VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A

ZONGYUAN LIU, SERGEI STEPANENKO, JEAN PICHON-PHARABOD, AMIN TIMANY, ASLAN ASKAROV, and LARS BIRKEDAL, Aarhus University, Denmark

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 hypercall ABIs 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 and security 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 VMSL, a novel 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, the guarantee that even if some of the virtual machines are compromised and execute unknown code, 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.

CCS Concepts: • **Theory of computation → Separation logic**; *Program verification*; • **Security and privacy → Virtualization and security**; *Logic and verification.*

Additional Key Words and Phrases: hypercall, FF-A, robust safety, separation logic, logical relation, Iris

ACM Reference Format:
Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI, Article 165 (June 2023), 25 pages. https://doi.org/10.1145/3591279

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 [Nienhuis et al. 2020, §VI], this is not always the case [Chidambaram 2018]. 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

Authors' address: Zongyuan Liu, zy.liu@cs.au.dk; Sergei Stepanenko, sergei.stepanenko@cs.au.dk; Jean Pichon-Pharabod, jean.pichon@cs.au.dk; Amin Timany, timany@cs.au.dk; Aslan Askarov, aslan@cs.au.dk; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Aarhus, Denmark.

2475-1421/2023/6-ART165
https://doi.org/10.1145/3591279

are 'live' [Appel et al. 2017], 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) ABIs of FF-A, the Arm Firmware Framework for Arm A-profile [Arm Ltd. 2022], as implemented by Google's Hafnium hypervisor [Hafnium development team 2022]. The hypercall ABIs allow 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 ABIs compared to the informal FF-A specifications, but still captures the essence (see Section 2.1 for details). We then validate it by exercising it to verify key scenarios of VMs using the ABIs 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 more 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 those 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 ABIs and their 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. VMSL effectively reduces the problem of verifying VMs communicating via the hypercall ABIs of FF-A to well-studied problems: cooperative multitasking, and functional correctness of assembly.

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. This requires carefully designed ABIs, 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 [Arm Ltd. 2022]. 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 specifications enforce robust safety.

We highlight the following features of our VMSL logic:

- VMSL is foundational [Appel 2001]: we mechanise the definition of VMSL and prove it sound in Coq using the Iris separation logic framework [Jung et al. 2018] and the Iris Proof Mode [Krebbers et al. 2017]. Both the definition of VMSL and the examples using VMSL extensively rely on the expressive power of Iris to make reasoning about low-level code tractable, and we point out where we utilise it throughout the paper.
- 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 logical resource sharing mechanisms to support reasoning about communication between VMs: (1) standard separation logic invariants, and (2) our *resumption conditions*, a logical sharing mechanism that offers more convenience than standard invariants for communication between VMs in the style of cooperative multitasking.

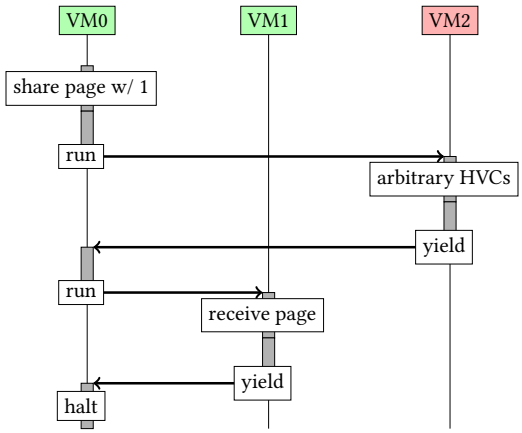

Fig. 1. A motivating example where a compromised VM2 is contained: the page sharing between VM0 and VM1 is guaranteed to succeed if the adversarial VM2 yields, no matter what other HVCs VM2 makes. The memory integrity of the page is guaranteed.

- VMSL is factored in 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 ABIs of FF-A.
- 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 ABIs, as implemented by Hafnium, in the form of 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 Iris. The Coq formalisation and the instructions of usage are available in the supplementary material [Liu et al. 2023].

Non-goals. We focus on exercising the HVC ABIs, 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 ABIs, 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 does not include interrupts, and assumes that there is no concurrency, as characterising the semantics of virtual memory in a concurrent setting is work in progress [Simner et al. 2022].

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 their memory. For instance,

adversaries could invoke arbitrary HVCs to try to interfere with 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 ABIS

As we focus on the HVC ABIs, 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.

2.1 Scope

We specify the hardware behaviours of virtualisation, including page table lookup and context switching, plus the following HVCs of FF-A: (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. This covers most of FF-A, apart of the 'secure world' trusted computing functionality involving TrustZone. We omit 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.

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 permissions. (2) We only model 1-to-1 sharing (as implemented by Hafnium) instead of 1-to-n, and accordingly simplify the format of transaction descriptors. These, along with other minor simplifications, help keep the size of our model manageable, but do not significantly omit specification details or impact expressivity. For instance, we believe the model can be adapted to support 1-to-n sharing.

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

2.2 Formalising HVCs

Informally, a hypervisor provides the illusion to VMs that they are running on a machine in which the whole HVC 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, and we explain it below.

2.2.1 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, the set of *VMID*s of the VMs that have access to the page, and a bit indicating whether it is exclusively owned (can only be accessed) by one VM. For instance, the status of page p in the example of

```
1  /* VM0 */
2  /* save x to p */
3  mov R5 <- #p
4  str R0 [R5]
5  /* prepare desc */
6  mov R5 <- #ptx
7  mov R4 <- 0
8  str R4 [R5]
9  ...
10 /* share p */
11 mov R0 <- #Share
12 mov R1 <- 4
13 hvc
14 /* send handle */
15 mov R5 #ptx
16 str R2 [R5]
17 mov R3 <- R2
18 mov R0 <- #Send
19 mov R1 <- 1
20 mov R2 <- 1
21 hvc
22 /* run VM2 */
23 mov R0 <- #Run
24 mov R1 <- 2
25 hvc
26 /* run VM1 */
27 mov R0 <- #Run
28 mov R1 <- 1
29 hvc
30 /* read x */
31 mov R1 <- #p
32 ldr R0 [R1]
33 halt
```

```
1  /* VM1 */
2  /* fetch handle */
3  mov R5 <- #rx
4  ldr R4 [R5]
5  mov R0 <- #MsgPoll
6  hvc
7  /* retrieve p */
8  mov R1 <- R4
9  mov R0 <- #Retrieve
10 hvc
11 /* x = x+2 */
12 mov R5 <- #p
13 ldr R3 [R5]
14 add R3 2
15 str R5 [R3]
16 /* yield */
17 mov R0 <- #Yield
18 hvc
```

Fig. 2. Code of the two known VMs in Figure 1. Additional notation is added to improve readablity. Symbols with prefix # are constant values: x is 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); *ptx* is the base addresses of the write-only messaging buffer (TX) of VM0, and *prx* is the read-only buffer (RX) of VM1. We assume that the two programs live at the start of two separate pages, pp_0 and pp_1.

Figure 2 is initially $(\texttt{Some}(0), \{0\}, \texttt{True})$, since VM0 has exclusive ownership on the page; and it is updated to $(\texttt{Some}(0), \{0, 1\}, \texttt{False})$ 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. str at line 4 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*.

