@h \mapsto \text{trr} \). In case “\( i,j,\text{Donate, False} \)” all of the resources are shared, as the receiver could Retrieve, which gives it ownership
of the pages in s. In case “j, i, Lend, True”, the VM is the receiver, it does not own page table resources nor the points-to for transaction, as there is no way for it to get ownership of those pages. Therefore, full ownerships of the three resources are not required by the proof rules of Retrieve or Relinquish. However, it owns half of the retrieval points-to so that it can remember the fact that it has retrieved after a suspension. In the last case “j, k, _” (which is also the case of our example), the VM is neither the sender nor the receiver (which is the case of VM2 in our example), only the minimum amount of resources is required (in our example, $\text{Trans}_{\text{h}}(\text{tran}_{i, j, s} 0, 1, \{p\}, \text{Share})$ and $\text{Pgt}_{\text{p}} \xrightarrow{\text{own}} 0 * \text{Pgt}_{\text{p}} \xrightarrow{\text{excl}} \text{False}$).

Resources specified in Figure 9 are distributed in three predicates. $\text{Tran}_{\text{Handles}}(\tau')$ at Line 6 includes the least amount of fractions required by all cases, i.e. $\frac{1}{3}, 0$, and $\frac{2}{3}$, of the three kinds of resources respectively, for each transaction in $\tau'$. Remaining owned and shared fractions are included in $\text{Pgt}_{\text{TtranP}}$ and $\text{Pgt}_{\text{TtranS}}$ respectively. Since we cannot prevent the VM from invoking Retrieve with a free handle referring to no transaction, we further include $\text{Fresh}_{\text{Handles}}(s_h)$ in $\text{Tran}_{\text{Handles}}$ for all free handles $s_h$: $\exists s_h, s_h \cup \text{dom}(\tau') = \text{All}_{\text{Handles}} * \text{Fresh}_{\text{Handles}}(s_h) *$

$\Psi_0 \overset{\text{Def}}{=} \exists \tau. \text{Tran}_{\text{Handles}}(\tau) * \text{Pgt}_{\text{TtranS}}(\tau) *$

$\text{Mem}_{\text{Pages}}(\text{shared}_{\text{pages}}(\tau)) * * * \text{RC}_{\text{i, j, s} \circ \text{acc}} \{\Psi_1\}$

We call such a protocol specified by the two resumption conditions the general protocol of VM. It is general in the sense that it specifies necessary resources to support arbitrary execution of VM, for arbitrary number of resumption times. It is used to reason about unknown VMs. In the case where the primary VM is unknown, we sometimes need an additional mechanism for reasoning about sharing between communicating VMs, see the example considered in Section 4.2.

**Proving the FTLR.** We show that the FTLR holds at every step of the execution. Since the program of the VM is unknown, we have to consider all possible instructions. For each instruction, we apply the corresponding general proof rule of our program logic for that instruction. See the Coq formalisation for the proof.

**Instantiating the FTLR.** We now demonstrate how we use the logical relation to reason about the full motivating example by instantiating the FTLR. Recall that our approach is to (1) show a weakest precondition for each of the three VMs, assuming resources describing the initial state of the machine; and (2) combine them to apply the adequacy theorem, which provides these resources.

The weakest precondition for VM can be proved as for the simplified example. To show the weakest precondition for VM2, we instantiate the FTLR with VMID 2. We then have to pick proper $s_{\text{acc}}$ and $\tau$ such that the required resources are disjoint and consistent with resources required by other two known VMs. That is, all initial resources are exclusively owned by one VM, and the protocols specified in resumption conditions agree with each other. We let $\tau$ be 0, since at the beginning there are no transactions, and we let $s_{\text{acc}} = \{p_{\text{v22}}, p_{\text{v2}; p_{\text{p2}}}; p_{\text{v2}}\}$. To show the weakest precondition for VM0, which now runs VM2 before VM1, we have to show the $\text{RC}$ of VM2 specified in $\text{InterpAccess}(2)$. In particular, we let $\tau' = [h \rightarrow (0, 1, \{p\}, \text{Share}, \text{False})]$, whose interpretation in $\text{Tran}_{\text{Handles}}$ will disallow any malicious HVC, such as an Retrieve access to $p$, by VM2. The same resources are included in $\Psi_0$ and given back, so this transfer does not affect the reasoning about the known VMs after running VM2.

**Capturing safety.** The fact that we are able to prove (using our logical relation) that VM0 and VM1 can safely share a page, even though VM2 runs in between and gets the opportunity to try to interfere, shows that our underlying machine-with-HVCs model is secure, in the sense that executing those HVCs will not break isolation unintentionally.

