Robust and Compositional Verification of Object Capability Patterns

DAVID SWASEY, DEEPAK GARG, and DEREK DREYER, MPI-SWS

In scenarios such as web programming, where code is linked together from multiple sources, *object capability patterns* (OCPs) provide an essential safeguard, enabling programmers to protect the private state of their objects from corruption by unknown and untrusted code. However, the benefits of OCPs in terms of program verification have never been properly formalized. In this paper, building on the recently developed Iris framework for concurrent separation logic, we develop OCPL, the first program logic for compositionally specifying and verifying OCPs in a language with closures, mutable state, and concurrency. The key idea of OCPL is to account for the interface between verified and untrusted code by adopting a well-known idea from the literature on security protocol verification, namely *robust safety*. Programs that export only properly wrapped values to their environment can be proven robustly safe, meaning that their untrusted environment cannot violate their internal invariants. We use OCPL to give the first general, compositional, and machine-checked specs for several commonly-used OCPs—including the *dynamic sealing*, *membrane*, and *caretaker* patterns—which we then use to verify robust safety for representative client code. All our results are fully mechanized in the Coq proof assistant.

General Terms: Object capabilities, robust safety, separation logic, logical relations, compositional verification.

1 INTRODUCTION

Suppose you have a mutable reference ℓ whose contents you care about, meaning that you want to impose some invariant on it (*e.g.,* ℓ always points to an even number). Suppose further that you want to share access to ℓ with code you did not write and that you do not trust to preserve the invariant on ℓ. To ensure the invariant on ℓ is maintained, you therefore do not want to pass the untrusted code the reference ℓ directly. Instead, you might construct a *read-only wrapper* w as follows:

$$\mathit{readonly} \triangleq \lambda r.\ \lambda\_.\ !r \qquad\qquad w \triangleq \mathit{readonly}\,\ell$$

Here, *readonly* transforms a reference r into a thunk that, when applied, returns the current contents of r. The expression w applies *readonly* to our reference of interest ℓ, constructing a function for reading ℓ's contents. You can now pass w to untrusted code without worrying about it corrupting your invariant on ℓ.

Wrappers like w are often called *object capabilities*, and the wrapper construction function *readonly* is a very simple example of an *object capability pattern* (OCP) (Miller et al. 2000). Although OCPs date back at least to the 1970s (Morris 1973), they have gained increased currency in recent years, both in new languages centered around object capabilities (Miller et al. 2000; Mettler et al. 2010; Spiessens and Roy 2004; Stiegler and Miller 2006) and in the context of web programming, where interfacing with untrusted code is commonplace. Web sandboxing systems like Yahoo!'s ADsafe (Crockford 2008; Politz et al. 2014) and Google's Caja (Miller et al. 2008), for example, elaborate untrusted JavaScript source code into an ostensibly safe subset of JavaScript while introducing wrappers that attenuate access to JavaScript libraries. Caja relies on the so-called membrane pattern, which automatically wraps all objects crossing to untrusted, sandboxed code so that access to the underlying objects is appropriately restricted.

OCPs are believed to provide crucial security guarantees because in a highly "dynamic" language like JavaScript, objects provide essentially no data abstraction on their own—the only way to enforce data abstraction is to hide private state in the environment of a closure (*e.g.,* as ℓ was hidden in the environment of *readonly* ℓ in the example above). So OCPs, which use closures to mediate access to private state in a systematic way, are one of the few effective mechanisms available in a language like JavaScript for enforcing data abstraction in the presence of possibly malicious code.

Unfortunately, despite the ubiquity of OCPs in modern web programming, remarkably little attention has been paid to the question of what exactly are the security guarantees that such OCPs are supposed to provide, and how might we prove that they actually provide them. Even in the case of the extremely basic *readonly* pattern shown above, it is not at all obvious what is the "right" formal specification for *readonly*. What, in particular, are the formal conditions on ℓ that are needed to guarantee that *readonly* ℓ can be "safely" shared with untrusted code? If ℓ merely points to an integer, no conditions may be necessary, but what if ℓ points to a closure or some other higher-order object? How do we know that giving readonly access to ℓ will not indirectly give untrusted code a way of gaining full access and violating important invariants maintained by the user of this OCP?

The most recent, state-of-the-art attempt to grapple with these types of questions is due to Devriese et al. (2016). They build a Kripke logical relations model for reasoning about object capabilities in a language with higher-order state, and they use their model to verify several concrete examples of capability-wrapped user code. In each case, they demonstrate that invariants on the private state of the user code are preserved even when the user code is shared with unknown attacker code. However, their model is limited in that it provides no way to *compositionally specify* what an OCP does, and it provides no clear specification of the general property that a piece of user code must satisfy in order to be safely shareable with untrusted code. Furthermore, they only consider very simple capability patterns, none as complex as, say, the aforementioned membrane pattern.

In this paper, we present **OCPL** (a ***L***ogic for ***OCP***s), the first formal system for *compositionally* specifying and verifying the security guarantees provided by OCPs, in the context of a simple but representative programming language with higher-order functions, state, and concurrency.[1] In contrast to prior work, OCPL enables one to reason modularly about both OCP implementations and user code that depends on them, and to specify a general property on user code that ensures such code can be safely shared with untrusted code without having its internal invariants violated. We use OCPL to reason about several commonly-used OCPs, including the dynamic sealing (sealer-unsealer), membrane, and caretaker patterns, and in so doing, provide the first formal explanation of what these OCPs achieve.

OCPL is a program logic derived from **Iris**, a recently developed framework for higher-order concurrent separation logic (Jung et al. 2015, 2016; Krebbers et al. 2017a). Iris was originally proposed as a very general logic with a few simple primitives, using which one can derive advanced proof principles from a variety of modern separation logics as needed. Moreover, Krebbers et al. (2017b) have recently developed a powerful new proof mode for Iris in the Coq proof assistant. The *Iris proof mode* enables one to carry out interactive, tactical, machine-checked proofs about programs in the Iris logic (embedded in Coq) in much the same way as one normally carries out interactive tactical proofs when working in the meta-logic of Coq itself. By virtue of building OCPL on top of Iris, we inherit the flexibility of the existing Iris framework, as well as its support for mechanizing proofs about programs in Coq. All our results and examples are, in fact, fully verified in Coq.[2]

The key idea in OCPL is how it characterizes the interface between verified user code and untrusted code, via the concept of a "low-integrity value" (or *low value* for short) adapted from the literature on verification of security protocols (Abadi 1999). Roughly speaking, a low value is a value that can be safely shared with untrusted code, such as the closure wrappers returned by OCPs. More precisely, a low value is a value from which no code can possibly extract a direct reference to private state. To formalize this notion, we employ a logical relation, which is easy to define using Iris's built-in support for guarded recursive predicates. Then, with the idea of low values in hand, it becomes clear how to specify when a piece of user code can be safely linked with untrusted code—namely, when the only values passed back and forth between them are low values.

[1]Despite the name, object capabilities are not fundamentally limited to object-oriented programming languages.
[2]The Coq formalization is available as supplementary material for the review process.

To make things concrete, let us return to our motivating example. Using the notion of low values, we can prove the following specification for the *readonly* pattern:

$$\forall \ell.\quad \{\top\}\ !\ell\ \{x.\ \mathsf{lowval}\, x\} \quad \Rightarrow \quad \{\top\}\ \mathit{readonly}\, \ell\ \{f.\ \mathsf{lowval}\, f\}$$

The precondition of this spec says that, in order to apply *readonly* to a location ℓ, we must first prove that dereferencing ℓ always produces a low value. (This makes sense because once we pass the read-only wrapper to untrusted code, it may invoke the wrapper and obtain the contents of ℓ at any time.) The payoff is in the postcondition: the closure returned by *readonly* ℓ is itself a low value and may therefore be safely shared with untrusted code.

Given this specification for *readonly*, let us now consider a client of *readonly* that wishes to maintain an invariant on its private (*i.e.,* high-integrity) state:

$$\begin{aligned} \mathit{usetwo} \triangleq\ & \mathtt{let}\ r = \mathtt{ref}\ 2\ \mathtt{in} \\ & \mathtt{let}\ w = \mathit{readonly}\, r\ \mathtt{in} \\ & \mathtt{let}\ \mathit{use} = \lambda\_.\ \mathtt{assert}\ ((!r) = 2)\ \mathtt{in} \\ & (\mathit{use}, w) \end{aligned}$$

Here, the expression *usetwo* allocates a reference cell r containing the value 2, and returns a pair of functions, one of which simply asserts that r continues to contain 2, and the other of which is the result of *readonly* r. The reference cell r is, however, kept private (*i.e.,* hidden in the environments of the closures *use* and w).

Our aim is to verify that *usetwo* can be safely linked with arbitrary untrusted code. To do this, we first prove that it satisfies the spec $\{\top\}\ \mathit{usetwo}\ \{x.\ \mathsf{lowval}\, x\}$ which means that we can always run *usetwo* and, if we do, it returns a low value. It is easy to see intuitively why this spec holds: reference cell r *always* contains 2, which is trivially low, so together with the spec for *readonly*, this is sufficient to ensure that the closures returned by *usetwo* are low as well.

Next, we appeal to a general meta-theorem—and one of the main technical results of this paper—called *robust safety*. Robust safety states that, if some user code satisfies a spec like the one given above for *usetwo*—*i.e.,* a spec whose postcondition stipulates that the resulting value is low-integrity—then we can run that verified user code under an arbitrary adversarial context C (*i.e.,* any context C that does not *itself* contain any `assert` statements), and we will be assured that the execution of the resulting program will never result in a violation of any of the user code's internal assertions. Thus, in the particular case of *usetwo*, we know that the assertion that r equals 2 will never fail.

Robust safety is a well-known meta-theorem in the security literature (Bengtson et al. 2011; Gordon and Jeffrey 2001), but it has not heretofore been employed in the context of object capability programming. One of the central contributions of this paper is the observation that robust safety is exactly the property a language must satisfy in order to support OCPs. Moreover, as we will demonstrate via a range of interesting examples, the notion of low-integrity values is essential to compositionally specifying an OCP's contribution toward the robust safety of programs that use it.