2.2.2 Configuration. A *configuration* is a pair of a *state* together with an *execution mode*. A *state* of our operational semantics is composed of the aforementioned components for modeling memory access, plus those for HVCs:

$$State \stackrel{\text{def}}{=} \left\{ \begin{array}{llll} mem & : Word \rightharpoonup Word; & pgt & : PageID \rightharpoonup PageStatus; \\ regs & : VMID \to RegisterFile; & curr & : VMID; \\ trans & : Transactions; & mb & : VMID \to Mailbox; \end{array} \right\}$$

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 stated above, `PageFault` is used for page faults.

2.2.3 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 $\texttt{Some}((0, 1, \{p\}, \texttt{Share}), \texttt{False})$ (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 a transaction descriptor specifying the receiver and the page IDs of the pages in its TX page (lines 5–9 in the example). Next, the sending VM invokes a memory sending HVC, asking the hypervisor to create a transaction of the type given by the 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 (2)

$$\frac{\sigma.\mathtt{curr} = i \qquad \mathit{valid\_instr}(\sigma, i) = \mathtt{Some}(\mathsf{hvc}, a) \qquad \mathit{valid\_share}(\sigma, i) = \mathtt{Some}(i_r, s, h) \qquad \sigma' = \left\{ \begin{array}{l} \mathtt{mem} = \sigma.\mathtt{mem}; \quad \mathtt{curr} = \sigma.\mathtt{curr}; \quad \mathtt{mb} = \sigma.\mathtt{mb}; \\ \mathtt{pgt} = \sigma.\mathtt{pgt}\left[\begin{array}{l} p \mapsto (\mathtt{Some}(i), \{i\}, \mathtt{False}) \\ \mid (p \in s) \end{array}\right]; \\ \mathtt{regs} = \sigma.\mathtt{regs}[i]\left[\begin{array}{l} \mathsf{pc} \mapsto a + 1; \\ \mathsf{R0} \mapsto \mathit{encode}(\mathsf{Succ}); \\ \mathsf{R2} \mapsto h \end{array}\right]; \\ \mathtt{trans} = \sigma.\mathtt{trans} \\ \qquad [h \mapsto \mathtt{Some}((i, i_r, s, \mathsf{Share}), \mathtt{False})]; \end{array} \right\}}{(\mathtt{Normal}, \sigma) \rightarrow (\mathtt{Normal}, \sigma')}$$

Fig. 3. Reduction rule for Share

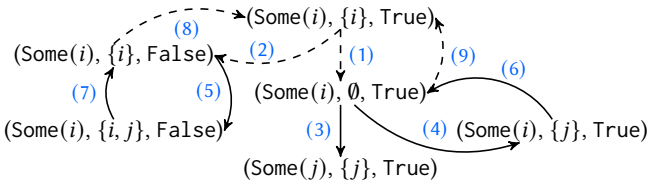

Fig. 4. The state transition system of the status of a page during a transaction. HVCs with dashed arrows are allowed for the sender i, and others are allowed for the receiver j. (1) Donate/Lend (2) Share (3) Retrieve(donation) (4) Retrieve(lending) (5) Retrieve(sharing) (6)(7) Relinquish (8)(9) Reclaim

and (5). 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 3). 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 metadata (*trans*).

VMs can invoke other HVCs with the same handle to refer to the transaction. For instance, with the hvc at line 10, VM1 Retrieves 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 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.

2.2.4 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*, where 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 3 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. We model the latter use case.

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.

2.2.5 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 where 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 can 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 6 of VM1), which also notifies the hypervisor that it is ready to take the next message (updates the RX buffer to `None`).

2.2.6 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).

3 REASONING ABOUT COMMUNICATING VMS

To validate our model of the FF-A HVC ABIs, we develop VMSL, a program logic designed to reason about key scenarios of VMs communicating using the FF-A ABIs. 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 so we base VMSL on separation logic [Reynolds 2002]. In particular, this will allow us to support *local reasoning* via the *frame rule* of separation logic, as we show below.

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 usually makes it possible to reason about instructions one at a time. This is a common problem, and we neatly capture a 'folklore' solution in a small Iris library in the form of *single-step weakest preconditions*. 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. We could 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 were designed for concurrency, and pose an undue burden in our setting where VMs are executed sequentially but not concurrently. Therefore, we introduce *resumption conditions*, an alternative

SS-MOV

$$\frac{(1)\mathrm{pc}@i \overset{\mathrm{reg}}{\longmapsto} a * (2)a \in_p s * (3)\mathrm{Pgt}@i \overset{\mathrm{acc}}{\longmapsto} s * (4)a \overset{\mathrm{mem}}{\longmapsto} \mathit{encode}(\mathrm{mov}\ r\ n) * (5)r@i \overset{\mathrm{reg}}{\longmapsto} -}{\mathit{SSWP}\ \mathtt{Normal}\ @\ i\ \left\{ (\mathtt{False}, \mathtt{Normal}).\ \left(\begin{array}{l} \mathrm{pc}@i \overset{\mathrm{reg}}{\longmapsto} a+1 * a \overset{\mathrm{mem}}{\longmapsto} \mathit{encode}(\mathrm{mov}\ r\ n) * \\ \mathrm{Pgt}@i \overset{\mathrm{acc}}{\longmapsto} s * r@i \overset{\mathrm{reg}}{\longmapsto} n \end{array} \right) \right\}}$$

Fig. 5. The proof rule for an immediate-to-register mov instruction. The updated resources are highlighted as in later rules. For simplicity, we omit the *encode* function that maps non-words including instructions and HVC identifiers to *Word*s in later rules. Also, we use $\mathit{IsInstr}@i(s, a, \mathrm{mov}\ r\ n)$ to represent that the mov instruction is stored at address a which belongs to the page that is one of VMi's accessible pages s ((1) to (4)).

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 our solutions to the two challenges on our example in Section 3.2; motivate and describe them in more detail in Section 3.3. With the solutions implemented in VMSL using Iris, we prove soundness of the logic with respect to the operational semantics of the machine model. All of VMSL's proof rules are sound with respect to our definition of weakest preconditions, 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. We use a simplified variant of Figure 2 without invoking the unknown VM2 (that is, with lines 22–25 of VM0 removed) as a running example.

3.1.1 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 $x + 2$.

3.1.2 Points-to Assertions. To state this formally, we introduce the classic register 'points-to' assertion, $r@i \overset{\mathrm{reg}}{\longmapsto} v$, 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 Table 1, we present a collection of similar points-to predicates of VMSL, together with their intuitive meanings. We introduce most of them gradually along with our explanation of how we use VMSL to reason about the example.