### 4.2. A logical relation for unknown primary VMs

We have shown how to reason in the presence of unknown secondary VMs using our first logical relation. However, secondary VMs also get some guarantees when the primary VM is unknown (and possibly compromised).
For example, consider the scenario in Figure 10: only two secondary VMs, VM1 and VM3, are known, and a page $p$ with $42$ stored in it is shared between them. We would like to show that VM3 can read that same value from the page, even with the unknown primary VM0 in addition to the unknown secondary VM2. In this example, as before, we can instantiate the FTLR to get a weakest precondition for VM2 representing its execution, but we cannot do the same for VM0.

To deal with scenarios with an unknown primary VM, we develop a second logical relation, whose FTLR gives a weakest precondition for the primary VM. We ensure that this second logical relation is also compatible with our previous logical relation. This enables us to show safety of scenarios with both arbitrary unknown primary and secondary VMs, including the example above. In such scenarios, programs of known secondaries have to be written defensively, as they may be scheduled at any point. In this section, we show how we design and use this second logical relation, and refer the reader to the Coq formalisation for the full definition.

The statement of the FTLR of the new logical relation is symmetrical to the previous one: we now require $i$ to be $0$. As before, $\text{InterpExecute}$ is defined as just $\text{WP 0 @ Normal} \{\top\}$, and moreover $\text{RCHolds}$ is not needed as we always run the primary first. The difference is in $\text{InterpAccess}$, which generalises the former to support running arbitrary secondary VMs, which is the extra power of the primary VM. From the perspective of resources, the new $\text{InterpAccess}$ includes (1) resources that supports VM0’s execution except for running other VMs, which is identical to what is required by a secondary VM as in Section 4.1, and (2) resources required by resumption conditions of all secondary VMs to support running other VMs, which is basically all resumption conditions plus the union of resources required by them.

The crux of defining the new $\text{InterpAccess}$ is specifying all the resumption conditions, i.e., protocols between secondaries and the primary. For unknown secondaries, as shown in the previous section, we can use the general protocol. For known secondaries, because we want our FTLR to be generic in their code, the protocol cannot depend on their code (so, here, we cannot take the approach we used for Figure 1). However, we cannot use the general protocol for known code either, as it is too general, and thus cannot be used to prove e.g. the example in Figure 10. The technical problem arises from: (1) the very loose assumption on the content of memory, which is quantified over existentially in the general protocol. That is, we want to show the content in $p$ is a specific number, but the general protocol only gives us that there is some number in $p$. (2) the fact that resumption conditions only allow transferring resources along the scheduling control flow via the primary VM (as illustrated on the left of Figure 11). With the cooperative scheduling mechanism we model, secondary VMs can only yield to the primary VM, not directly from one secondary VM to another. This means that in this example, the shared page $p$ can only be transferred between VM1 and VM3 with VM0 as a man in the middle.

Our approach. Instead, we exclude the page $p$ from the general protocol, and share it between VM1 and VM3 in another way (which we can do since $p$ is not accessible to VM0). To do this, we use invariants as a complementary resource sharing mechanism, for resources that cannot or should not be shared via the general protocol. In this example, assuming $p$’s value is always $42$ after VM1 writes to it, we can establish a trivial invariant, as illustrated in Figure 11 with the memory points-tos of page $p$.

How we implement our approach. Recall that the general protocol specifies the resources a secondary shares with all other VMs, although they are only ever transferred via the primary. It indicates that it is safe to run an unknown primary without resources that secondaries shared with other secondaries in the general protocol. We therefore can divide the resources of the general protocol into slices, one for each pair of VMIDs, which only contain one-to-one shared resources. This way, we can now safely remove secondary-to-secondary slices from the general protocol between a secondary and the primary. We then parametrise the logical relation by the secondary-to-secondary slices, thereby allowing the user of the FTLR to decide which of those slices are (partially) transferred via the unknown primary. For instance, resources that VM1 shares using its general protocol are divided into three slices containing resources that it shares with (1) VM0; (2) VM2; and (3) VM3. We say the slice from VM1 to VM2 is full if it contains all related resources required by the general protocol between VM1 and the primary. We then instantiate the FTLR with full slices (1) and (2), and (3) minus the memory of page $p$, to exclude that page from the VM1-to-VM3 slice. By doing so, yielding of VM1 will not require the resources for page $p$, and therefore we can use it to establish the invariant. Moreover, by letting slices from VM2 to other VMs be full, we can actually recover the general protocol of VM2, therefore making the two logical relations compatible.