The remainder of the paper is structured as follows. First, in §2, we introduce some basics of OCPL, along with our formalization of low values, explain how we verify our motivating *readonly* example, and state the key metatheorems of adequacy and robust safety. In §§3–5, we present specifications for several more complex and realistic OCPs, along with examples of how to use those specifications to verify representative clients. Finally, in §6, we conclude with a discussion of related work.

$$
\begin{aligned}
\mathit{Lit} \ni \mathit{lit} &::= n \mid b \\
\mathit{Exp} \ni e &::= x \mid \mathtt{rec}\ f\ x.\ e \mid e\ e \mid \mathit{lit} \mid \mathit{unop}\ e \mid \mathit{binop}\ e\ e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid () \mid (e, e) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid \\
&\quad\ \mathtt{inl}\ e \mid \mathtt{inr}\ e \mid \mathtt{case}\ e\ \mathtt{of}\ \mathtt{inl}\ x \Rightarrow e \,|\, \mathtt{inr}\ x \Rightarrow e \mid \mathtt{fork}\ e \mid \ell \mid \mathtt{ref}\ e \mid\ !e \mid e \leftarrow e \mid \mathtt{CAS}\ e\ e\ e \mid \\
&\quad\ \mathtt{assert}\ e \\
\mathit{Val} \ni v &::= \mathtt{rec}\ f\ x.\ e \mid \mathit{lit} \mid () \mid (v, v) \mid \mathtt{inl}\ v \mid \mathtt{inr}\ v \mid \ell \\
\mathit{Ectx} \ni K &::= \text{(standard left-to-right evaluation contexts)}
\end{aligned}
$$

$$
h \in \mathit{Heap} \triangleq \mathit{Loc} \overset{\text{fin}}{\rightharpoonup} \mathit{Val} \qquad\qquad g \in \mathit{GoodnessBit} \triangleq \{\mathtt{OK}, \mathtt{Fail}\}
$$

$$
\sigma \in \mathit{State} \triangleq \mathit{Heap} \times \mathit{GoodnessBit} \qquad\qquad T \in \mathit{Tpool} \triangleq \mathit{Exp}^*
$$

$\boxed{e; \sigma \rightarrow_{\mathrm{r}} e'; \sigma'; \overrightarrow{e_f}}$ Head reduction (selected rules)

$$
\frac{(\mathtt{rec}\ f\ x.\ e)\ \text{closed}}{(\mathtt{rec}\ f\ x.\ e)\ v; \sigma \rightarrow_{\mathrm{r}} e[v/x, \mathtt{rec}\ f\ x.\ e/f]; \sigma; \varepsilon} \qquad \mathtt{assert}\ \mathtt{true}; \sigma \rightarrow_{\mathrm{r}} (); \sigma; \varepsilon
$$

$$
\mathtt{assert}\ \mathtt{false}; (h, g) \rightarrow_{\mathrm{r}} (); (h, \mathtt{Fail}); \varepsilon \qquad \mathtt{fork}\ e; \sigma \rightarrow_{\mathrm{r}} (); \sigma; e \qquad \mathtt{ref}\ v; (h, g) \rightarrow_{\mathrm{r}} \ell; (h \uplus [\ell \mapsto v], g); \varepsilon
$$

$$
\frac{\sigma.h(\ell) = v}{!\ell; \sigma \rightarrow_{\mathrm{r}} v; \sigma; \varepsilon} \qquad \frac{h(\ell) = v}{\ell \leftarrow w; (h, g) \rightarrow_{\mathrm{r}} (); (h[\ell \mapsto w], g); \varepsilon} \qquad \frac{\sigma.h(\ell) \neq v_1}{\mathtt{CAS}\ \ell\ v_1\ v_2; \sigma \rightarrow_{\mathrm{r}} \mathtt{false}; \sigma; \varepsilon}
$$

$$
\mathtt{CAS}\ \ell\ v_1\ v_2; (h \uplus [\ell \mapsto v_1], g) \rightarrow_{\mathrm{r}} \mathtt{true}; (h \uplus [\ell \mapsto v_2], g); \varepsilon
$$

$\boxed{T; \sigma \longrightarrow T'; \sigma'}$ Threadpool reduction

$$
\frac{e; \sigma \rightarrow_{\mathrm{r}} e'; \sigma'; \overrightarrow{e_f}}{(T_1, K[e], T_2); \sigma \longrightarrow (T_1, K[e'], T_2, \overrightarrow{e_f}); \sigma'}
$$

Fig. 1. Heap language with assertions

2 ROBUST SAFETY AND OCPL

In this section, we define the semantics of the programming language we will be reasoning about (§2.1), present the main ideas of OCPL and show how to use it to reason formally about our motivating example (§2.2), and state key metatheorems for OCPL (§2.3).

2.1 A higher-order concurrent heap language with assertions

The programming language we consider in this paper, which we call HLA, is a fairly standard higher-order concurrent imperative language, essentially the same as the one studied by Krebbers et al. (2017a), but with one important extension. We define the language in Fig. 1. The core of the language is a call-by-value λ-calculus with recursive functions, products, sums, references, and fork-based concurrency. The definitions of free and bound variables and capture-avoiding substitution are standard and omitted. We endow the language with an operational semantics in the style of Felleisen and Hieb (1992). The head reduction judgment $e; \sigma \rightarrow_{\mathrm{r}} e'; \sigma'; \overrightarrow{e_f}$ describes the single steps of computation that may be taken by individual threads. It means that, starting in state σ, a thread running expression e may step to expression e' while changing the state to σ' and forking off a list of threads running expressions $\overrightarrow{e_f}$. Threadpool reduction $T; \sigma \longrightarrow T'; \sigma'$ lifts head reduction to lists of concurrently running threads. We leave implicit the choice of operators *unop* and *binop*, but note that they reduce according to (partial) evaluation functions and are subject to a semantic constraint; for example, the reduction rule for unary operators

$$\mathit{unop}\, v; \sigma \rightarrow_{\mathrm{r}} v'; \sigma; \varepsilon \qquad \text{if eval } \mathit{unop}\, v = v'$$

employs a partial function eval : $\mathit{Unop} \rightarrow \mathit{Val} \rightharpoonup \mathit{Val}$ (which, for the robust safety theorem presented below, is assumed to send low-integrity values to low-integrity values). Using compare-and-swap (CAS), one can implement locks (and other concurrency primitives).[3] We use the symbol '_' in place of a bound variable to indicate any variable not occurring free in the scope of the binder. In addition to standard derived forms (*e.g.*, *n*-ary products and let expressions), we write

$$\lambda x.\, e \triangleq \texttt{rec}\ \_\ x.\, e$$

$$\lambda x_1 \cdots x_n.\, e \triangleq \lambda x_1.\ \cdots.\ \lambda x_n.\, e$$

$$\texttt{rec}\ f\ x_1 \cdots x_n.\, e \triangleq \texttt{rec}\ f\ x_1.\ \lambda x_2 \cdots x_n.\, e$$

and we use pattern-matching notation to deconstruct products, rather than a series of `fst` and `snd` projections.

The one new feature in HLA, not present in Krebbers et al.'s calculus, is *assertion expressions*, `assert` *e*, which serve to specify safety properties that *should* always hold dynamically. Assertions in HLA operate a little differently from assertions in languages like C or Java, where failed assertions abort execution or raise an exception. Here, *e* is a boolean expression. If *e* evaluates to `true`, then `assert` *e* has no effect. But if *e* evaluates to `false`, then `assert` *e* has the effect of irreversibly setting a "goodness bit" *g*, which is maintained as part of the machine state, to `Fail`. (The bit is initially set to `OK`.) This goodness bit simply checks whether any assertion expression has dynamically failed during execution. Ultimately, our robust safety theorem will use this goodness bit to ensure that, for properly verified code, the goodness bit will remain `OK` throughout execution (and hence all dynamic `assert` checks must succeed).

The reader may wonder why we employ a goodness bit instead of just saying that `assert false` gets stuck or aborts the program. The reason is simple: we wish to use OCPL to reason not only about fully verified code but also about the behavior of verified code when linked with untrusted code. So it is important that we have a way of verifying Hoare triples even for code that may very well get stuck (*i.e.,* fail to make progress) thanks to dynamic type errors introduced by the untrusted code. We nevertheless want to say that `assert` expressions in such code always succeed, so we must differentiate `assert false` from other stuck states.

2.2 A program logic for reasoning about OCPs

We now present some of the essential features of OCPL, our logic for reasoning about OCPs.

Progressive vs. non-progressive triples. OCPL is a Hoare-style program logic derived from Iris, a modern separation logic for higher-order concurrent imperative programs.[4] As such, one of the main assertions in OCPL is the *Hoare triple*, which is used to specify the behavior of expressions in terms of preconditions and postconditions. The Hoare triple $\{P\}\ e\ \{x.\ Q\}_p$ asserts that, assuming *e* is executed starting in a state satisfying the precondition *P*, then it will execute without any dynamically failing `assert` expressions, and if it terminates with value *v*, then the final state will satisfy the postcondition *Q* (which may mention the bound variable *x*). The *progress bit*

[3]The expression CAS $\ell\ v_1\ v_2$ atomically compares the contents of location ℓ to value v_1 and, if equal, writes v_2 to ℓ and returns `true`; otherwise, it returns `false`. For simplicity, we do not restrict CAS to comparing first-order, "word-sized" values. This is unrealistic, but we use CAS in only realistic ways. (Spinlocks, in particular, are compatible with such a restriction and suffice for our purposes.)

[4]Readers unfamiliar with separation logic can, to a first approximation, read separating conjunction $P * Q$ as conjunction $P \wedge Q$ and magic wand $P \mathrel{-\!\!*} Q$ as implication $P \Rightarrow Q$.

PointstoExclusive
$\ell \hookrightarrow v * \ell \hookrightarrow w \vdash \bot$

Alloc
$\{\top\}\ \mathtt{ref}\ v\ \{\ell, \mathsf{ret}\ \ell.\ \ell \hookrightarrow v\}$

Load
$\{\ell \hookrightarrow v\}\ !\ell\ \{\mathsf{ret}\ v.\ \ell \hookrightarrow v\}$

Store
$\{\ell \hookrightarrow v\}\ \ell \leftarrow w\ \{\mathsf{ret}\ ().\ \ell \hookrightarrow w\}$

(a) High locations

HighNotLow
$\ell \hookrightarrow v * \mathsf{lowloc}\,\ell \vdash \bot$

AllocLow
$\{\mathsf{lowval}\,v\}\ \mathtt{ref}\ v\ \{\ell, \mathsf{ret}\ \ell.\ \mathsf{lowloc}\,\ell\}$

LoadLow
$\{\mathsf{lowloc}\,\ell\}\ !\ell\ \{x.\ \mathsf{lowval}\,x\}$

StoreLow
$\{\mathsf{lowloc}\,\ell * \mathsf{lowval}\,v\}\ \ell \leftarrow v\ \{\mathsf{ret}\ ().\ \top\}$

(b) Low locations

Fig. 2. Selected proof rules for locations

$p \in \{\mathtt{progress}, \mathtt{noprogress}\}$ indicates whether the triple is *progressive* (*i.e.,* ensures that e does not get stuck) or *non-progressive.*[5] As shorthand, we will write $\{P\}\ e\ \{x.\ Q\}$ and $\{P\}\ e\ \{x.\ Q\}_?$ for progressive and non-progressive triples, respectively; the former assertion implies the latter. We often write the postcondition of a Hoare triple as $\{x_1 \cdots x_n, \mathsf{ret}\ pat.\ Q\}$ which binds the x_i and specifies that any returned value will be precisely pat. This notation is (roughly) syntactic sugar for the post-condition $\{x.\ \exists x_1, \ldots, x_n.\ x = pat * Q\}$.