3.1.3 Formal Specification. Returning to the example, starting from a state where $\mathrm{R0}@0 \overset{\mathrm{reg}}{\longmapsto} x$, with other resources and 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 $\mathrm{R0}@0 \overset{\mathrm{reg}}{\longmapsto} x + 2$. We phrase this in VMSL by using a *weakest precondition* predicate $\mathit{WP}\ m\ @\ i\ \{Q\}$ which expresses the partial correctness of the VMi, i.e., we execute the VM with mode m and, if it terminates, then the postcondition Q holds:

Table 1. Selected collection of resources of VMSL

Predicate	Intuition
$r@i \overset{\text{reg}}{\longmapsto} w$	register r of VMi contains word w
$a \overset{\text{mem}}{\longmapsto} w$	value w is at location a
$\text{Pgt}@i \overset{\text{acc}}{\longmapsto} s$	VMi has access to pages s
$\text{Pgt}@p \overset{\text{own}}{\longmapsto} i$	VMi owns page p
$\text{Pgt}@p \overset{\text{excl}}{\longmapsto} i$	VMi's access to page p is exclusive
$\text{Tran}@h \overset{\text{tran}}{\longmapsto} t$	transaction t is bound to handle h
$\text{Tran}@h \overset{\text{rtrv}}{\longmapsto} b$	status of transaction bound to h is b
$\text{Mb}@i \overset{\text{rx}}{\longmapsto} p$	VMi's RX page is p
$\text{Mb}@i \overset{\text{tx}}{\longmapsto} p$	VMi's TX page is p
$\mathit{MemPage}(p, ws)$	content of page p is ws
$\mathit{FreshHandles}(hs)$	handles hs are fresh

$$\text{R0}@0 \overset{\text{reg}}{\longmapsto} x * \ldots (\text{other resources}) \vdash \mathit{WP}\ \texttt{Normal}\ @\ 0\ \left\{\texttt{m.m} = \texttt{Halted} * \text{R0}@0 \overset{\text{reg}}{\longmapsto} x + 2\right\}$$

3.2 Proving the Specification

3.2.1 First Instruction. To safely execute the first instruction of VM0, mov R5 #p (where p is an immediate), we need, as captured in our SS-MOV proof rule for an immediate-to-register mov, to know/show:

(1) The value a of the program counter, which indicates the location of the current instruction in the memory, as captured by the points-to for registers $\text{pc}@i \overset{\text{reg}}{\longmapsto} a$ (here, $\text{pc}@0 \overset{\text{reg}}{\longmapsto} pp_0$).
(2) Knowledge that the page at address a (here, pp_0) is in the accessible set s of *PageID*s...
(3) ...that are mapped for the current VM, as captured by ownership of the page tables points-to assertion, $\text{Pgt}@i \overset{\text{acc}}{\longmapsto} s$ (here, $\text{Pgt}@0 \overset{\text{acc}}{\longmapsto} s$).
(4) Ownership of the memory points-to resource for that memory location, $a \overset{\text{mem}}{\longmapsto} w$ (here, $pp_0 \overset{\text{mem}}{\longmapsto} w$), which contains a word w that is the encoding of an immediate-to-register mov instruction (here, mov R0 #p).
(5) Ownership of the register points-to resource for the affected register (here, $\text{R5}@0 \overset{\text{reg}}{\longmapsto} -$); we do not need to know what it contains (as signified by the use of $-$), but we must have the right to update it.

After the mov instruction, the VM does not lose control (so the switching bit is `False`), and the execution mode is still `Normal`. We get the updated resources back in our context; in particular, the program counter has been incremented, $\text{pc}@0 \overset{\text{reg}}{\longmapsto} pp_0 + 1$, and the register now contains the immediate, $\text{R5}@0 \overset{\text{reg}}{\longmapsto} p$; 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 keep the proof effort manageable.

SS-SHARE

$$(1)\mathit{ValidDesc}(\mathit{memtx}, i, j, \mathit{ps}) \land (2)\mathit{ps} \subseteq s \land (3)\mathit{hs} \neq \emptyset \land \mathit{IsHVC@}i(s, a, \mathsf{Share}) *$$
$$\mathsf{R1@}i \overset{\text{reg}}{\longmapsto} l * \mathsf{R2@}i \overset{\text{reg}}{\longmapsto} - * (4)\mathsf{Mb@}i \overset{\text{tx}}{\longmapsto} \mathit{ptx} * (5)\mathit{MemPage}(\mathit{ptx}, \mathit{memtx}) *$$
$$(6) \mathop{*}_{p \in ps} (\mathsf{Pgt@}p \overset{\text{own}}{\longmapsto} i * \mathsf{Pgt@}p \overset{\text{excl}}{\longmapsto} \mathsf{True}) * (7)\mathit{FreshHandles}(\mathit{hs})$$

$$\overline{\mathit{SSWP}\ \mathtt{Normal}\ @\ i\ \left\{(\mathtt{False}, \mathtt{Normal}).\ \begin{pmatrix} \mathsf{pc@}i \overset{\text{reg}}{\longmapsto} a+1 * a \overset{\text{mem}}{\longmapsto} \mathsf{hvc} * \mathsf{Pgt@}i \overset{\text{acc}}{\longmapsto} s * \\ \mathsf{R0@}i \overset{\text{reg}}{\longmapsto} \mathsf{Succ} * \mathsf{R1@}i \overset{\text{reg}}{\longmapsto} l * \\ \mathsf{Mb@}i \overset{\text{tx}}{\longmapsto} \mathit{ptx} * \mathit{MemPage}(\mathit{ptx}, \mathit{memtx}) * \\ \mathop{*}_{p \in ps} \mathsf{Pgt@}p \overset{\text{own}}{\longmapsto} i * \mathsf{Pgt@}p \overset{\text{excl}}{\longmapsto} \mathsf{False} * \\ \exists h.\, h \in \mathit{hs} \land \mathsf{R2@}i \overset{\text{reg}}{\longmapsto} h * \mathit{FreshHandles}(\mathit{hs} \setminus \{h\}) * \\ \mathsf{Tran@}h \overset{\text{tran}}{\longmapsto} (i, j, \mathit{ps}, \mathsf{Share}) * \mathsf{Tran@}h \overset{\text{rtrv}}{\longmapsto} \mathsf{False} \end{pmatrix}\right\}}$$

SS-RUN

$$(1)i \neq 0 \land \mathit{IsHVC@}0(s, a, \mathsf{Run}) * \mathsf{R1@}0 \overset{\text{reg}}{\longmapsto} i * (2)\mathit{RC}_{1/2}\mathsf{@}i\,\{\Psi_i\} * (3)\mathit{RC}_1\mathsf{@}0\,\{-\} *$$
$$(4)\left(\begin{pmatrix} \mathsf{pc@}0 \overset{\text{reg}}{\longmapsto} a+1 * a \overset{\text{mem}}{\longmapsto} \mathsf{hvc} * \mathsf{Pgt@}0 \overset{\text{acc}}{\longmapsto} s * \\ \mathsf{R0@}0 \overset{\text{reg}}{\longmapsto} \mathsf{Run} * \mathsf{R1@}0 \overset{\text{reg}}{\longmapsto} i * \Phi_{\mathit{othr}} * \mathit{RC}_1\mathsf{@}0\,\{\Psi_0\} \end{pmatrix} \mathrel{-\!\!*} \Psi_i * \Phi_{\mathit{rest}}\right) * (5)\Phi_{\mathit{othr}}$$
$$\overline{\mathit{SSWP}\ \mathtt{Normal}\ @\ 0\ \left\{(\mathtt{True}, \mathtt{Normal}).\ \mathit{RC}_{1/2}\mathsf{@}0\,\{\Psi_0\} * \Phi_{\mathit{rest}}\right\}}$$

WP-SSWP

$$\mathit{WP}\ \mathtt{m}\ @\ i\ \{\Phi\} \dashv\vdash \mathit{SSWP}\ \mathtt{m}\ @\ i\ \left\{(b, \mathtt{m}').\ \left((b \land \mathit{RCHolds@}i) \lor (\neg b)\right) \mathrel{-\!\!*} \mathit{WP}_{\mathcal{E}}\ \mathtt{m}'\ @\ i\ \{\Phi\}\right\}$$

RC-HOLD

$$\mathit{RCHolds@}i * \mathit{RC}_{1/2}\mathsf{@}i\,\{\Psi\} \vdash \triangleright \Psi * \mathit{RC}_1\mathsf{@}i\,\{\Psi\}$$

Fig. 6. Selected rules of VMSL

The SS-MOV rule, and all other single-instruction proof rules, use *SSWP*, our single-step variant of weakest preconditions. A *single-step weakest precondition* captures an intuitive idea (see §5): it is like a weakest precondition that only specifies the behaviour of a single step (an instruction). Applying a single-step weakest precondition 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). Single-step weakest preconditions allow us to reason about one instruction at a time. We show how to formally apply it to *weakest precondition* in Section 3.3.