5. Related work

Hypervisor verification. There are several lines of work on hypervisor verification, including Hyper-V [19], SeKVM [20, 21, 22]. Our work is complementary.

Microsoft’s Hyper-V is an industrial hypervisor partially verified with VCC verification suite [23] which aims for verification of low-level concurrent C code. Most of the verification effort was put on the correctness of hypervisor implementation, but not on validating the top-level specification and investigating security properties.

The main focus of SeKVM is on hypervisor verification. As part of it, they capture generic isolation properties.
between virtual machines and their hypervisor (based on KVM) in the form of non-interference results about their combined model of the machine and the hypervisor, capturing both integrity and secrecy. They support memory sharing in a much more restrictive way, only allowing a VM to share encrypted data with the less privileged portion of the hypervisor to support I/O virtualization. Our approach factors the integrity (but not the secrecy) part of their result into a logic to reason about concrete programs, and a logical relation that captures isolation. This enables us to verify individual concrete scenarios, whereas, in our terms, their result is concerned with composing exclusively unknown VMs.

All of the work above makes drastic simplifying assumptions, as the actual behaviour of page tables, especially in the presence of concurrency, is only starting to be understood [8].

Work has been done towards hypervisor verification against authoritative models. Nienhuis et al. [1] and Bauereiss et al. [24] prove security properties above full-scale, authoritative, formal ISA models of the CHERI and Morello capability architectures. These properties are fine-grained because of the capabilities, but weaker in that they are architectural invariants, and thus cannot rely on known code. Sammler et al. [25] develop a separation logic above authoritative, formal ISA models of Arm-A and RISC-V by specialising the ISA definition to partially concrete opcodes through (unverified) symbolic evaluation [26]. They focus on verifying local specifications of known code, including (the sequential aspects of) some exception handlers.

**Reasoning about low-level code.** Variants of separation logic have been used before for reasoning about assembly code [27], [?] and in these works, the logics have proof rules corresponding to sequential composition of individual instructions. Since we are basing our logic on Iris, our variant of separation logic is more expressive, e.g., it supports invariants. Georges et al. [28] reason about assembly level capability machine code in a logic, which like ours, is built on top of Iris. We believe that our approach using single-step weakest precondition could also simplify reasoning about the capability machine of Georges et al.

**Capability machines.** Capabilities [29], [30], [31], [32] are an alternative hardware mechanism for access control, in the form of dynamically checked unforgeable tokens of authority, typically granting some type of access to a portion of memory. Capabilities give finer-grained access to blocks of memory (not being restricted to page size), and are self-managed, in the sense that they do not depend on a hypervisor, merely on the underlying hardware mechanism. Proofs of safety for capability machines have also used unary, untyped logical relations, e.g. [33], [34], [35]. However, these logical relations are quite different from ours, because of the different underlying mechanisms.

**Validating hypervisor ABI.** The HASPOC project [36], [37] aims for a secure virtualisation platform for ARMv8 for which they prove information-flow security. They introduce an idealised model in which information-flow security holds by construction, and prove a bisimuation between it and the platform model. Their approach is similar to ours, in that only execution of VMs is modelled, and the hypervisor is invisible. However, in their model, each VM’s memory is isolated and cannot be shared; their inter-VM communication mechanism is more like messaging.

## 6. Conclusion

We have formalised a substantial fragment of Arm’s FF-A ABI as an operational semantics in which HVCs are primitive steps and we have demonstrated that the model is secure, in the sense that VMs running unknown and possibly malicious code cannot break isolation unintentionally. In more detail, we have developed VMSL, a novel separation logic for modular reasoning about known VMs communicating above FF-A. In particular, VMSL supports ‘VM-local’ reasoning via its notion of resumption conditions. Moreover, we have shown how to use the logic to develop logical relations that capture the intended isolation guarantees and which can be used to formally prove robust safety for communicating known VMs that interact with VMs running unknown code.

Future work includes extending our model with concurrency and non-cooperative scheduling. We are also interested in adapting our model to the pKVM [38], [39], [40] ABI, which is different from the FF-A ABI but similar in spirit. It would also be interesting to show that an implementation of a hypervisor is a formal refinement of (a more detailed version of) our model.