Assertion expressions. OCPL supports the following rule for assertion expressions.

Assert
$$\{P\}\ e\ \{x.\ x = \mathtt{true} * Q\}_p \vdash \{P\}\ \mathtt{assert}\ e\ \{\mathsf{ret}\ ().\ Q\}_p$$

Assert enables reasoning about successful assertions. It says that, to prove a Hoare triple for `assert` e with postcondition Q and return value (), it suffices to prove that e evaluates to `true` with postcondition Q.

High vs. low locations. OCPL divides memory locations (*i.e.,* mutable references) into two types: *high-integrity* and *low-integrity* (or just *high* and *low* for short). High locations are locations that are private to user code, on which it may place invariants of its choosing, and to which untrusted code should not be given direct, unfettered access. Low locations are locations that may be freely shared with—and may in fact have been allocated by—untrusted code. Note that there is no distinction between high and low locations in the operational semantics of HLA; rather, this distinction is merely something we track in OCPL in order to formally specify the interface between user code and untrusted code.

Reasoning about high locations. Since high locations are locations controlled privately by user code, OCPL supports reasoning about them in essentially the same way as one reasons about pointers in traditional separation logic—via the classic "points-to" assertion $\ell \hookrightarrow v$. This points-to assertion denotes the knowledge that location ℓ currently points to value v as well as ownership of ℓ. What this means is that if we own ℓ, then no other party (*i.e.,* no other thread) can be simultaneously making any assertion about ℓ, so we should have permission to freely read from and write to ℓ without worrying about violating any other party's reasoning. This reasoning is formalized with standard rules about high locations displayed in Fig. 2a.

Reasoning about low locations. We represent the knowledge that a location ℓ is low via the assertion $\mathsf{lowloc}\,\ell$. In contrast to the points-to assertion for high locations, the assertion $\mathsf{lowloc}\,\ell$ does not denote any ownership of ℓ because as soon as a location is considered safe to be shared with untrusted code, there is no way of knowing what that code will do with it. Rather, $\mathsf{lowloc}\,\ell$ merely tells us that ℓ is low now and will remain low forever.

[5]Note: this is a point of difference from the original Iris, which only supported progressive triples.

$$\mathsf{lift}\,\Psi\,(\mathtt{rec}\,f\,x.\,e) \triangleq \triangleright\forall v.\ \{\mathsf{lift}\,\Psi\,v\}\ e[v/x, \mathtt{rec}\,f\,x.\,e/f]\ \{y.\ \mathsf{lift}\,\Psi\,y\}_{?} \qquad (\textsc{LiftRec})$$

$$\mathsf{lift}\,\Psi\,\ell \triangleq \Psi\,\ell \qquad (\textsc{LiftLoc})$$

$$\mathsf{lift}\,\Psi\,\mathit{lit} \triangleq \top \qquad (\textsc{LiftLit})$$

$$\mathsf{lift}\,\Psi\,() \triangleq \top \qquad (\textsc{LiftUnit})$$

$$\mathsf{lift}\,\Psi\,(v_1, v_2) \triangleq \triangleright(\mathsf{lift}\,\Psi\,v_1 * \mathsf{lift}\,\Psi\,v_2) \qquad (\textsc{LiftPair})$$

$$\mathsf{lift}\,\Psi\,(\mathtt{inl}\,v) \triangleq \triangleright\mathsf{lift}\,\Psi\,v \qquad (\textsc{LiftInl})$$

$$\mathsf{lift}\,\Psi\,(\mathtt{inr}\,v) \triangleq \triangleright\mathsf{lift}\,\Psi\,v \qquad (\textsc{LiftInr})$$

Fig. 3. Lifting location predicates to value predicates

Moreover, if ℓ is low—and hence safely shareable with untrusted code—then we know that the value v it points to must also be safely shareable with untrusted code—*i.e.,* v is a *low value.*

Of course, this begs the question: what is a low value? Intuitively, a low value is a value from which there is no way to extract a high location. We will make this more precise in a moment. For now, assume we have a predicate $\mathsf{lowval}\,v$, which says that v is a low value.

The description of low locations given above is formalized in the proof rules of Fig. 2b. Rule HighNotLow says that a location cannot be high and low at the same time. Rules AllocLow, LoadLow, and StoreLow mimic the corresponding rules for allocation, reading, and writing of high locations, except that for low locations, we do not track the precise contents of the location ℓ—we merely insist that ℓ always points to a low value.

Lifting low locations to low values. We return now to the question of what it means for a *value* v to be low. Intuitively, a low value is one from which the language constructs of HLA provide no way to get direct access to any high location. This is fundamentally an *extensional* property, *i.e.,* a property about the observations that a program can make (*i.e.,* the information it can extract) when passed the value v. As such, for those readers familiar with classic techniques from program semantics, it will not come as a surprise that a natural way of formally accounting for this property is via a *logical relation.*

Fig. 3 shows the definition of this logical relation, presented in a somewhat more general form, $\mathsf{lift}\,\Psi\,v$. Essentially, lift takes as input Ψ, a predicate on locations, and lifts it to a predicate on values, with the property that $\mathsf{lift}\,\Psi\,v$ is true if Ψ holds for any location that can be extracted from v. Given this definition, $\mathsf{lowval}\,v$ can be defined simply as $\mathsf{lift}\,\mathsf{lowloc}\,v$—*i.e.,* a value is low if any location that can be extracted from it is low.

On literals and unit, $\mathsf{lift}\,\Psi$ is trivially true since no location can be extracted from them; on locations, $\mathsf{lift}\,\Psi$ is simply Ψ; and on products and sums, $\mathsf{lift}\,\Psi$ is defined in the obvious recursive manner. The only interesting case is the one for functions. For a function value f, we want to say that $\mathsf{lift}\,\Psi\,f$ if, whenever we apply f, the resulting term only ever produces values that satisfy $\mathsf{lift}\,\Psi$. (This property on applications of f is expressed via a *non-progressive* Hoare triple since f may be applied to arguments constructed by untrusted code.) But what arguments should we consider when applying f? This is where the logical relation comes in: we need only consider argument values that themselves satisfy $\mathsf{lift}\,\Psi$, because ultimately we will make sure that all values passed to untrusted code satisfy $\mathsf{lift}\,\Psi$.

Note that the definition of $\mathsf{lift}\,\Psi$ in the function case is rather circular, since it quantifies over values that satisfy $\mathsf{lift}\,\Psi$. To ensure that this definition is well-founded, we rely on OCPL's support for guarded recursive definitions (inherited from Iris). In OCPL, a definition may be arbitrarily recursive so long as any recursive references are guarded by an occurrence of the $\triangleright$ ("later") modality. Under the hood, all assertions in OCPL are implicitly indexed by steps of computation—*i.e.,* the model of OCPL is a kind of step-indexed model (Birkedal et al. 2011)—and $\triangleright P$ means that P holds, roughly speaking, "after one step of computation". Since the fine points of the "later" modality are not particularly relevant to our main contribution, and our use of "later" is standard, we refer the reader to prior work for a more detailed presentation (Appel et al. 2007).

To make concrete this rather abstract tour of OCPL, we return to our motivating example. To verify that *readonly* satisfies it specification, we have to show that the function it returns, $\lambda\_.\ !\ell$, is a low value given a triple $\{\top\}\ !\ell\ \{x.\ \mathsf{lowval}\,x\}$ for dereferencing ℓ. This follows easily by LiftRec.

To verify that the expression *usetwo* returns a low value, we proceed as follows. From the allocation `ref` 2 we obtain $\ell \hookrightarrow 2$ for some location ℓ (by Alloc). Since we own ℓ, we are free to transfer ownership of ℓ into a shared invariant, which in Iris (and OCPL) is written $\boxed{\ell \hookrightarrow 2}$, stipulating that ℓ will *always* contain 2. (OCPL inherits Iris's support for dynamically allocated invariants. We omit rules for working with them here since they are orthogonal to the focus of this paper.) By Load, we easily obtain a lemma

$$\boxed{\ell \hookrightarrow 2} \vdash \{\top\}\ !\ell\ \{\mathsf{ret}\ 2.\ \top\} \qquad (\dagger)$$

stating that, under our invariant, dereferencing ℓ always returns 2. To prove that the value w returned by *readonly* ℓ is low-integrity, we apply *readonly*'s specification and have to show that dereferencing ℓ returns a low value (which is trivial, by (†) and LiftLit). To prove that the function $use \triangleq \lambda\_.\ \mathtt{assert}\ ((!\ell) = 2)$ is low-integrity, we apply LiftRec. In this subproof, we have to prove that, under our invariant, the assertion returns a low value (trivial by Assert, (†), LiftUnit). It remains to show that the pair (w, use) is low-integrity given knowledge that both w and *use* are low-integrity, which is trivial by LiftPair.

2.3 Metatheory

OCPL inherits (omitted) adequacy theorems for reasoning about return values and progressive triples from Iris. The metatheorems that concern us deal with dynamically checked safety properties.