3.2.2 Sharing. The following instructions prepare the descriptor and arguments for the Share HVC at line 13. They only involve register manipulations, which can be reasoned about in a similar way to the first instruction, and memory accesses. 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 SS-MOV.

Before reasoning about this specific Share, let us first consider the expected behaviour of a general Share HVC, specified by the SS-SHARE rule. To share pages represented by a set of PageIDs *ps*, VM*i* invokes a Share HVC with a descriptor in its TX page describing information about the transaction. Therefore, the proof rule requires (4) the TX page *ptx*; (5) ownership of the page with content *memtx*, which is expressed as memory points-tos for all locations of the page, connected by $*$; and (1) knowledge that the descriptor stored in *memtx* is valid. In addition, after validating the descriptor, the page table is examined to check whether VM*i* is allowed to share those pages in *ps*. Therefore, the rule requires (6) page ownership $\mathsf{Pgt}@p \overset{\text{own}}{\longmapsto} i$ and exclusiveness $\mathsf{Pgt}@p \overset{\text{excl}}{\longmapsto} \mathtt{True}$ to VM*i* of each page *p* in *ps*. The side condition (2) plus the resource for page access (included in *IsHVC*) further ensure that VM*i* has access to those pages. This information, combined, ensures 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 *hs* 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 postcondition. It is worth-noting that in practice a predicate can be built upon these resources, e.g. *TranHandles* shown in Section 4, leveraging the resource separation to guarantee that fresh and allocated handles are disjoint, which would reduce the handle availability reasoning to easy-to-discharge set disjointness side goals.

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 $\{pp_0; p; p_{tx}\}$. (3) is justified by assuming a non-empty *hs* in the specification. After applying the proof rule, we get $\mathsf{Tran}@h \overset{\text{tran}}{\longmapsto} (0, 1, p, \mathsf{Share})$ and $\mathsf{Tran}@h \overset{\text{rtrv}}{\longmapsto} \mathtt{False}$, 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.

3.2.3 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 into 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 VM1's RX page with a resource $\mathsf{RXState}@1 \mapsto \mathtt{Some}(1, 0)$ in the example, expressing that VM0 has passed one word to VM1.

3.2.4 Scheduling. At line 29, 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 VM*i*, denoted as $RC_1@i\,\{\Psi\}$, captures the resources Ψ that need to be handed over to VM*i* to resume its execution. We use *resumption condition*s to express communication protocols (reminiscent of session types [Honda et al. 2011; Yoshida and Gheri 2020]) 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 VM*i*:

(1) The VM being run is not the primary VM itself.
(2) VM0 has to satisfy the *resumption condition* of VM*i*, Ψ_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, Ψ_0, that VM*i* will have to satisfy to yield back.
(4) The magic wand $P \mathrel{-\!\!*} Q$ 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 Ψ_i, intuitively

the resources transferred to VMi, and the left over Φ_{rest}, i.e. the resources that are required by the rule, but not needed to show Ψ_i, that are still owned by VM0 afterwards.

(5) Other resources required to justify Ψ_i.

By picking the right Ψ_i and Ψ_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 Ψ_i as follows:

$$\Psi_i \stackrel{\text{def}}{=} p \overset{\text{mem}}{\longmapsto} x * \mathsf{Tran@}h \overset{\text{tran}}{\longmapsto} (0, 1, \{p\}, \mathsf{Share}) * \mathsf{Tran@}h \overset{\text{rtrv}}{\longmapsto} \mathtt{False} * \mathsf{Mb@1} \overset{\text{rx}}{\longmapsto} prx *$$
$$\mathsf{RXState@1} \mapsto \mathsf{Some}(1, 0) * prx \overset{\text{mem}}{\longmapsto} h * \mathsf{R0@0} \overset{\text{reg}}{\longmapsto} \mathsf{Run} * \mathsf{R1@0} \overset{\text{reg}}{\longmapsto} 1 * RC_{1/2}@0\,\{\Psi_0\}$$

Note that when VM1 yields back control to VM0, it needs to have established VM0's resumption condition, so we also include $RC_{1/2}@0\,\{\Psi_0\}$ in Ψ_i. VM1 thus can refer to Ψ_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 Ψ_0:

$$\Psi_0 \stackrel{\text{def}}{=} p \overset{\text{mem}}{\longmapsto} x + 2 * \mathsf{R0@0} \overset{\text{reg}}{\longmapsto} \mathsf{Yield} * \mathsf{R1@0} \overset{\text{reg}}{\longmapsto} 1$$

To justify (4), we let Φ_{othr} be Ψ_i except for its last three assertions, and Φ_{rest} naturally be the resources that are in the premise but not required by Ψ_i.

We get Φ_{rest} and $RC_{1/2}@0\,\{\Psi_0\}$ after applying the rule. To explain how to get resources stated in Ψ_0 out, we first introduce *RCHolds@i*. It assumes the resumption of VMi and can interact with the *resumption condition* of VMi by 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 Ψ holds *later*, i.e. after taking a step in the underlying model (this is used to break circularity of definitions [Jung et al. 2016, 2015]). Back to the example, we already get $RC_{1/2}@0\,\{\Psi_0\}$ in the postcondition, so we would be able to apply this rule and proceed with the proof with the transferred-back resources in Ψ_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.

3.2.5 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, $\mathtt{m} = \mathtt{Halted} \wedge \mathsf{R0@0} \overset{\text{reg}}{\longmapsto} x + 2$, and conclude the proof.

The proof of VM0 does not consider the code of VM1, due to the 'VM-modularity' of VMSL. All we needed was an abstract characterisation 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, but concludes in a different way, as VM1 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 Single-step Weakest Preconditions and Resumption Conditions

The example above shows how single-step weakest preconditions and resumption conditions are the two key components that make reasoning with VMSL manageable. We now discuss them in more detail, and point out how an expressive higher-order separation logic like Iris makes reasoning sound and tractable.

3.3.1 Single-step Weakest Preconditions. Single-step weakest preconditions allow us to reason about a single instruction at a time. Rule 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 WP-SSWP to transform a goal formulated in terms of weakest precondition into one formulated in terms of single-step weakest precondition, so that we can apply proof rules for individual instructions, and then proceed with the reasoning of the remaining instructions.