## References


Appendix

1. Machine model

We first show our minimalistic, simplified subset of the Arm-A instruction set. pc is the program counter, and nz is the negative-zero flags register set by cmp and used by branches; these are system registers that cannot be accessed directly by VMs.

\[ r \in GPReg ::= R^k \ (k \in \{0..31\}) \quad R \in Reg ::= pc \ | \ nz \ | \ r \quad n \in \mathbb{N} \quad a \in \mathbb{N} \quad Arg ::= r \ | \ n \]

\[ Instr ::= \text{nop} \ | \ \text{mov} \ r \ a \ | \ \text{ldr} \ r_{dst} \ r_{addr} \ | \ \text{str} \ r_{val} \ r_{dst} \ | \ \text{cmp} \ r \ a \ | \ \text{bne} \ r \ | \ \text{br} \ r \ | \ \text{add} \ r_{dst} \ r_{arg} \ | \ \text{sub} \ r_{dst} \ r_{arg} \ | \ \text{mult} \ r_{dst} \ r_{arg} \ | \ \text{halt} \ | \ \text{hvc} \]

The configuration of the operational semantics is the following:

\[ \text{Configuration} \overset{\text{def}}{=} \text{ExecMode} \times \text{State} \]

\[ \text{ExecMode} \overset{\text{def}}{=} \text{Normal} \ | \ \text{Halted} \ | \ \text{Failed} \ | \ \text{PageFault} \]

\[ \text{State} \overset{\text{def}}{=} \{ \text{mem} : \text{Memory} ; \ \text{pqt} : \text{PageTable} ; \ \text{regs} : \text{RegisterFiles} ; \ \text{curr} : \text{VMID} ; \ \text{trans} : \text{Transactions} ; \ \text{mb} : \text{Mailboxes} ; \} \]

\[ \text{Memory} \overset{\text{def}}{=} \text{Word} \rightarrow \text{Word} \]

\[ \text{PageTable} \overset{\text{def}}{=} \text{PageID} ightarrow \text{PageStatus} \]

\[ \text{PageID} \overset{\text{def}}{=} \text{Word} \ (\text{page-aligned}) \]

\[ \text{PageStatus} \overset{\text{def}}{=} \{ \text{owned} : \text{option VMID} ; \ \text{exclusive} : \mathbb{B} ; \ \text{accessible} : \mathcal{P}_{\text{fin}}(\text{VMID}) ; \} \]

\[ \text{RegisterFiles} \overset{\text{def}}{=} \text{VMID} \rightarrow \text{RegisterFile} \]

\[ \text{RegisterFile} \overset{\text{def}}{=} \text{Reg} \rightarrow \text{Word} \]

\[ \text{Reg} \overset{\text{def}}{=} \text{pc} \ | \ \text{nz} \ | \ \text{R}^k \ (k \in \{0..31\}) \]

\[ \text{CurrentVM} \overset{\text{def}}{=} \text{VMID} \]

\[ \text{Transactions} \overset{\text{def}}{=} \text{Word} \rightarrow \text{option Transaction} \]

\[ \text{Transaction} \overset{\text{def}}{=} \text{MetaData} \times \mathbb{B} \]

\[ \text{MetaData} \overset{\text{def}}{=} \{ \text{sndr} : \text{VMID} ; \ \text{rcvr} : \text{VMID} ; \ \text{pgs} : \mathcal{P}_{\text{fin}}(\text{PageID}) ; \ \text{type} : \text{TransactionType} ; \} \]

\[ \text{TransactionType} \overset{\text{def}}{=} \text{Share} \ | \ \text{Donate} \ | \ \text{Lend} \]

\[ \text{MailBoxes} \overset{\text{def}}{=} \text{VMID} \rightarrow \text{Mailbox} \]

\[ \text{Mailbox} \overset{\text{def}}{=} \text{TXBuffer} \times \text{RXBuffer} \]

\[ \text{TXBuffer} \overset{\text{def}}{=} \text{PageID} \]

\[ \text{RXBuffer} \overset{\text{def}}{=} \text{PageID} \times \text{option} (\text{Word} \times \text{VMID}) \]

2. Program logic

In Figure [12], we present the essential rules of RC and SSWP and the definitions of two weakest preconditions and resumption condition from which the rules are derived. The definitions extensively use Iris primitives that are not introduced in the paper. Next, a selection of the proof rules needed to prove the example in Section [3] are presented. In all these rules, we omit a side condition saying the instruction is not in the write-only TX page.