Theorem AdequacySafety**.** If an expression is verified and run in a good state, we can observe that every reachable state is good (*i.e.,* no assertions fail):

$$\frac{\{\top\}\ e\ \{x.\ \top\}_p \qquad (e); (h, \mathsf{OK}) \longrightarrow^* T'; (h', g')}{g' = \mathsf{OK}}$$

Theorem RobustSafety**.** Let *AdvCtx* denote the set of HLA contexts containing neither locations nor assertions. If expression e is closed and has been verified to return only low values, then for every adversarial context C, on running $C[e]$ from an initial state, we can observe that every reachable state is good (*i.e.,* no assertions fail):

$$\frac{C \in AdvCtx \qquad e\ \mathrm{closed} \qquad \{\top\}\ e\ \{x.\ \mathsf{lowval}\,x\}_p \qquad (C[e]); (\emptyset, \mathsf{OK}) \longrightarrow^* T'; (h', g')}{g' = \mathsf{OK}}$$

3 DYNAMIC SEALING

We now consider one of the oldest and most influential OCPs: *dynamic sealing*, also called the sealer-unsealer pattern. Originally proposed by Morris (1973), dynamic sealing makes it possible to encode something like an abstract data type (ADT) even in the absence of static typing. In this section, we show how OCPL supports compositional reasoning about dynamic sealing. In particular, we show how to implement dynamic sealing in HLA and how to give a compositional specification for this implementation, from which we derive useful specifications for interesting abstractions built on top of it and prove robust safety for representative clients. (We consider an alternative implementation of the OCP, satisfying a slightly weaker specification, in Appendix A. We also show how ideal specifications for cryptographic signing and encryptions primitives can be derived from the specification of dynamic sealing in Appendix B.)

The functionality of dynamic sealing. Morris (1973) introduced dynamic sealing to enforce data abstraction while interoperating with untrusted, potentially ill-typed code. He stipulated a function *makeseal* for generating pairs of functions (*seal*, *unseal*), such that (i) for every value v, $\mathit{seal}\,v$ returns a value v' serving as an *opaque, low-integrity proxy* for v; and (ii) for every value v', $\mathit{unseal}\,v'$ returns v, if v' was produced by $\mathit{seal}\,v$, and otherwise gets stuck. The key point is that this *seal-unseal* pair supports data abstraction: the client of these functions can freely pass sealed values to untrusted code since they are low-integrity, while at the same time imposing whatever internal invariant it wants on the underlying values that they represent.

To see how this is useful, consider a simple client of dynamic sealing, namely Morris's example of a library for integer intervals:

$$
\begin{aligned}
\mathit{intervals} = \lambda\_.\ & \texttt{let}\ (\mathit{seal}, \mathit{unseal}) = \mathit{makeseal}\,()\ \texttt{in} \\
& \texttt{let}\ \mathit{makeint} = \lambda n_1\, n_2.\ \mathit{seal}\,(\texttt{if}\ n_1 \le n_2\ \texttt{then}\ (n_1, n_2)\ \texttt{else}\ (n_2, n_1))\ \texttt{in} \\
& \texttt{let}\ \mathit{imin} = \lambda i.\ \texttt{fst}\,(\mathit{unseal}\,i)\ \texttt{in} \\
& \texttt{let}\ \mathit{imax} = \lambda i.\ \texttt{snd}\,(\mathit{unseal}\,i)\ \texttt{in} \\
& \texttt{let}\ \mathit{isum} =\ \lambda i.\ \texttt{let}\ x = \mathit{unseal}\,i\ \texttt{in}\ \lambda j.\ \texttt{let}\ y = \mathit{unseal}\,j\ \texttt{in} \\
& \qquad\qquad\ \mathit{seal}\,(\texttt{fst}\ x + \texttt{fst}\ y, \texttt{snd}\ x + \texttt{snd}\ y) \\
& \texttt{in}\ (\mathit{makeint}, \mathit{imin}, \mathit{imax}, \mathit{isum})
\end{aligned}
$$

Let $[n_1, n_2]$ denote the set $\{n_1, n_1 + 1, \ldots, n_2 - 1, n_2\}$. The function *intervals* returns several interval-manipulating routines: $\mathit{makeint}\,n_1\,n_2$ constructs $[\min n_1\, n_2, \max n_1\, n_2]$, $\mathit{imin}\,[n_1, n_2]$ returns n_1, $\mathit{imax}\,[n_1, n_2]$ returns n_2, and $\mathit{isum}\,[n_1, n_2]\,[n_1', n_2']$ returns $[n_1 + n_1', n_2 + n_2']$.

These routines use dynamic sealing to enforce the internal data representation of intervals, namely that the interval $[n_1, n_2]$ is represented by the pair (n_1, n_2), which is critical to ensuring correctness of the library. In particular, notice that *seal* and *unseal* are kept private to the *intervals* implementation, which means it can enforce that the only values sealed with *seal* are pairs (n_1, n_2) representing intervals (*i.e.,* where $n_1 \le n_2$). Consequently, the *imin* (resp. *imax*) function can simply return the first (resp. second) component of its argument after unsealing it, because it *knows* that, even if the argument comes from untrusted code, *so long as the unsealing succeeds*, the resulting value will be a pair where the first (resp. second) component represents the lower (resp. upper) bound of the input interval. Furthermore, a client of *intervals* knows that if it applies both *imin* and *imax* to an arbitrary untrusted value v, and both operations succeed, producing values v_1 and v_2, then v must indeed represent a proper interval $[n_1, n_2]$ (with $n_1 \le n_2$), and the resulting values of *imin* and *imax* are precisely v's lower and upper bounds (*i.e.,* with $v_1 = n_1$ and $v_2 = n_2$). In this way, dynamic sealing affords programmers the essential functionality of data abstraction, even when interfacing with untrusted code, at the cost of some simple dynamic checks at the boundaries of the abstraction.

Implementation of dynamic sealing. We implement dynamic sealing in HLA as follows.[6]

$$
\begin{aligned}
\mathit{makeseal} \triangleq \lambda\_.\ & \texttt{let}\ \mathit{tbl} = \texttt{ref}\ \mathit{mapempty}\ \texttt{in} \\
& \texttt{let}\ \mathit{sync} = \mathit{makesync}\,()\ \texttt{in} \\
& \texttt{let}\ \mathit{seal} = \lambda x.\ \texttt{let}\ k = \texttt{ref}\,()\ \texttt{in}\ \mathit{sync}(\lambda\_.\ \mathit{tbl} \leftarrow \mathit{mapinsertnew}\,(!\,\mathit{tbl})\,k\,x); k\ \texttt{in} \\
& \texttt{let}\ \mathit{unseal} = \lambda k.\ \texttt{assume}\,(\texttt{isloc}\ k); \mathit{sync}(\lambda\_.\ \mathit{maplookup}\,(!\,\mathit{tbl})\,k)\ \texttt{in}
\end{aligned}
$$

[6]We discuss the straightforward functions *makesync*, *mapempty*, *mapinsertnew*, and *maplookup* in Appendix F. Note that $\mathit{maplookup}\,f\;k$ gets stuck if location k is not in the domain of the partial function represented by value f.

Data abstraction.

MakeSealSpec
$$\{\top\}\ \mathit{makeseal}\,()\ \{v_1\, v_2\, \gamma, \mathsf{ret}\ (v_1, v_2).\ \mathsf{isseal}\,\gamma\, v_1\, \phi * \mathsf{isunseal}\,\gamma\, v_2\, \phi\}$$

SealSpec
$$\{\mathsf{isseal}\,\gamma\, s\, \phi * \phi\, v\}\ s\, v\ \{x'.\ \mathsf{issealed}\,\gamma\, v\, x'\, \phi\}$$

UnsealSpec
$$\{\mathsf{isunseal}\,\gamma\, u\, \phi * \mathsf{issealed}\,\gamma\, v\, v'\, \phi\}\ u\, v'\ \{\mathsf{ret}\ v.\ \top\}$$

UnsealAnySpec
$$\{\mathsf{isunseal}\,\gamma\, u\, \phi\}\ u\, v'\ \{x.\ \mathsf{issealed}\,\gamma\, x\, v'\, \phi\}_?$$

SealedInv
$$\mathsf{issealed}\,\gamma\, v\, v'\, \phi \vdash \phi\, v$$

SealedAgree
$$\mathsf{issealed}\,\gamma\, v_1\, v'\, \phi * \mathsf{issealed}\,\gamma\, v_2\, v'\, \phi \vdash v_1 = v_2$$

Low values.

SealedLow
$$\mathsf{issealed}\,\gamma\, v\, v'\, \phi \vdash \mathsf{lowval}\, v'$$

SealLow
$$\frac{(\forall v)\ \mathsf{lowval}\, v \vdash \phi\, v}{\mathsf{isseal}\,\gamma\, s\, \phi \vdash \mathsf{lowval}\, s}$$

UnsealLow
$$\frac{(\forall v)\ \phi\, v \vdash \mathsf{lowval}\, v}{\mathsf{isunseal}\,\gamma\, u\, \phi \vdash \mathsf{lowval}\, u}$$

Fig. 4. Sealing interface (presupposing ϕ persistent)

(*seal*, *unseal*)

The function *makeseal* allocates a fresh lookup table *tbl*, which is used to store the mapping between sealed values and their proxies, and access to which is shared by the *seal* and *unseal* functions that *makeseal* returns. To ensure proper synchronization between calls to *seal* and *unseal*, *makeseal* also creates a lock via the call to *makesync*. (The latter returns a higher-order function *sync*, which ensures mutual exclusion among any expressions e_i run under $\mathit{sync}\,(\lambda\_.\ e_i)$. For more details on *makesync*, see Appendix F.) An entry $[k \mapsto v]$ in *tbl* signifies that location k is a low-integrity proxy for sealed value v. To seal a value v, we allocate a fresh, low location k, extend the table, and return k. To unseal a value v', we require that v' be a location k and then look up k in the table.

To test whether the argument to *unseal* is a location, we use a primitive unary operator `isloc` (with boolean return type), as well as an *assumption expression*. The expression `assume` e resembles a C- or Java-style assertion: it returns unit if e evaluates to `true`; otherwise, it gets stuck. It is easy to define assumption expressions using a stuck term `abort`:

$$\mathtt{abort} \triangleq 0\,0 \qquad\qquad \mathtt{assume}\ e \triangleq \mathtt{if}\ e\ \mathtt{then}\ ()\ \mathtt{else\ abort}$$

These satisfy the following specifications.

Abort
$$\{\top\}\ \mathtt{abort}\ \{x.\ Q\}_?$$

Assume
$$\{P\}\ e\ \{x.\ x = \mathtt{true} \twoheadrightarrow Q\}_? \vdash \{P\}\ \mathtt{assume}\ e\ \{\mathsf{ret}\ ().\ Q\}_?$$

Abort gives `abort` a non-progressive triple because in fact `abort` always gets stuck and never produces a value; correspondingly, it can also be given an *arbitrary* postcondition Q. Compared to Assert on page 6, Assume (i) uses non-progressive triples since `assume` e may get stuck, and (ii) only requires the postcondition Q to be shown to hold under the *assumption* that e returns `true`.