3.3.2 Resumption Conditions. We achieve modular reasoning between VMs through resumption conditions, which provide a form of rely-guarantee reasoning tailored for cooperative multitasking between VMs. 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 relinquishing control, and in exchange we can assume it when getting control back (as in the postcondition of WP-SSWP). This allows us to write specifications for individual VMs, and prove them separately without having 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 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 [Bornat et al. 2005; Boyland 2003]: 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 enough for both VMs, since SS-RUN requires merely half to run the secondary, and RC-HOLD requires merely half to obtain ownership of the resources in the *RC*. In the example above, the protocol is specified by the *RC* of VM1 with the *RC* of VM0 embedded into it. The *RC* of VM1 is split into two fractions owned by the two VMs so that they conform to the same protocol.

Many concurrent separation logics, including Iris, already define a standard mechanism to reason about concurrent programs: invariants. However, resumption conditions are more convenient for the scenarios we consider, as they only require the user to consider interference from other VMs when it occurs, namely at the point of yielding; invariants would force us to consider it (and show that it is not present) at every step of the program. Iris also defines 'non-atomic' invariants, which are a closer fit for our scenarios, as they can group multiple execution steps as a single critical section when holding an exclusive token. However, they do not address the issue completely: a sharing mechanism like invariants is still required to transfer those exclusive token between VMs.

Recursive Resumption Conditions. 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 unbounded. Consider a 'ping-pong' example, in which a primary VM and a secondary VMi just keep running each other; we can model this protocol as follows:

$$\Psi_i \stackrel{\text{def}}{=} \mathsf{R0@0} \overset{\text{reg}}{\longmapsto} \mathsf{Run} * \mathsf{R1@0} \overset{\text{reg}}{\longmapsto} i * RC_{1/2}@0 \left\{ \mathsf{R0@0} \overset{\text{reg}}{\longmapsto} \mathsf{Yield} * \mathsf{R1@0} \overset{\text{reg}}{\longmapsto} i * RC_{1/2}@i \, \{\Psi_i\} \right\}$$

The use of *RC* in Ψ_i ensures that Ψ_i is well-defined by the soundness of Iris higher-order ghost states and guarded recursion. Technically, *RC*s are defined using so-called saved propositions, which means that the recursive occurrence of Ψ_i is automatically guarded (even without an explicit 'later'

modality ▷) and hence Ψ_i is well-defined. Using a logic with guarded recursion like Iris means we do not need to be concerned about soundness of these definitions, as one would have to be if working directly over the operational semantics.

3.3.3 Formalising in Iris. We formalise VMSL using Iris because it allows us to capture and generalize the well-established ideas behind the two logical constructs. We use Iris's primitives and leverage its advanced features, such as higher-order ghost states and guarded recursion, as demonstrated in the recursive example above. The resulting solution is sound and compatible with existing Iris logical constructs thanks to our foundational approach. We use the combination of resumption conditions and invariants in Section 4, and believe such compatibility would also be useful to tackle for example interrupts and proper concurrency. Moreover, our solution is language/model-agnostic, therefore can be instantiated with different low-level languages and used to the reasoning of them – e.g., VMSL is obtained by instantiating it with the HVC model.

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 change its contents. 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, strong properties hold for the whole system. (2) When the primary VM is compromised, because the primary VM is where the scheduler resides, and because it therefore interacts with all the secondary VMs (at least for scheduling), these strong properties do not hold, but some 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 weakest preconditions 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 weakest precondition 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 weakest precondition for an unknown VM, and whether we can use VMSL to establish one.

Inspired by models for capabilities [Devriese et al. 2016; Georges et al. 2022a; Swasey et al. 2017], 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 VMi. Our logical relation states that, given the state of the page table and in-flight transactions that determine which memory pages VMi has or may get access to, as defined by *InterpAccess*, the execution of VMi can be safely resumed, as defined by *InterpExecute*:

$$\mathcal{R}(i) \stackrel{\text{def}}{=} \mathit{InterpAccess}(i) \twoheadrightarrow \mathit{InterpExecute}(i)$$

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

$$\forall i.\, i \neq 0 \longrightarrow \mathcal{R}(i)$$

From the perspective of proving the *FTLR*, *InterpAccess* can be regarded as a predicate specifying the exact resources we need to prove the execution of VMi. We define *InterpExecute* in terms of a *weakest precondition* 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:

$$\mathit{InterpExecute}(i) \stackrel{\text{def}}{=} \mathit{RCHolds@}i \twoheadrightarrow \mathit{WP}\ \texttt{Normal}\ @\ i\ \{\top\}$$

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).

4.1.1 Defining InterpAccess. During the execution, VMi may execute any valid instructions, and so we cannot make assumptions about the content of memory of VMi 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 *InterpAccess* for a VMi follows two principles: (1) It must allow us to characterise the behaviour of VMi enough to prove our desired safety property, whatever instructions VMi 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 instructions. (2) It should not needlessly limit our ability to reason about other VMs. Giving to VMi resources that VMj could own means we might not have necessary resources to prove the specification of VMj. Therefore, *InterpAccess*(i) should contain just enough resources to reason about VMi. These two principles make *InterpAccess*(i) the footprint of running an arbitrary program on VMi. Figure 7 shows the top-level definition of *InterpAccess*.

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

Among all the resources of *InterpAccess*(i), some are exclusively owned by VMi, and some have to be shared between VMi and other VMs due to the communication allowed by HVCs. The shared part is transferred from the primary to VMi upon resumption (via Ψ_i) and is given back to the primary upon yielding (via Ψ_0), using *RC*s. Ψ_i and Ψ_0 are parametrised by an extra τ', to represent new transactions allocated or updated during the suspension of VMi. The connection between τ and τ' is captured by the relation $\tau \sim \tau'$, that is that, the transactions in which VMi is the sender or receiver in τ cannot be touched by other VMs during its suspension, and therefore remain

$$\mathit{InterpAccess}(i) \stackrel{\text{def}}{=} \forall s_{acc}, \tau.\ (1)\mathsf{Pgt}@i \overset{\text{acc}}{\longmapsto} s_{acc} * (2)\mathit{PgtOea}(s_{oea}) * (3)\mathit{MemPages}(s_{oea} \cup \mathit{excl\_pages}(\tau)) * (4)\mathit{PgtTranP}(\tau) * (5)RC_{1/2}@i\,\{\Psi_i\} * \ldots$$

$$\Psi_i \stackrel{\text{def}}{=} \exists \tau'.\ \tau \sim \tau' \wedge (6)\mathit{TranHandles}(\tau') * (7)\mathit{PgtTranS}(\tau') * (8)\mathit{MemPages}(\mathit{shared\_pages}(\tau')) * (9)RC_{1/2}@0\,\{\Psi_0\} * \ldots$$

Fig. 7. The shape of the definition of *InterpAccess*(i). All predicates are implicitly parametrised by i if i is mentioned in their definitions. We refer readers to the Coq formalisation for the full definition.

unchanged in τ'. This relation allows us to unify the two, safely replacing τ with τ'. We then only work with τ', which includes all ongoing transactions when VMi is actually executed.

We present this definition by first considering the resources interpreting s_{acc} and τ' 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 them into owned and shared portions.