3. Logical relations

We present the full definition of \text{InterpAccess} of the logical relation for secondary VMs at the end.
∀ (∅ E) \[ \Rightarrow \], \exists i \Psi \Rightarrow \{ \sigma \} \{(m \in \{RCHolds\}) \land \forall \Phi \text{RC-RCAuth} \Rightarrow \{ \Phi \} \land \forall \Phi \text{RC-RCAuth} \Rightarrow \{ \Phi \}

WP-SSWP

WP \( m \oplus i \{ \Phi \} \vdash\text{SSWP}\ m \oplus i \{ (b, \emptyset) \land (b \land RCHolds \oplus i) \lor \neg b \} \rightarrow WP \_m' \oplus i \{ \Phi \} \}

(a) Rules for RC and SSWP

\[
\begin{align*}
\text{RC-AGREE} & : \text{RCAuth} \oplus i \Phi \land \text{RC}_q \oplus i \{ \Psi \} \\
\text{RC-SPLIT} & : \text{RC} \oplus i \{ \Phi \} \vdash \text{RC}_{1/2} \oplus i \{ \Phi \} \land \text{RC}_{1/2} \oplus i \{ \Phi \}
\end{align*}
\]

\[
\begin{align*}
\text{RC-HOLD} & : \text{RCHolds} \oplus i \land \text{RC}_{1/2} \oplus i \{ \Psi \} \vdash \Psi \land \text{RC} \oplus i \{ \Psi \} \\
\text{RC-UPDATE} & : \text{RCAuth} \oplus i \Phi \land \text{RC}_q \oplus i \{ \Psi \} \rightarrow \text{RCAuth} \oplus i \Phi' \land \text{RC}_q \oplus i \{ \Phi' \}
\end{align*}
\]

(b) Resumption conditions

\[
\begin{align*}
WP \_m \oplus i \{ \Phi \} \& \{ \text{terminated}(m) \land \Rightarrow \_E \Phi(m) \} \lor \\
\{ \text{terminated}(m) \land \forall n, \sigma. \text{scheduled}(\sigma, i) \rightarrow \text{stateInterp}(n, \sigma) \equiv 0 \land \text{reducible}(m, \sigma) \} \land \\
\forall m', \sigma'. (\exists \Psi. \text{RCAuth} \oplus i \Psi) \rightarrow \text{primStep}(m, \sigma, m', \sigma') \equiv 0 \\
\Rightarrow 0 \Rightarrow E (\exists \Psi. \text{RCAuth} \oplus i \Psi) \land \left( \begin{array}{c}
\forall i' \in \text{justScheduled}(n, \sigma, \sigma') \text{RC}_{1/2} \oplus i' \land \text{stateInterp}(n, \sigma') \land \\
\text{scheduled}(\sigma', i) \lor \text{terminated}(m') \lor \neg \text{scheduled}(\sigma', i) \land \neg \text{terminated}(m') \land \text{RCHolds}_{1/2} \oplus i) \rightarrow WP \_m' \oplus i \{ \Phi \}
\end{array} \right)
\end{align*}
\]

(c) weakest precondition

\[
\begin{align*}
SSWP \_m \oplus i \{ \Phi \} \& \{ \text{terminated}(m) \land \Rightarrow \_E \Phi(\text{False}, m) \} \lor \\
\{ \neg \text{terminated}(m) \land \forall n, \sigma. \text{scheduled}(\sigma, i) \rightarrow \text{stateInterp}(n, \sigma) \equiv 0 \land \text{reducible}(m, \sigma) \} \land \\
\forall m', \sigma'. (\exists \Psi. \text{RCAuth} \oplus i \Psi) \rightarrow \text{primStep}(m, \sigma, m', \sigma') \equiv 0 \\
\Rightarrow 0 \Rightarrow E (\exists \Psi. \text{RCAuth} \oplus i \Psi) \land \left( \begin{array}{c}
\forall i' \in \text{justScheduled}(n, \sigma, \sigma') \text{RC}_{1/2} \oplus i' \land \text{stateInterp}(n, \sigma') \land \\
\Phi(\neg \text{scheduled}(\sigma', i) \land \neg \text{terminated}(m')) \land m')
\end{array} \right)
\end{align*}
\]

(d) single-step weakest precondition

Figure 12: Selected definitions and rules of VMSL
SS-send-prim
\[ i \neq 0 \land \text{IsInstr} @ i (s, a, hvc) \Rightarrow \text{R0} @ i \xrightarrow{\text{reg}} \text{encode} (\text{Send}) \Rightarrow \text{R1} @ 0 \xrightarrow{\text{reg}} i \Rightarrow \text{RC} @ 0 \xrightarrow{\text{reg}} l * \]
\[ \text{Mb} @ 0 \xrightarrow{\text{tx}} \text{ptx} \Rightarrow \text{MemPage} (\text{ptx}, \text{memtx}) \Rightarrow \text{Mb} @ i \xrightarrow{\text{tx}} \text{prx} \Rightarrow \text{RXState} @ i \Rightarrow \text{None} \Rightarrow \text{MemPage} (\text{prx}, \text{memrx}) \]
\[ \text{SSWP Normal } @ 0 \begin{cases} (\text{False, Normal}) \end{cases} \]