Specifying dynamic sealing. The motivation for dynamic sealing given above was completely informal. Let us now see how OCPL lets us formalize it.

Properties.

MinSpec
$$\{\mathsf{ismin}\,\gamma\,\mathit{imin} * \mathsf{isinterval}\,\gamma\,n_1\,n_2\,i\}\ \mathit{imin}\,i\ \{\mathsf{ret}\ n_1.\ \top\}$$

IntervalInv
$$\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i \vdash n_1 \le n_2$$

IntervalAgree
$$\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i * \mathsf{isinterval}\,\gamma\,n_1'\,n_2'\,i \vdash n_1 = n_1' * n_2 = n_2'$$

Non-progressive triples.

MinAnySpec
$$\{\mathsf{ismin}\,\gamma\,\mathit{imin}\}\ \mathit{imin}\,v\ \{n_1\,n_2, \mathsf{ret}\ n_1.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v\}_?$$

MaxAnySpec
$$\{\mathsf{ismax}\,\gamma\,\mathit{imax}\}\ \mathit{imax}\,v\ \{n_1\,n_2, \mathsf{ret}\ n_2.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v\}_?$$

Low values.

IntervalLow
$$\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i \vdash \mathsf{lowval}\,i$$

MinLow
$$\mathsf{ismin}\,\gamma\,\mathit{imin} \vdash \mathsf{lowval}\,\mathit{imin}$$

MaxLow
$$\mathsf{ismax}\,\gamma\,\mathit{imax} \vdash \mathsf{lowval}\,\mathit{imax}$$

Fig. 5. Intervals interface (selected rules—see Appendix C)

We specify dynamic sealing in Fig. 4. Rule MakeSealSpec for allocating a sealer-unsealer pair ties the two functions it returns to an abstract (logical) name γ and representation invariant ϕ. In particular, the assertions[7] $\mathsf{isseal}\,\gamma\,v_1\,\phi$ and $\mathsf{isunseal}\,\gamma\,v_2\,\phi$ represent knowledge that v_1 and v_2 are the seal and unseal functions associated with the sealer-unsealer pair named γ with representation invariant ϕ. Similarly, the assertion $\mathsf{issealed}\,\gamma\,v\,v'\,\phi$ represents knowledge that v' is a low-integrity proxy for value v (for the sealer-unsealer pair named γ with representation invariant ϕ). This relation is functional in v'—*i.e.,* unsealing the same v' twice will produce the same v (SealedAgree)—and implies $\phi\,v$—*i.e.,* that the unsealed value v must satisfy the representation invariant (SealedInv). Conversely, to seal a value v, one is required to *prove* $\phi\,v$ (SealSpec). The progressive triple UnsealSpec is suitable for use when one already knows that v and v' are related; it returns v. The non-progressive triple UnsealAnySpec returns *the* value v related to v', or gets stuck.

Finally, the three rules at the bottom of Fig. 4 concern the lowval relation. Rule SealedLow says that all sealed values are low values—indeed, the whole point of using sealing is to produce low-integrity proxy values—while rules SealLow and UnsealLow say when the seal and unseal functions themselves can be considered low values. Intuitively, seal and unseal can be considered low values whenever doing so does not either (i) violate the representation invariant ϕ or (ii) result in high-integrity values flowing to a low context. Thus, seal is low so long as any value that is passed to it (*i.e.,* any low value) satisfies ϕ, and unseal is low so long as any value returned by it (*i.e.,* any value that satisfies ϕ) is low. (We show how to gainfully employ low seal and unseal functions in Appendix B.)

The specification for dynamic sealing from Fig. 4 can be straightforwardly used to derive an analogous specification for the interval-manipulation routines generated by *intervals*, as we described above. In particular, by choosing the representation invariant $\phi\,v \triangleq \exists n_1, n_2.\ v = (n_1, n_2) * n_1 \le n_2$, we can easily verify the rules shown in Fig. 5 (the full specification for *intervals* is given in Appendix C). The assertion $\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i$ represents knowledge that value i represents interval $[n_1, n_2]$ in the instance of the intervals library with abstract

[7]In our specifications, predicates whose names start with "is" produce, when fully applied, *persistent* propositions (and are themselves called persistent predicates). Persistent propositions may be regarded as intuitionistic (technically, they may be freely duplicated). A sealer-unsealer pair's representation invariant ϕ must be persistent.

name γ. A value may represent at most one interval (IntervalAgree). A set of progressive triples (*e.g.,* MinSpec) account for our informal specification and are suited to reasoning about values known to represent intervals. A set of non-progressive triples are suited to reasoning about values that *might* represent intervals. Rule MinAnySpec, for example, says that for any value v, $imin\, v$ might get stuck, but if it terminates with value $\hat{v}$, then v represents some interval with minimum n_1 and $\hat{v} = n_1$. Lastly, there are rules (*e.g.,* IntervalLow and MinLow) expressing the fact that values representing intervals, and the interval routines themselves, may be safely shared with untrusted code. Crucial to the soundness of these rules is the fact that the seal function used internally by *intervals* is *not* shared with untrusted code (*i.e.,* not low): according to rule SealLow, in order for seal to be treated as low, the representation invariant ϕ would have to be satisfied by all low values, which it clearly is not.

With the spec for *intervals* in hand, we can readily prove that the following simple but illustrative client of *intervals* is robustly safe.

$$
\begin{aligned}
client \triangleq\ & \mathtt{let}\ cap = intervals\,()\ \mathtt{in}\\
& \mathtt{let}\ (makeint, imin, imax, isum) = cap\ \mathtt{in}\\
& \mathtt{let}\ check = \lambda j.\ \mathtt{assert}\ (imin\, j \le imax\, j)\ \mathtt{in}\\
& (check, cap)
\end{aligned}
$$

The expression constructs an instance *cap* of the intervals library, along with a function *check* which takes an arbitrary value j as its argument and asserts that $imin\, j$ is no greater than $imax\, j$. Intuitively, even if *client* is shared with untrusted code, this assertion must always succeed, because if the applications of *imin* and *imax* do not get stuck, it means that j is a proper interval value, whose lower bound is $\le$ its upper bound.

Formally, this is guaranteed as follows. First, using the spec for *intervals*, we can prove $\{\top\}\ client\ \{x.\ \mathsf{lowval}\, x\}$. The key step in the proof involves showing that the assertion in *check* succeeds for arbitrary j, which follows directly from MinAnySpec, MaxAnySpec, IntervalAgree, and IntervalInv. Second, we appeal to Theorem RobustSafety from §2.3, which implies that the assertion in *check* will not fail, even when *client* is linked with untrusted code.

4 CARETAKER

Next, we consider another well-known OCP, the *caretaker pattern* (Miller and Shapiro 2003; Miller 2006). This OCP allows allows verified (trustworthy) code to grant untrusted code access to a high-integrity resource (a high-integrity location or an API that modifies high-integrity locations), and subsequently disable or enable the access at any time. When the caretaker is enabled, untrusted code can access the resource; when the caretaker is disabled, untrusted code cannot access the resource and the verified code has full control over the resource. The caretaker pattern is useful when the verified code wants to ensure that the untrusted code can access the resource only while some invariant holds. Disabling the caretaker allows the verified code to temporarily break the invariant, secure in the knowledge that untrusted code won't be able to access the resource until the caretaker is re-enabled.

In this section, we first implement a *caretaker for APIs* in HLA, and specify and verify it in OCPL. Then, we use the API caretaker to implement a second *caretaker for locations* and, again, specify and verify it. Finally, we present a simple client for the location caretaker and establish that it is robustly safe.

API caretaker. The API caretaker allows verified code to share revokable access to any set of functions, which may have side-effects.

$$
\begin{aligned}
makecaretaker &\triangleq \lambda\_.\ \mathtt{let}\ enabled = \mathtt{ref\ false\ in\ let}\ sync = makesync()\ \mathtt{in}\ (sync, enabled)\\
wrap &\triangleq \lambda(sync, enabled)\, f\, x.\ sync\,(\lambda\_.\ \mathtt{assume}\ (!\,enabled); f\, x)\\
enable &\triangleq \lambda(sync, enabled).\ sync\,(\lambda\_.\ enabled \leftarrow \mathtt{true})
\end{aligned}
$$

EnabledExclusive

$\mathsf{enabled}\,\gamma\,b_1 * \mathsf{enabled}\,\gamma\,b_2 \vdash \bot$

MakeCaretakerSpec

$\{\top\}\ \mathit{makecaretaker}\,()\ \{ct\,\gamma, \mathsf{ret}\ ct.\ \mathsf{iscaretaker}\,\gamma\ ct\,R * \mathsf{enabled}\,\gamma\,\mathtt{false}\}$

CanWrap

$\mathsf{canwrap}\,f\,R \triangleq \forall v.\ \{\mathsf{lowval}\,v * R\}\ f\ v\ \{x'.\ \mathsf{lowval}\,x' * R\}_?$

WrapSpec

$\{\mathsf{iscaretaker}\,\gamma\ ct\,R * \mathsf{canwrap}\,f\,R\}\ \mathit{wrap}\,ct\,f\ \{f'.\ \mathsf{lowval}\,f'\}$

EnableSpec

$\{\mathsf{iscaretaker}\,\gamma\ ct\,R * \mathsf{enabled}\,\gamma\,\mathtt{false} * R\}\ \mathit{enable}\,ct\ \{\mathsf{ret}\ ().\ \mathsf{enabled}\,\gamma\,\mathtt{true}\}$

DisableSpec

$\{\mathsf{iscaretaker}\,\gamma\ ct\,R * \mathsf{enabled}\,\gamma\,\mathtt{true}\}\ \mathit{disable}\,ct\ \{\mathsf{ret}\ ().\ \mathsf{enabled}\,\gamma\,\mathtt{false} * R\}$

Fig. 6. API caretaker interface

$$\mathit{disable} \triangleq \lambda(\mathit{sync}, \mathit{enabled}).\ \mathit{sync}\,(\lambda\_.\ \mathit{enabled} \leftarrow \mathtt{false})$$

The function *makecaretaker* returns a new caretaker *ct*, which comprises a fresh lock *sync*, whose purpose is described soon, and a fresh boolean reference *enabled*. The caretaker is disabled (enabled) when *enabled* is `false` (`true`). The functions *enable ct* and *disable ct* enable and disable the caretaker *ct* by setting *enabled* appropriately. The function *wrap ct f* wraps the function *f* in the caretaker *ct*, returning a function that behaves exactly like *f* when *ct* is enabled, and gets stuck when *ct* is disabled. Additionally, accesses to all functions wrapped in *ct* are serialized using the lock *sync*.