4.1.2 Intepreting s_{acc}. The interpretation of s_{acc} is split as follows: First, (1) states that these pages are accessible to VMi, which is required by all the proof rules (e.g. (3) of SS-MOV). Second, (2) provides page table resources for pages that VMi owns and has exclusive access to (denoted as s_{oea} and computed from s_{acc} and τ), which is defined as $\ast_{p \in s_{oea}} \mathsf{Pgt}@p \overset{\text{own}}{\longmapsto} i * \mathsf{Pgt}@p \overset{\text{excl}}{\longmapsto} \mathtt{True}$ (or $\mathit{PgtOE}(s_{oea}, i, \mathtt{True})$ in short). Those resources are required by the proof rules (e.g. (6) of SS-SHARE) if VMi shares pages that are in s_{oea}.

These two components are exclusively owned by VMi since no other VMs may require them. Another necessary but partially shared component is the memory of s_{acc}, $\mathit{MemPages}(s_{acc})$, which is required by rules for memory access instructions. We divide s_{acc} (and the predicate correspondingly) in two parts: memory pages that VMi has exclusive access to, and the remainder that is shared with other VMs. The former is captured by s_{oea} plus pages that are lent to VMi, collected by $\mathit{excl\_pages}(\tau')$, as in (3); the latter is collected by $\mathit{shared\_pages}(\tau')$ as in (7).

4.1.3 Interpreting τ'. In general, three kinds of resources could be necessary to allow VMi to perform memory sharing HVCs on t: $\mathsf{Tran}@h \overset{\text{tran}}{\longmapsto} t.\mathsf{meta}$ is necessary to refer to t for any sharing HVCs; $\mathsf{Tran}@h \overset{\text{rtrv}}{\longmapsto} t.\mathsf{retri}$ is necessary to retrieve the access to shared pages $t.\mathsf{pgs}$; and $\mathit{PgtOE}(t.\mathsf{pgs}, \_, \_)$ is necessary to update the status of the shared pages.

These resources are split into fractions such that some are owned by VMi, and some are shared. The owned and shared fractions are used to interpret transactions of τ and τ' respectively, and unified later by $\tau \sim \tau'$ (so they both interpret τ'). For instance, a points-to for transactions is split into three fractions that must agree on their values. One third in some cases is owned by VMi, and at least another one third is shared in all cases. The points-tos 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 t are shared, which allows us to derive the fact that pages shared by two transactions are disjoint by leveraging the exclusivity of $\mathit{PgtOE}_{2/3}$ that is derived from that of the underlying page table points-tos.

Now let us zoom in on several representative cases outlined in table 2 to see why those resources are distributed like this. In case "i, j, Share, `False`", VMi is the sender, and therefore the owner of the shared pages. 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. The owned fractions allow VMi to remember that it has shared $t.\mathsf{pgs}$ even after a suspension. The receiver

Table 2. Select cases of how a transaction t is interpreted. Column one gives metadata and state of t, where j and k are VMIDs of two other VMs. Columns two to four give the required fractions of the three kinds of required resources. $1/3 + 2/3$ under column two means $\mathsf{Tran}@h \overset{\mathsf{tran}}{\longmapsto}_1 t.\mathsf{meta}$ is required in total, with $1/3$ of it owned by the unknown VMi, and $2/3$ shared.

t.sndr, t.rcvr, t.type, t.retri	$\mathsf{Tran}@h \overset{\mathsf{tran}}{\longmapsto} t.\mathsf{meta}$	$\mathsf{Tran}@h \overset{\mathsf{rtrv}}{\longmapsto} t.\mathsf{retri}$	$\mathit{PgtOE}(t.\mathsf{pgs}, \_, \_)$
i, j, Share, `False`	$1/3 + 2/3$	1	$1/3 + 2/3$
i, j, Donate, `False`	1	1	1
j, i, Share, `True`	$2/3$	$1/2 + 1/2$	$2/3$
j, i, Lend, `True`	$2/3$	$1/2 + 1/2$	$2/3$
j, k, _, _	$1/3$	0	$2/3$

doesn't need them to Retrieve or Relinquish. In case "i, j, Donate, `False`", all resources are shared, as the receiver could Retrieve, which gives it ownership of the pages $t.\mathsf{pgs}$. In case "j, i, Lend, `True`", VMi as the receiver does not own page table resources nor the points-tos for transaction, as there is no way for it to get ownership of those pages (and full ownership of the three resources is 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, _, _", VMi 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, $\mathsf{Tran}@h \overset{\mathsf{tran}}{\longmapsto}_{1/3} (0, 1, \{p\}, \mathsf{Share})$ and $\mathsf{Pgt}@p \overset{\mathsf{own}}{\longmapsto} 0 * \mathsf{Pgt}@p \overset{\mathsf{excl}}{\longmapsto} \mathtt{False}$).

Resources specified in Table 2 are distributed in (4), (6), and (7). (6) includes the least amount of fractions required by all cases, i.e. $1/3$, 0, and $2/3$, of the three kinds of resources respectively, for each transaction in τ':

$$\mathop{\Large *}_{h \mapsto t \in \tau'} \mathsf{Tran}@h \overset{\mathsf{tran}}{\longmapsto}_{1/3} t.\mathsf{meta} * \mathit{PgtOE}_{2/3}(t.\mathsf{pgs}, t.\mathsf{sndr}, (t.\mathsf{type} =? \mathsf{Share}))$$

Remaining owned and shared fractions are distributed in (4) and (7) respectively with definitions of similar shapes as (6).

4.1.4 General Protocols. (9) in Figure 7 is one half of the resumption condition specifying which resources are supposed to be returned back to the primary VM to resume its execution. Generally speaking, the same resources transferred to VMi are passed back, plus the recursive resumption condition of VMi which allows the primary to run VMi multiple times.

$$\Psi_0 \overset{\mathsf{def}}{=} \exists \tau.\ \mathit{TranHandles}(\tau) * \mathit{PgtTranS}(\tau) * \mathit{MemPages}(\mathit{shared\_pages}(\tau)) * \ldots * \mathit{RC}_{1/2}@i\ \{\Psi_i\}$$

We call such a protocol specified by the two resumption conditions the *general protocol* of VMi. It is general in the sense that it specifies necessary resources to support arbitrary execution of VMi, for arbitrary numbers of resumptions, and 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.

4.1.5 Proving the FTLR. To show that the *FTLR* holds, we have to consider all possible instructions since the program of the VM is unknown. For each instruction, we apply the corresponding general proof rule of VMSL. See the Coq formalisation for the proof.

4.1.6 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

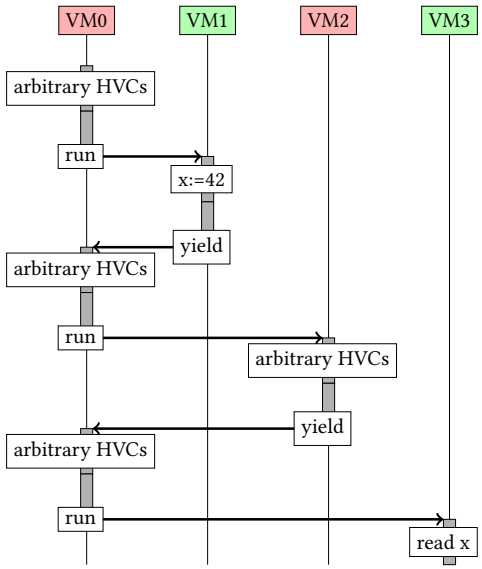

Fig. 8. A compromised primary VM is also contained: memory integrity (illustrating defensive code). This assumes VM1 and VM3 initially exclusively share a page p containing location x.