SS-poll
\[ \text{IsInstr} @ i (s, a, hvc) \Rightarrow \text{R0} @ i \xrightarrow{\text{reg}} \text{encode} (\text{Poll}) \Rightarrow \text{R1} @ i \xrightarrow{\text{reg}} s \Rightarrow \text{R2} @ i \xrightarrow{\text{reg}} - \Rightarrow \text{RXState} @ i \Rightarrow \text{Some} (l, 0) \Rightarrow \text{msg} \Rightarrow \text{msg} \subseteq \text{memtx} \wedge \text{dom} (\text{msg}) = \{ \text{ptx} + 0; \ldots ; \text{ptx} + l - 1 \} \wedge \text{MemPage} (\text{prx}, \text{zip} ([\text{prx} + 0; \ldots ; \text{prx} + l - 1], \text{cod} (\text{msg})) \cup \text{memrx}) \]
\[ \text{SSWP Normal } @ 0 \begin{cases} (\text{False, Normal}) \end{cases} \]

SS-halt
\[ \text{IsInstr} @ i (s, a, \text{halt}) \Rightarrow \text{SSWP Normal } @ i \begin{cases} (\text{False, Halted}) \end{cases} \]

SS-retrieve-share
\[ \text{IsInstr} @ i (s, a, hvc) \Rightarrow \text{R0} @ i \xrightarrow{\text{reg}} \text{encode} (\text{Retrieve}) \Rightarrow \text{R1} @ i \xrightarrow{\text{reg}} s \Rightarrow \text{R2} @ i \xrightarrow{\text{reg}} l \Rightarrow \text{RXState} @ i \Rightarrow \text{None} \Rightarrow \text{MemPage} (\text{prx}, \text{memrx}) \]
\[ \text{SSWP Normal } @ i \begin{cases} (\text{False, Normal}) \end{cases} \]

SS-yield
\[ i \neq 0 \land \text{IsInstr} @ i (s, a, hvc) \Rightarrow \text{R0} @ i \xrightarrow{\text{reg}} \text{encode} (\text{Yield}) \Rightarrow \text{R0} @ 0 \xrightarrow{\text{reg}} - \Rightarrow \text{R1} @ 0 \xrightarrow{\text{reg}} - \Rightarrow \text{RC} @ 1/2 @ 0 \{ \Psi_0 \} \Rightarrow \text{RC} @ 1 @ i \{ - \} \Rightarrow \Phi_\text{other} \Rightarrow \text{SSWP Normal } @ i \begin{cases} (\text{True, Normal}) \end{cases} \]

SS-ldr
\[ a_{\text{addr}} \neq p, \text{ptx} \land \text{IsInstr} @ i (s, a, \text{ldr} r_{\text{dst}} r_{\text{addr}}) \Rightarrow r_{\text{dst}} @ i \xrightarrow{\text{reg}} s \Rightarrow r_{\text{addr}} @ i \xrightarrow{\text{reg}} w \Rightarrow a_{\text{addr}} \Rightarrow \text{Mb} @ i \xrightarrow{\text{tx}} \text{ptx} \Rightarrow a_{\text{addr}} \Rightarrow \text{mem} \Rightarrow w \]
\[ \text{SSWP Normal } @ 0 \begin{cases} (\text{False, Normal}) \end{cases} \]

SS-try
\[ \text{IsInstr} @ i (s, a, hvc) \Rightarrow \text{R0} @ i \xrightarrow{\text{reg}} \text{encode} (\text{Try}) \Rightarrow \text{R1} @ i \xrightarrow{\text{reg}} s \Rightarrow \text{R2} @ i \xrightarrow{\text{reg}} j \Rightarrow \text{RXState} @ i \Rightarrow \text{None} \Rightarrow \text{MemPage} (\text{prx}, \text{memrx}) \]
\[ \text{SSWP Normal } @ i \begin{cases} (\text{False, Normal}) \end{cases} \]