To use this interface, verified code creates a caretaker *ct* and holds it privately. It can then wrap any number of API functions using *wrap* and disclose the wrapped functions to untrusted code. The untrusted code's access to all those functions can be simultaneously disabled and enabled by calling *disable ct* and *enable ct*, respectively.

API caretaker specification. The goal of using the API caretaker is to ensure that untrusted code can access wrapped functions only while some logical resource or invariant R holds on the heap. Our specification of the API caretaker in Fig. 6 formalizes this intuition. The assertion $\mathsf{iscaretaker}\,\gamma\ ct\,R$ represents knowledge that value *ct* is a caretaker with (logical) name γ whose wrappers allow (untrusted code) access to wrapped functions only when R holds. The assertion $\mathsf{enabled}\,\gamma\,b$ represents exclusive ownership of the (verified code's) right to enable and disable all wrappers created using the caretaker named γ as well as knowledge that the caretaker is currently enabled (if $b = \mathtt{true}$) or disabled (otherwise).

When applying MakeCaretakerSpec, one may pick an arbitrary invariant R. The spec returns a disabled caretaker tied to R and a fresh name γ. To enable *ct*, the (verified) code calling *enable* must establish that R holds (rule EnableSpec). Dually, rule DisableSpec says that disabling *ct* provides ownership of R to the caller (verified code).

Rule WrapSpec is at the heart of our specification. It says that to wrap a function f using *ct*, one has to prove that f, when applied to any low value v (possibly provided by the untrusted code), *preserves* the invariant R and returns a low value. So, any wrapped function f can assume R in its precondition but it must re-establish R if it terminates (note that the triple for $f\ v$ in rule CanWrap is non-progressive). This ensures that while *ct* is enabled, calls to the wrapped functions by the untrusted code preserve R. While *ct* is disabled, verified code can change the state to break R but, prior to re-enabling *ct*, it must re-establish R.

The implementation of the API caretaker verifies against this specification. We note that the specification and verification of the API caretaker closely resemble similar efforts for specifying and verifying locks in concurrent programs (Dinsdale-Young et al. 2010). Based on this observation, we offer a shorter implementation of the API

$$\mathsf{enabled}\,\gamma\,b_1 * \mathsf{enabled}\,\gamma\,b_2 \vdash \bot$$

IsRmon

$$\mathsf{isrmon}\,f\,\Psi \triangleq \forall v.\ \{\Psi\,v\}\ f\ v\ \{x'.\ \Psi\,v * \mathsf{lowval}\,x'\}_?$$

IsWmon

$$\mathsf{iswmon}\,f\,\Psi \triangleq \forall v.\ \{\mathsf{lowval}\,v\}\ f\ v\ \{x'.\ \Psi\,x'\}_?$$

MakeLocCtSpec

$$\{\mathsf{isrmon}\,f_\mathsf{r}\,\Psi * \mathsf{iswmon}\,f_\mathsf{w}\,\Psi\}\ \mathit{makeloccct}\ f_\mathsf{r}\ f_\mathsf{w}\ \ell\ \{\mathit{ct}\,\gamma\,v, \mathsf{ret}\ (\mathit{ct}, v).\ \mathsf{isloccct}\,\gamma\,\mathit{ct}\,\ell\,\Psi * \mathsf{enabled}\,\gamma\,\mathtt{false} * \mathsf{lowval}\,v\}$$

LocCtEnableSpec

$$\{\mathsf{isloccct}\,\gamma\,\mathit{ct}\,\ell\,\Psi * \mathsf{enabled}\,\gamma\,\mathtt{false} * \ell \hookrightarrow v * \Psi\,v\}\ \mathit{enable}\,\mathit{ct}\ \{\mathsf{ret}\ ().\ \mathsf{enabled}\,\gamma\,\mathtt{true}\}$$

LocCtDisableSpec

$$\{\mathsf{isloccct}\,\gamma\,\mathit{ct}\,\ell\,\Psi * \mathsf{enabled}\,\gamma\,\mathtt{true}\}\ \mathit{disable}\,\mathit{ct}\ \{v, \mathsf{ret}\ ().\ \mathsf{enabled}\,\gamma\,\mathtt{false} * \ell \hookrightarrow v * \Psi\,v\}$$

Fig. 7. Location caretaker interface

caretaker in Appendix D. This implementation uses lock release and acquire instead of the boolean reference *enabled* to enable and disable the caretaker.

The API caretaker is a very general OCP. It can be used to build caretakers for other kinds of resources. As an instance, we show next how to build a caretaker for locations using the API caretaker.

Location caretaker. The location caretaker, defined by the function *makeloccct* below, facilitates *revokable and mediated* read and write access to a location. Specifically, it ensures that the location is accessed by untrusted code only while some stipulated predicate Ψ holds on the location's content.

$$\begin{aligned}\mathit{makeloccct} \triangleq \lambda \mathit{rmon}\,\mathit{wmon}\,r.\ &\mathtt{let}\ \mathit{ct} = \mathit{makecaretaker}\,()\ \mathtt{in}\\ &\mathtt{let}\ \mathit{read} = \mathit{wrap}\,\mathit{ct}\,(\lambda\_.\ \mathit{rmon}\,(!r))\ \mathtt{in}\\ &\mathtt{let}\ \mathit{write} = \mathit{wrap}\,\mathit{ct}\,(\lambda x.\ r \leftarrow (\mathit{wmon}\,x)\ \mathtt{in}\\ &(\mathit{ct}, (\mathit{read}, \mathit{write}))\end{aligned}$$

The function *makeloccct*, when applied to a function f_r, a function f_w, and a location ℓ, returns a pair $(\mathit{ct}, (\mathit{read}, \mathit{write}))$. The API caretaker *ct* controls whether or not the functions $\{\mathit{read}, \mathit{write}\}$ are enabled. When *ct* is enabled, the function *read* reads ℓ and filters the read value through f_r. Similarly, *write* writes ℓ after filtering the value to be written through f_w. Thus, f_r and f_w mediate reads from and writes to ℓ. Consider, for example, an application $\mathit{write}\,v$, representing a request to write value v to location ℓ. If *ct* is enabled, the application reduces to $\ell \leftarrow (f_\mathsf{w}\,v)$ and f_w gets to decide what is written.

To use the location caretaker on a location ℓ, verified code invokes $\mathit{makeloccct}\ f_\mathsf{r}\ f_\mathsf{w}\ \ell$ with appropriate f_r and f_w. It holds ℓ and the returned API caretaker *ct* private, but passes *read* and *write* to untrusted code. It can then enable and disable access to the protected location using the calls *enable ct* and *disable ct*, respectively. While *ct* is enabled, all reads and writes to ℓ are mediated by f_r and f_w, respectively.

Location caretaker specification. Our location caretaker specification (Fig. 7) ensures that the value in the protected location ℓ satisfies some stipulated predicate Ψ whenever the location is accessible to the untrusted code. The proposition $\mathsf{isloccct}\,\gamma\,\mathit{ct}\,\ell\,\Psi$ means that value *ct* is an API caretaker for location ℓ with logical name γ and predicate Ψ governing the contents of location ℓ when the caretaker is enabled. Logical names and propositions of the form $\mathsf{enabled}\,\gamma\,b$ serve the same purpose as in the API caretaker interface.

The triple MakeLocCtSpec specifies the function *makeloccct* and imposes conditions on the function's first two arguments, f_r and f_w. Since f_r receives a value read from the protected location, f_r's input can be assumed to satisfy Ψ. Since the output of f_{r} will be returned to untrusted code, that output must be low. Hence, f_{r} must transform values satisfying Ψ to low values. Dually, since f_{w} receives a value from untrusted code and its output is written to the protected location, f_{w} must transform low values to values satisfying Ψ. The postcondition for *makelocct* says that the returned functions *read*, *write* are low, so they can be passed to untrusted code safely. The precondition of *enable* insists that ℓ point to a value v satisfying Ψ before the caretaker is enabled (rule LocCtEnableSpec). Dually, the postcondition of *disable* reveals the fact that ℓ points to a value v satisfying Ψ (rule LocCtDisableSpec).

The location caretaker's code satisfies this specification, assuming the specification of the API caretaker (Fig. 6). To illustrate how programs may use a location caretaker, and how such programs may be verified, we consider the following very simple program, *client*.

$$\begin{aligned}
\mathit{asserteven} &\triangleq \lambda n.\ \mathtt{assert}\ (\mathtt{even}\ n); n \\
\mathit{assumeeven} &\triangleq \lambda n.\ \mathtt{assume}\ (\mathtt{even}\ n); n
\end{aligned}$$

$$\begin{aligned}
\mathit{client} \triangleq\ & \mathtt{let}\ r = \mathtt{ref}\ 0\ \mathtt{in} \\
& \mathtt{let}\ (ct, w) = \mathit{makelocct}\ \mathit{asserteven}\ \mathit{assumeeven}\ r\ \mathtt{in} \\
& \mathit{enable}\ ct; \\
& \mathtt{let}\ \mathit{sync} = \mathit{makesync}\ ()\ \mathtt{in} \\
& \mathtt{let}\ \mathit{use} = \mathit{sync}\ (\lambda\_.\ \mathit{disable}\ ct; \mathtt{assert}\ \mathtt{even}\ (!r); \\
& \qquad\qquad\qquad\qquad r \leftarrow 1; r \leftarrow 0; \mathit{enable}\ ct) \\
& \mathtt{in}\ (\mathit{use}, w)
\end{aligned}$$

The expression *client* constructs a reference cell r (initially containing 0) and constructs and immediately enables a location caretaker *ct* serving access to r. The goal is to maintain the invariant that untrusted code only sees even numbers in r. To ensure this invariant, the location caretaker *ct* is created with the write filter $f_{\mathrm{w}} = \mathit{assumeeven}$, which gets stuck unless the value being written is an even number. The read monitor, $f_{\mathrm{r}} = \mathit{asserteven}$, checks that the value being read from r is indeed even. Additionally, *client* exposes to its (untrusted) context a function *use* that (within the scope of an exclusive lock, *sync*) locally disables the caretaker and temporarily breaks the invariant by writing 1 to r.