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 VM1 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_{acc} and τ such that the required resources are disjoint and consistent with resources required by the 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 τ be $\emptyset$, since at the beginning there are no transactions, and we let s_{acc} be $\{p_{tx2}; p_{rx2}; pp_2\}$. To show the weakest precondition for VM0, which now runs VM2 before VM1, we have to show the resumption condition of VM2 specified in *InterpAccess*(2). In particular, we let τ' be $\{h \mapsto (0, 1, \{p\}, \mathsf{Share}, \mathtt{False})\}$, whose interpretation in *TranHandles* will disallow any malicious HVCs, such as retrieving access to p, by VM2. The same resources are included in Ψ_0 and given back, so this transfer does not affect the reasoning about the two known VMs after running VM2.

4.1.7 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 8: 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

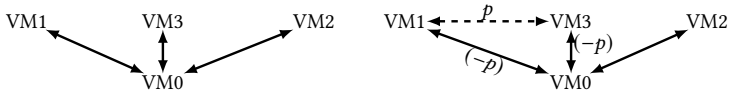

Fig. 9. An illustration of how resources are shared among VMs in Figure 8. Regular arrows represent the resources of the general protocol, where $(-p)$ means the resources of page p are excluded. Instead, those resources are shared via an invariant represented as the dashed arrow.

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, 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, *InterpExecute* is defined as just *WP* `0 @ Normal {⊤}`, and moreover *RCHolds* is not needed as we always run the primary first. The difference is in *InterpAccess*, which generalises the former to support running arbitrary secondary VMs, namely the extra power of the primary VM. From the perspective of resources, the new *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 4.1.1; and (2) resources required by resumption conditions of all secondary VMs to support running these VMs, which is basically their resumption conditions plus the union of resources required by them.

The crux of defining the new *InterpAccess* is specifying *all* the resumption conditions, i.e. protocols between all secondaries and the primary. For unknown secondaries, as shown in the previous subsection, 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 the example in Figure 1). Moreover, we cannot use the general protocol for known code either, as it is too general to be used to prove e.g. the example in Figure 8. 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 shared page p contains 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 9). 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 middleperson.

4.2.1 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 9, with the memory resources of page p.

4.2.2 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 and OS verification. There are several lines of work on hypervisor verification, including HASPOC [Baumann et al. 2016, 2019], SeKVM [Li et al. 2021a,b; Tao et al. 2021], Hyper-V [Leinenbach and Santen 2009], and seL4 [Klein et al. 2014, 2009].

The HASPOC project is aimed at designing 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 bisimulation between it and the concrete platform model. In their model, each VM's memory is isolated and cannot be shared; instead, inter-VM communication is restricted to a messaging mechanism similar to the one we model.

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.

Microsoft's Hyper-V is an industrial hypervisor partially verified with the VCC verification suite [Cohen et al. 2009], and their verification effort focuses on low-level concurrent C code. Most of their verification effort relates the hypervisor implementation to its specification, but not on validating that top-level specification, nor on its security properties.

seL4 is a formally verified OS kernel. Whereas in our setting, scheduling is outsourced to a primary VM, in their setting, scheduling is done by seL4 itself. In addition to functional correctness, seL4 includes a proof of some non-interference properties [Murray et al. 2013], which they prove over the kernel specification. The integrity result for seL4 [Sewell et al. 2011] considers a small operating system that manages a set of capabilities with various authorities (write, read, send, receive, grant, etc.) over various objects. Their operating system corresponds to the combination of our hypervisor and a "receptive" primary that waits for requests, checks they are allowed, and executes them. In that setting, they consider what kind of capabilities are accessible through privilege escalation. This is similar to the way in which our logical relations have to consider what can be acquired transitively through transactions and memory.

These efforts primarily focus on verifying the implementation of system software (including APIs exposed to clients). Our work is complementary, in that our approach factors the integrity (but

not the secrecy) part of their security results into a logic to reason about concrete programs using hypercall APIs, and a logical relation that captures isolation. This, in contrast to their approaches, enables us to give specifications and verify individual concrete scenarios, whereas, in our terms, their results are concerned with composing exclusively unknown VMs.

In addition, these lines of work make drastic simplifying assumptions, as the actual behaviour of page tables, especially in the presence of concurrency, is only beginning to be understood precisely enough for verification [Simner et al. 2022]. Nonetheless, there is some work on hypervisor verification against authoritative models: Nienhuis et al. [Nienhuis et al. 2020] and Bauereiss et al. [Bauereiss et al. 2022] prove security properties above full-scale, authoritative, formal ISA models of the CHERI and Morello capability architectures. These properties are finer-grained than ours thanks to capabilities, but weaker in that they are architectural invariants, and thus cannot rely on properties of known code. Sammler et al. [Sammler et al. 2022] 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 [Armstrong et al. 2021]. They focus on verifying local specifications of known code, including some exception handlers.

Reasoning about low-level code. The details of low-level code make it a natural target for mechanisation, and there is extensive work on the topic. Our work follows in the footsteps of the CAP [Feng and Shao 2005; Ni and Shao 2006; Ni et al. 2007; Yu and Shao 2004] family of Hoare logics for low-level code, which tackle for example code pointers and cooperative multitasking (which we return to later). In mechanising their logics directly in Coq, without an intermediate logic like Iris, they identify challenges concerning higher-order code (via code pointers), separation, rely-guarantee reasoning, etc., and also note opportunities offered by mechanisation, for example 'open' proof rules that are defined as lemmas over the operational semantics rather than hard-coded into the logic. Concurrently with the CAP work, mechanised variants of separation logic have long been used to reason about assembly code [Cai et al. 2007; Jensen et al. 2013; Kennedy et al. 2013; Myreen and Gordon 2007]. Iris generalises this approach, building on separation logic to encapsulate the logical constructions that are helpful to reason about programming languages in a language-independent way. Our work (like Georges et al. [Georges et al. 2021a]) demonstrates how such a rich logic does indeed make it tractable to tackle many of the challenges of low-level code identified by the CAP line of work.

Single-step weakest preconditions. Decomposing reasoning about a sequence of instructions into reasoning about each instruction one by one is quite intuitive, but often raises proof engineering challenges, and some solutions are 'folklore'. For example, Erbsen et al. [Erbsen et al. 2021, §4.3] capture individual steps, and compose them with an 'eventually' operator similar to a transitive closure. Our single-step weakest precondition, like the standard Iris weakest precondition, is defined purely in terms of the type of operational semantics that Iris takes as input, and thus factors out this aspect of instantiating Iris for low-level code, independently of the language. For example, we believe that our approach could be used by the capability machine formalisation of Georges et al. [Georges et al. 2021a] to simplify some of their proof engineering.

Cooperative multitasking and resumption conditions. Programming over the fragment of the FF-A hypercall API we consider, where secondary VMs run until they explicitly yield to the primary VM, is effectively a form of cooperative multitasking with a programmed scheduler. Again, we follow in the footstep of the CAP line of work [Feng and Shao 2005; Yu and Shao 2004], but benefit from a modern, mechanised separation logic. Moreover, in our terms, the CAP setting corresponds to only composing known secondary VMs sharing some pages with a primary that merely schedules

secondary VMs. Using our logical relations which capture bounds on the effect of arbitrary code, we go further, and capture the composition of known and unknown code.