SS-tryas
\[ \text{IsInstr} @ i (s, a, hvc) \Rightarrow \text{R0} @ i \xrightarrow{\text{reg}} \text{encode} (\text{Try}) \Rightarrow \text{R1} @ i \xrightarrow{\text{reg}} s \Rightarrow \text{R2} @ i \xrightarrow{\text{reg}} j \Rightarrow \text{RXState} @ i \Rightarrow \text{None} \Rightarrow \text{MemPage} (\text{prx}, \text{memrx}) \]
\[ \text{SSWP Normal } @ i \begin{cases} (\text{False, Normal}) \end{cases} \]
\(\text{InterpAccess}(i) \triangleq \forall s_{\text{acc}}, \tau. \exists \omega. \text{total}(\omega) \land \ast \, r \mapsto i \xrightarrow{\text{reg}} w \ast \exists p_{tx}, p_{rx}. \text{TXPage}(p_{tx}) \ast \text{Mb} \mapsto i \xrightarrow{\text{rx}} p_{rx} \ast \)

\(\text{Pgt}@i \xrightarrow{\text{acc}} s_{\text{acc}} \ast \text{PgtOea}(s_{\text{o ea})} \ast \text{MemPages}(s_{\text{o ea}} \cup \text{exclusive\_pages}(\tau)) \ast \text{PgtTranP}(\tau) \ast \text{ValidAccess}(s_{\text{acc}}, \tau) \ast \text{RC}_{i,j}@i \{\Psi_i(\tau)\}
\)

where \(s_{\text{acc}} \triangleq s_{\text{acc}} \setminus \{p_{tx}, p_{rx}\} \setminus \text{accessible\_pages}(\tau); \)

\(\text{accessible\_pages}(\tau) \triangleq \text{pages}(\text{filter}(\langle \lambda t. t.\, \text{sndr} = i \land t.\, \text{type} = \text{Share} \lor t.\, \text{rcvr} = i \land t.\, \text{retri} = \text{True}, \rangle, \tau)); \)

\(\text{exclusive\_pages}(\tau) \triangleq \text{pages}(\text{filter}(\langle \lambda t. t.\, \text{rcvr} = i \land \neg (t.\, \text{type} = \text{Lend} \land t.\, \text{retri} = \text{True}), \rangle, \tau)); \)

\(\text{PgtTranP}(\tau) \triangleq \left(\begin{array}{c}
\text{Tran}@h \xrightarrow{\text{trn}} \tau_{i,j} \text{t.meta} \ast \\
\text{Tran}@h \xrightarrow{\text{rtv}} \tau_{i,j} \text{t.retri} \ast 
\end{array}\right) \ast \left(\begin{array}{c}
\text{Tran}@h \xrightarrow{\text{trn}} \tau_{i,j} \text{t.meta} \ast \\
\text{Tran}@h \xrightarrow{\text{rtv}} \tau_{i,j} \text{t.retri} \ast 
\end{array}\right); \)

\(\text{ValidAccess}(s_{\text{acc}}, \tau) \triangleq \{p_{tx}, p_{rx}\} \in s_{\text{acc}} \land \text{accessible\_pages}(\tau) \subseteq s_{\text{acc}} \setminus \{p_{tx}, p_{rx}\}; \)

\(\Psi_i(\tau) \triangleq \exists \tau', \pi, \tau_{\text{only}}. \tau \sim \tau' \land \tau_{\text{only}} = \text{only}(\tau') \land \text{TranHandles}(\tau') \ast \text{PgtTranS}(\tau_{\text{only}}) \ast \text{MemPages}(\text{shared\_pages}(\tau_{\text{only}})) \ast \text{R0}@0 \xrightarrow{\text{reg}} \text{encode}(\text{Run}) \ast \text{R1}@0 \xrightarrow{\text{reg}} i \ast \text{R2}@0 \xrightarrow{\text{reg}} - \ast \)

\(\text{AllRXPages}(\pi) \ast \text{RC}_{i,j}@0 \{\Psi_0(\tau', \pi)\}; \)

\(\text{only}(\tau) \triangleq \text{filter}(\langle \lambda t. t.\, \text{sndr} = i \lor t.\, \text{rcvr} = i, \rangle, \tau); \)

\(\tau \sim \tau' \triangleq (\text{map}(\langle \lambda t. t.1, \rangle, (\text{filter}(\langle \lambda t. t.\, \text{sndr} = i \land t.\, \text{type} = \text{Donate}, \rangle), \tau))); = (\text{map}(\langle \lambda t. t.1, \rangle, (\text{filter}(\langle \lambda t. t.\, \text{rcvr} = i \land t.\, \text{retri} = \text{True}, \rangle), \tau))); = (\text{filter}(\langle \lambda t. t.\, \text{rcvr} = i \land t.\, \text{retri} = \text{True}, \rangle), \tau'); \)

\(\text{TranHandles}(\tau') \triangleq \exists s_h, s_h \cup \text{dom}(\tau') = \text{AllHandles} \ast \text{FreshHandles}(s_h) \ast \)