Using the specification of the location caretaker, we prove that *client* satisfies the triple $\{\top\}\ \mathit{client}\ \{x.\ \mathsf{lowval}\,x\}$. By Theorem RobustSafety, *client* is robustly safe. This means that irrespective of how the (untrusted) context calls *use* and the wrapped read/write interface w, it can never observe a non-even number in r (else, the assertion in *asserteven* would fail).

5 MEMBRANE

When verified (trustworthy) and untrusted code interoperate, a general concern is that values passed from verified to untrusted code may accidentally reveal high-integrity locations. The membrane pattern (Miller 2006; Miller et al. 2008; Google, Inc. 2015) was designed to offset this possibility. A membrane is a bidirectional transformation on values passing from verified to untrusted code and vice versa. In the verified-to-untrusted direction, it sanitizes values to make the extraction of high-integrity locations impossible. A trivial way to implement the membrane is to use the *seal* function of §3, which replaces all values with proxies. However, these proxies cannot be *used* by untrusted code, other than by passing them back to the verified code. To address this shortcoming, the membrane pattern uses a *deep inspection* of values to selectively hide high-integrity locations inside values, while preserving their overall structure (pairs map to pairs, functions map to functions, etc.). This allows the untrusted code to use the transformed values, while still ensuring that it won't get access to high-integrity locations.

While the specifics of the transformation depend on the language and security objectives of the application, a common core of all membranes is a function that *lifts* a use case-specific bidirectional transformation on locations to a bidirectional transformation on values. We first define this function, *membrane*, in HLA below, show its

OCPL specification and verify the function. Then, we use this function to implement a specific membrane that we call the *public membrane*. This membrane replaces each high-integrity location nested in a value with a fresh *shadow* location and provides the verified code special functions to inspect those shadow locations. This is similar to how the membrane pattern is implemented in Google's Caja, a JavaScript library for securing communication between mutually distrusting domains in web applications (Miller et al. 2008; Google, Inc. 2015).[8]

Membrane code. The function *membrane* that forms the common core of the membrane pattern is defined below. This is a higher-order function that takes as its first two arguments two other functions, *locout* and *locin*, both of type $Loc \to Val$. The function *locout* defines how *locations* crossing from verified to untrusted code are transformed by the membrane. Dually, the function *locin* defines how locations crossing in the other direction, from untrusted to verified code, are transformed. Given these two arguments, *membrane locout locin* is a function of type $Val \to Val$ that transforms *values* passing from verified to untrusted code. The function for transforming values passing in the other direction—from untrusted to verified code—is obtained by reversing the arguments to *membrane*, *i.e.,* it is *membrane locin locout*.

```
$membrane \triangleq$ rec $memb\ locout\ locin\ x$.
    let $wrap = memb\ locout\ locin$ in
    if isfun $x$ then let $unwrap = memb\ locin\ locout$ in
                 $\lambda y.\ wrap(x(unwrap\ y))$
    else if isloc $x$ then $locout\ x$
    else if islit $x$ then $x$
    else if $x = ()$ then $()$
    else if ispair $x$ then $(wrap($fst $x), wrap($snd $x))$
    else ifinl $x$ as inl $x' \Rightarrow$ inl$(wrap\ x')$
    else ifinr $x$ as inr $x' \Rightarrow$ inr$(wrap\ x')$
    else assert false
```

Internally, *membrane* recurses on the structure of the value x being transformed. When the value is a location, the result is simply $locout\ x$. When the value is a literal, it is returned immediately. When the value is a pair, *membrane* recurses on the two components of the pair. The interesting case arises when the value x is a function. In this case, *membrane* returns a function, which when applied (by the untrusted code), first recursively applies the membrane to the untrusted argument y, then applies the given function x to the transformed argument, and then re-applies the membrane recursively to the result. Importantly, the membrane is applied in the untrusted-to-verified direction to the function argument y and in the verified-to-untrusted direction to the result of the function. Technically, the function x is transformed to $wrap \circ x \circ unwrap$, where $wrap \triangleq memb\ locout\ locin$ and $unwrap \triangleq memb\ locin\ locout$ are recursive instantiations of *membrane* in the verified-to-untrusted and untrusted-to-verified directions, respectively.

Membrane specification. Our specification of *membrane* formalizes the intuition that *membrane* lifts transformations on locations to transformations on values. The specification is shown in Fig. 8. The defined predicate ismon $p\ v\ \Psi_1\ \Psi_2$ means that v is a function that transforms values satisfying the predicate Ψ_1 to values satisfying the predicate Ψ_2 (the progress bit p is needed for technical reasons, that readers may ignore). The specification of *membrane* says that if *locout* transforms locations satisfying Ψ_1 to values satisfying Ψ_2 and *locin* does the

[8]Google's Caja partly automates the invocation of the membrane transformation by rewriting untrusted code. The automation is orthogonal to the specification and verification of the membrane pattern itself, so we do not model it here.

$$\mathsf{ismon}\, p\, v\, \Psi_1\, \Psi_2 \triangleq \forall a.\ \{\Psi_1\, a\}\ v\, a\ \{a', \mathsf{ret}\ a'.\ \Psi_2\, a'\}_p$$

MembraneSpec

$$\{\mathsf{ismon}\, p\, \mathit{locout}\, \Psi_1\, \Psi_2 * \mathsf{ismon}\, p'\, \mathit{locin}\, \Psi_2\, \Psi_1\}\ \mathit{membrane}\ \mathit{locout}\ \mathit{locin}\ \{w.\ \mathsf{ismon}\, p\, w\, (\mathsf{lift}\,\Psi_1)\, (\mathsf{lift}\,\Psi_2)\}$$

Fig. 8. Membrane specification (presupposing Ψ_1, Ψ_2 persistent)

reverse, then *membrane locout locin* transforms values satisfying lift Ψ_1 to values satisfying lift Ψ_2, where lift is the predicate transformer defined in Fig. 3. Hence, *membrane* really "lifts" the transformation on locations to a transformation on values in a precise technical sense.

This specification of *membrane* is very general, since it holds for any predicates Ψ_1 and Ψ_2. In any use of *membrane*, these predicates can be instantiated to match what the arguments *locout* and *locin* do. (Later, we show a specific instantiation with $\Psi_2 = \mathsf{lowloc}$.)

Despite the specification's generality, *membrane* is easily verified against it. In the recursive cases, the proof obligations match the inductive hypotheses precisely because of the concordance between the pattern matches in the definitions of lift (Fig. 3) and *membrane*.

Public membrane. Next, we describe how the function *membrane* can be used to construct a specific membrane, which we call the *public membrane*. This membrane is similar to the membrane used in Google's Caja library (Miller et al. 2008; Google, Inc. 2015). The public membrane maintains a unique low-integrity shadow location for every high-integrity location that the verified code declares as important. When values cross from verified to untrusted code, all nested high-integrity locations in them are effectively "replaced" with the corresponding low-integrity shadows (this transformation is implemented using *membrane*). Thus, the untrusted code only sees shadow locations, not the high-integrity locations. Additionally, the public membrane provides the verified code special functions, *shadowread* and *shadowwrite*, to read and write the contents of the shadow locations.

To understand why this is useful, consider a library that allocates an integer reference ℓ, and shares it with (untrusted) clients as an I/O buffer. Clients are expected to write only positive integers to ℓ, although the library does not strictly require this and the library's algorithms can execute safely even if the integer is not positive. Over time, many clients of this library have been written. Now, suppose that the library is updated to use different algorithms that *really require ℓ to always be positive* (else they crash). The obvious way to do this would be to rewrite the library to hold ℓ private, and to export two closures that read and write ℓ, the latter only after checking that the value being written is positive. However, this change *breaks compatibility with all existing clients*, since they must now be rewritten to invoke the new closures to access ℓ. The public membrane offers a general solution to this problem. Rather than export closures, the library can deploy a public membrane and declare ℓ as high-integrity. The membrane consistently replaces ℓ with a low-integrity shadow, say ℓ', for the library's clients. *Importantly, the library's clients don't have to change.* After a client updates ℓ' (believing that it updated ℓ), the library can access ℓ' using *shadowread* and copy it to ℓ *if* the updated value is a positive integer. Additionally, whenever the library updates ℓ internally, it can also copy the update to ℓ' using *shadowwrite*. This way, the library can maintain its new invariant *and* retain complete compatibility with existing clients.

The functions comprising the public membrane are listed in Fig. 9.[9] The expression *makepub* () creates a public membrane (consistently denoted *m*), which comprises a pair (*sync*, *tbl*) where the reference cell *tbl* contains a finite partial bijection on locations protected by (the lock buried in) *sync*. When a public membrane's table sends location ℓ_1 to ℓ_2, we say that location ℓ_1 is a (high-integrity) *private location* and location ℓ_2 its (low-integrity) shadow location. The functions *pubout* and *pubin* transform between private and shadow locations by consulting

[9] We discuss the straightforward functions *makesync*, *bijempty*, *bijinsertnew*, *bijlookup*, and *bijinvert* in Appendix F.

$$\begin{aligned}
\mathit{makepub} &\triangleq \lambda\_.\ \texttt{let}\ \mathit{tbl} = \texttt{ref}\ \mathit{bijempty}\ \texttt{in}\\
&\qquad \texttt{let}\ \mathit{sync} = \mathit{makesync}\,()\ \texttt{in}\\
&\qquad (\mathit{sync}, \mathit{tbl})\\
\mathit{pubout} &\triangleq \lambda(\mathit{sync}, \mathit{tbl})\, r_1.\\
&\quad \mathit{sync}(\lambda\_.\ \mathit{bijlookup}\,(!\,\mathit{tbl})\, r_1)\\
\mathit{pubin} &\triangleq \lambda(\mathit{sync}, \mathit{tbl})\, r_2.\\
&\quad \mathit{sync}(\lambda\_.\ \mathit{bijlookup}\,(\mathit{bijinvert}\,(!\,\mathit{tbl}))\, r_2)\\
\mathit{pubwrap} &\triangleq \lambda m.\ \mathit{membrane}\,(\mathit{pubout}\, m)\,(\mathit{pubin}\, m)
\end{aligned}$$

$$\begin{aligned}
\mathit{pubunwrap} &\triangleq \lambda m.\ \mathit{membrane}\,(\mathit{pubin}\, m)\,(\mathit{pubout}\, m)\\
\mathit{pubref} &\triangleq \lambda m\, x_1.\\
&\quad \texttt{let}\ r_1 = \texttt{ref}\ x_1\ \texttt{in}\\
&\quad \texttt{let}\ r_2 = \texttt{ref}\ (\mathit{pubwrap}\, m\, x_1)\ \texttt{in}\\
&\quad \texttt{let}\ (\mathit{sync}, \mathit{tbl}) = m\ \texttt{in}\\
&\quad \mathit{sync}(\lambda\_.\ \mathit{tbl} \leftarrow \mathit{bijinsertnew}\,(!\,\mathit{tbl})\, r_1\, r_2);\\
&\quad r_1\\
\mathit{shadowread} &\triangleq \lambda m\, r.\ \mathit{pubunwrap}\, m\,(!\,(\mathit{pubout}\, m\, r))\\
\mathit{shadowwrite} &\triangleq \lambda m\, r\, x.\ \mathit{pubout}\, m\, r \leftarrow \mathit{pubwrap}\, m\, x
\end{aligned}$$

Fig. 9. Public membrane implementation

the table. Both functions get stuck on locations not in the table. The functions *pubwrap* and *pubunwrap* lift these to transformations between *values* by applying the function *membrane*. Effectively, *pubwrap* replaces all private locations in the value passed to it with their shadow locations and *pubunwrap* does the opposite.