Capability machines. Capabilities [Arm 2021; Carter et al. 1994; Watson et al. 2019; Wilkes and Needham 1979] are an alternative hardware mechanism for access control, in the form of dynamically checked unforgeable tokens of authority, typically granting some type of finer-grained access to a portion of memory. Proofs of safety for capability machines have also used unary, unityped logical relations, e.g. [Georges et al. 2021b, 2022b; Skorstengaard et al. 2019]. However, these logical relations are quite different from ours, because of the different underlying mechanisms. Their logical relation involves recursion through the heap, as a capability can give access a portion of the heap which gives access to further capabilities; whereas in our setting, there is a clear stratification of page tables 'above' the memory accessible to VMs. Because we do not have this recursion, a VM does not need to hand over all of its memory to a global invariant, and instead can locally keep the resources for the memory that it does not share, which leads to more direct reasoning at the expense of some complexity in the definition of our logical relation.

6 CONCLUSION

We have formalised a substantial fragment of Arm's FF-A ABIs 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*, which capture interaction between VMs and thereby reduces reasoning about their interaction to sequential reasoning. 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. Finally, we have applied these to prove security in key scenarios that capture the typical interaction cases between VMs with various trust relations.

Future work includes extending our model with concurrency and non-cooperative scheduling. We are also interested in adapting our model to the pKVM [Deacon 2020; Google LLC 2021; Perret 2020] ABIs, which is different from the FF-A ABIs 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for comments and suggestions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and in part by Google Android Security and PrIvacy REsearch (ASPIRE) Awards to Pharabod-Pichon and Birkedal. We would also like to thank Alix Trieu for earlier discussions.

REFERENCES

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 247–256. https://doi.org/10.1109/LICS.2001.932501

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. In *Philosophical Transactions of the Royal Society A*, Vol. 375. Issue 2104. https://doi.org/10.1098/rsta.2016.0331

Arm. 2021. Morello project. Retrieved July 6, 2021 from https://www.morello-project.org/

Arm Ltd. 2022. *Arm Firmware Framework for Arm A-profile version 1.1 - DEN0077A.* Technical Report. https://documentation-service.arm.com/static/624d5f52dc9d4f0e74a54e5f

Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, Alexandra Silva and K. Rustan M. Leino (Eds.). 303–316. https://doi.org/10.1007/978-3-030-81685-8_14

Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, Ilya Sergey (Ed.). 174–203. https://doi.org/10.1007/978-3-030-99336-8_7

Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. 2016. A high assurance virtualization platform for ARMv8. In *2016 European Conference on Networks and Communications (EuCNC)*. 210–214. https://doi.org/10.1109/EuCNC.2016.7561034

Christoph Baumann, Oliver Schwarz, and Mads Dam. 2019. On the verification of system-level information flow properties for virtualized execution platforms. In *J Cryptogr Eng 9*. 243–261. https://doi.org/10.1007/s13389-019-00216-4

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified Self-Modifying Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/1250734.1250743

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-Based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 319–327. https://doi.org/10.1145/195473.195579

Vijay Chidambaram. 2018. We found a bug in a verified file system! Twitter. https://twitter.com/vj_chidambaram/status/1047505696533741568

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

Will Deacon. 2020. Virtualisation for the Masses: Exposing KVM on Android. http://linux-kernel.uio.no/pub/linux/kernel/people/will/slides/kvmforum-2020-edited.pdf.

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 147–162. https://doi.org/10.1109/EuroSP.2016.22

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration verification across software and hardware for a simple embedded system. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 604–619. https://doi.org/10.1145/3453483.3454065

Xinyu Feng and Zhong Shao. 2005. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 254–267. https://doi.org/10.1145/1086365.1086399

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021a. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2022a. *Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code.* Technical Report. Aarhus University. https://cs.au.dk/~birke/papers/cerise.pdf

Aïna Linn Georges, Armaël Guéneau, Thomas Van-Strydonck, Amin Timany, Dominique Trieu, Alix Devriese, and Lars Birkedal. 2021b. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. In *Journées Francophones des Langages Applicatifs 2021*. https://cris.vub.be/ws/portalfiles/portal/55081793/paper.pdf

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022b. Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 74 (apr 2022), 30 pages. https://doi.org/10.1145/3527318

Google LLC. 2021. pKVM. https://android-kvm.googlesource.com/linux/+/refs/heads/pkvm/.

Hafnium development team. 2022. Hafnium — A security-focussed type-1 hypervisor. https://opensource.google/projects/hafnium.

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6536)*, Raja Natarajan and Adegboyega K. Ojo (Eds.). Springer, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4

Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-Level Separation Logic for Low-Level Code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 301–314. https://doi.org/10.1145/2429069.2429105

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. 2013. Coq: the world's best macro assembler?. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, Ricardo Peña and Tom Schrijvers (Eds.). ACM, 13–24. https://doi.org/10.1145/2505879.2505897

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70. https://doi.org/10.1145/2560537

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis Dams (Eds.), Vol. 5850. Springer, 806–809. https://doi.org/10.1007/978-3-642-05089-3_51

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021a. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3953–3970. https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021b. A Secure and Formally Verified Linux KVM Hypervisor. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1782–1799. https://doi.org/10.1109/SP40001.2021.00049

Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. Supplementary material: Coq development of VMSL. https://doi.org/10.5281/zenodo.7813157

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy*. 415–429. https://doi.org/10.1109/SP.2013.35

Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal) *(TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 568–582. https://doi.org/10.5555/1763507.1763565

Zhaozhong Ni and Zhong Shao. 2006. Certified assembly programming with embedded code pointers. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 320–333. https://doi.org/10.1145/1111037.1111066

Zhaozhong Ni, Dachuan Yu, and Zhong Shao. 2007. Using XCAP to Certify Realistic Systems Code: Machine Context Management. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4732)*, Klaus Schneider and Jens Brandt (Eds.). Springer, 189–206. https://doi.org/10.1007/978-3-540-74591-4_15

Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP40000.2020.00055

Quentin Perret. 2020. Protected KVM: Memory protection of KVM guests in Android. https://linuxplumbersconf.org/event/7/contributions/780/.

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). 825–840. https://doi.org/10.1145/3519939.3523434

Thomas Sewell, Simon Winwood, Peter Gammie, Toby C. Murray, June Andronick, and Gerwin Klein. 2011. seL4 Enforces Integrity. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 325–340. https://doi.org/10.1007/978-3-642-22863-6_24

Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 143–173. https://doi.org/10.1007/978-3-030-99336-8_6

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 89, 26 pages. https://doi.org/10.1145/3133913

Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 866–881. https://doi.org/10.1145/3477132.3483560

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. 2019. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7).* Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html

M. V. Wilkes and R. M. Needham. 1979. *The Cambridge CAP Computer and Its Operating System.* Elsevier. https://www.microsoft.com/en-us/research/publication/the-cambridge-cap-computer-and-its-operating-system/

Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 11969)*, Dang Van Hung and Meenakshi D'Souza (Eds.). Springer, 73–93. https://doi.org/10.1007/978-3-030-36987-3_5

Dachuan Yu and Zhong Shao. 2004. Verification of safety properties for concurrent assembly code. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 175–188. https://doi.org/10.1145/1016850.1016875

Received 2022-11-10; accepted 2023-03-31