\(\left(\begin{array}{c}
\text{Tran}@h \xrightarrow{\text{trn}} \tau_{i,j} \text{t.meta} \ast \\
\text{Tran}@h \xrightarrow{\text{rtv}} \tau_{i,j} \text{t.retri} \ast 
\end{array}\right) ; \)

\(\text{PgtTranS}(\tau) \triangleq \left(\begin{array}{c}
\text{Tran}@h \xrightarrow{\text{trn}} \tau_{i,j} \text{t.meta} \ast \\
\text{Tran}@h \xrightarrow{\text{rtv}} \tau_{i,j} \text{t.retri} \ast 
\end{array}\right) \ast \left(\begin{array}{c}
\text{Tran}@h \xrightarrow{\text{trn}} \tau_{i,j} \text{t.meta} \ast \\
\text{Tran}@h \xrightarrow{\text{rtv}} \tau_{i,j} \text{t.retri} \ast 
\end{array}\right); \)

\(\text{shared\_pages}(\tau) \triangleq \text{pages}(\text{filter}(\langle \lambda t. \neg (t.\, \text{type} = \text{Lend} \land t.\, \text{retri} = \text{True}), \rangle, \tau)); \)

\(\text{AllRXPages}(\pi) \triangleq \text{total}(\pi) \land \text{MemPage}(p_{rx}) \ast (\forall \pi_j, \pi_i = \pi_j \rightarrow \text{RXState}@i \rightarrow \pi_i) \ast \text{RXPages}(\text{delete}(\pi, i)); \)

\(\text{RXPages}(\pi) \triangleq \left(\begin{array}{c}
\pi_j = \text{None} \rightarrow \text{RXState}@j \rightarrow \text{None} \ast \exists p_{rxj}. \text{Mb} \mapsto j \xrightarrow{\text{rx}} p_{rxj} \ast \text{MemPage}(p_{rxj}) \lor \\
(\pi_j = \text{Some}_\langle \rangle \rightarrow \text{RXState}@j \rightarrow \pi_j)
\end{array}\right); \)

\(\Psi_0(\tau', \pi) \triangleq \exists \tau''', \pi_{\text{ret}}, \tau_{\text{ret}}. \tau_{\text{ret}} = \text{only}(\tau''') \cup \text{except}(\tau') \lor \text{only}(\tau''') \land \text{except}(\tau') \lor \forall j, i \neq i \rightarrow \tau' \sim \tau_{\text{ret}} \land \text{TranHandles}(\tau_{\text{ret}}) \ast \text{PgtTranS}(\tau_{\text{ret}}) \ast \text{MemPages}(\text{shared\_pages}(\tau_{\text{ret}})) \ast \)

\(\text{RXState}@i \mapsto \pi_i \ast \exists p_{rxj}. \text{Mb} \mapsto i \xrightarrow{\text{rx}} p_{rx} \ast \text{MemPage}(p_{rx}) \ast \text{RetRXReg}(\pi_i', \pi) \ast \text{RC}_{i,j}@i \{\Psi_i(\tau_{\text{ret}})\}; \)

\(\text{except}(\tau) \triangleq \text{filter}(\langle \lambda t. \neg (t.\, \text{sndr} = i \lor t.\, \text{rcvr} = i), \rangle, \tau); \)

\(\text{RetRXReg}(\pi_i', \pi) \triangleq \left(\begin{array}{c}
(\text{R0}@0 \xrightarrow{\text{reg}} \text{encode}(\text{Yield}) \lor \text{R0}@0 \xrightarrow{\text{reg}} \text{encode}(\text{Wait}) \lor \pi_i' = \text{None} \ast \\
\text{RXPages}(\text{delete}(\pi, i)) \ast \text{R1}@0 \xrightarrow{\text{reg}} i \ast \text{R2}@0 \xrightarrow{\text{reg}} - \lor \\
(\text{R0}@0 \xrightarrow{\text{reg}} \text{encode}(\text{Send}) \ast j \mapsto l, p_{rxj}, \text{RXState}@j \mapsto i \rightarrow j \rightarrow \pi_i, \ast \text{Mb} \mapsto j \xrightarrow{\text{rx}} p_{rxj} \ast \text{MemPage}(p_{rxj}) \ast \\
\text{RXPages}(\text{delete}(\pi, i))[j \mapsto \text{Some}(l, i)] \ast \pi[j] = \text{None} \ast \text{R1}@0 \xrightarrow{\text{reg}} j \ast \text{R2}@0 \xrightarrow{\text{reg}} l); \right) \)