The expression *pubref* $m\, v_1$ constructs a new private location initially containing value v_1. In addition, it allocates a shadow location (called r_2 in the code) that initially contains the low-integrity counterpart of v_1. It stores an association between the two locations in m's table. The shadow location r_2 can be retrieved from the table by applying either *pubout* or *pubwrap* to the private location returned by *pubref*.

The function *shadowread* inspects a private location's *shadow* location, converting its low-integrity contents into a high-integrity value using *pubunwrap*, whereas *shadowwrite* updates a private location's shadow location after converting the given value to a low-integrity value using *pubwrap*.

To use this public membrane interface, the verified code first creates a new public membrane using *makepub*. Subsequently, it allocates private locations that *might* flow to untrusted code using the function *pubref* instead of the language construct `ref`. Before sending any value to untrusted code, it invokes *pubwrap* to replace all nested private locations with their shadows. Dually, after receiving any value from untrusted code, it applies *pubunwrap* to replace nested shadow locations with corresponding private locations. At any point, the verified code can access shadow locations using *shadowread* and *shadowwrite*. Appendix E shows an example of how the public membrane is used.

Public membrane specification. The OCPL specification of the public membrane is shown in Fig. 10. The assertion $\mathsf{ismembrane}\,\gamma\, m$ associates a public membrane m to a logical name γ. The abstract predicate $\mathsf{isprivloc}\,\gamma\,\ell$ represents knowledge that location ℓ is a private location for the membrane named γ, *i.e.*, ℓ was previously allocated using *pubref* on the public membrane named γ. The assertion $\mathsf{isprivval}\,\gamma\, v$ is defined as $\mathsf{lift}\,(\mathsf{isprivloc}\,\gamma)\, v$ by rule IsPrivval. In particular, the assertion means that any location "extractable" from v is in the domain of γ's table. We call such values v *private values*.

The specification rules are straightforward, so we explain only some salient points here. The triple for *pubref* (rule PubAllocSpec) says that the application *pubref* $m\, v$ returns a location $\ell \hookrightarrow v$ together with knowledge that ℓ is a private location, represented by $\mathsf{isprivloc}\,\gamma\,\ell$. Importantly, $\mathsf{isprivloc}\,\gamma\,\ell$ appears in the postcondition of only this rule. Hence, if $\mathsf{isprivloc}\,\gamma\,\ell$ holds, then ℓ must have been previously allocated by invoking *pubref* on the membrane named γ. The triples PubWrapSpec and PubUnwrapSpec say that the function *pubwrap* m converts private values to low values, while *pubunwrap* m does the opposite. In particular, the output of *pubwrap*, being low,

IsPrivval
$$\mathsf{isprivval}\,\gamma\,v \triangleq \mathsf{lift}\,(\mathsf{isprivloc}\,\gamma)\,v$$

MakePubSpec
$$\{\top\}\ \mathit{makepub}\,()\ \{m\,\gamma, \mathsf{ret}\ m.\ \mathsf{ismembrane}\,\gamma\,m\}$$

PubAllocSpec
$$\{\mathsf{ismembrane}\,\gamma\,m * \mathsf{isprivval}\,\gamma\,v\}\ \mathit{pubref}\,m\,v\ \{\ell, \mathsf{ret}\ \ell.\ \mathsf{isprivloc}\,\gamma\,\ell * \ell \hookrightarrow v\}$$

PubWrapSpec
$$\{\mathsf{ismembrane}\,\gamma\,m * \mathsf{isprivval}\,\gamma\,v_1\}\ \mathit{pubwrap}\,m\,v_1\ \{x_2.\ \mathsf{lowval}\,x_2\}$$

PubUnwrapSpec
$$\{\mathsf{ismembrane}\,\gamma\,m * \mathsf{lowval}\,v_2\}\ \mathit{pubunwrap}\,m\,v_2\ \{x_1.\ \mathsf{isprivval}\,\gamma\,x_1\}_?$$

ShadowReadSpec
$$\{\mathsf{ismembrane}\,\gamma\,m * \mathsf{isprivloc}\,\gamma\,\ell\}\ \mathit{shadowread}\,m\,\ell\ \{x.\ \mathsf{isprivval}\,\gamma\,x\}_?$$

ShadowWriteSpec
$$\{\mathsf{ismembrane}\,\gamma\,m * \mathsf{isprivloc}\,\gamma\,\ell * \mathsf{isprivval}\,\gamma\,v\}\ \mathit{shadowwrite}\,m\,\ell\,v\ \{\mathsf{ret}\ ().\ \top\}$$

Fig. 10. Public membrane specification

can be freely shared with untrusted code. Also, *pubunwrap* has a non-progressive triple because unwrapping an arbitrary low value v_2 may apply $\mathit{pubin}\,m\,\ell_2$ with some low-integrity location ℓ_2 unknown to m, and our implementation of *pubin* gets stuck in such cases.[10]

The implementation of the public membrane satisfies this specification. As an example, we briefly describe how we verify the *pubwrap* specification, PubWrapSpec. First, we prove the following two lemmas showing that for a valid membrane m, $\mathit{pubout}\,m$ and $\mathit{pubin}\,m$ produce functions that satisfy the predicate ismon.

$$\{\mathsf{ismembrane}\,\gamma\,m\}\ \mathit{pubout}\,m\ \{f.\ \mathsf{ismon}\,\mathtt{progress}\,f\,(\mathsf{isprivloc}\,\gamma)\,\mathsf{lowloc}\}$$

$$\{\mathsf{ismembrane}\,\gamma\,m\}\ \mathit{pubin}\,m\ \{f.\ \mathsf{ismon}\,\mathtt{noprogress}\,f\,\mathsf{lowloc}\,(\mathsf{isprivloc}\,\gamma)\}$$

Next, we instantiate MembraneSpec (Fig. 8) with $\Psi_1 = \mathsf{isprivloc}\,\gamma$, $\Psi_2 = \mathsf{lowloc}$ and the two lemmas above to obtain

$$\mathsf{ismon}\,\mathtt{progress}\,\mathit{pubwrap}\,(\mathsf{isprivval}\,\gamma)\,\mathsf{lowval}$$

The triple PubWrapSpec follows by unfolding the definition of ismon (Fig. 8).

6 RELATED WORK

This paper focuses on the formal specification and verification of OCPs. While we know of only rather preliminary prior work in this space, a major point of our paper is to observe that the tools needed for effective verification of OCPs are already to a large extent available—the pieces just need to be assembled properly. In particular, our logic, OCPL, gets significant mileage out of existing verification techniques, notably robust safety and concurrent separation logic. In the following, we discuss related work on robust safety, the (limited amount of) existing work on specification of OCPs, and some other relevant work on object capabilities.

Robust safety. The concept of robust safety arose in the context of verifying security protocols that interact with adversaries. Early work used typing to prove "correspondence properties" for cryptographic protocols modeled in the spi calculus (Gordon and Jeffrey 2001). In their work on the refinement type-checker F7, Bengtson et al. (2011) generalized robust safety to a richer class of integrity properties for a process calculus, RCF, with higher-order

[10]For simplicity, we do not specify the functionality of *pubwrap* and *pubunwrap* beyond "returns a low, resp., private value". More useful specifications would allow one to prove that, say, after wrapping private location ℓ_1 twice and obtaining values v_2 and v_2', there exists a low location ℓ_2 such that $v_2 = \ell_2 = v_2'$.

state. We inherit from their work the basic idea of using a notion of low-integrity values and proving robust safety, but the approaches differ greatly in detail. First, we show how to apply this idea to OCPs, a completely different domain. Second, we show how low-integrity values are directly encodable in modern separation logics, using a simple logical relation.

Object capabilities. There has been only very preliminary work on specifying and verifying functional properties of object capabilities and OCPs. In his seminal paper on dynamic sealing, Morris (1973) proposed informal reasoning principles for programmers using dynamic sealing, but did not prove anything formally. Drossopoulou et al. (2015a) proposed predicates modeling trust and risk and used those predicates to specify a capability-based escrow exchange example (Miller et al. 2013). They focused on this one example, whereas we develop general specifications for several OCPs. Further, they focus on syntactic specifications and do not define the semantics of their predicates. A subsequent manuscript (Drossopoulou et al. 2015b) bridges this gap to the semantics, but their Hoare logic seems inadequate for the examples we consider, *e.g.,* it lacks a rule for dynamic allocation.

Perhaps the most closely related work to our own is that of Devriese et al. (2016). As discussed in the introduction, they use a Kripke logical relation and a meta-property called effect parametricity to verify integrity properties for several examples of capability-wrapped user code in a language with higher-order state. There are several points of difference between our work and theirs. First, we work in a concurrent separation logic rather than directly in a low-level logical relation. As a result, in addition to being able to conduct our proofs at a much higher level of abstraction, we can give *compositional* specifications for higher-order object capability *patterns* (*i.e.,* libraries), whereas they only verify specific programs that use object capabilities. We also exploit the notion of robust safety in verifying integrity properties of code that uses OCPs, whereas corresponding arguments only appear implicitly in their proofs. On the other hand, they develop semantic variants of the so-called "reference graph properties" from the literature on object capabilities. These properties are important but they are also orthogonal to verification and, hence, we do not examine them. Last but not least, our proofs are machine-checked in Coq.

While specifying OCPs has received little attention so far, there has been a lot of other work on object capabilities and OCPs. Responding to renewed interest in dynamic sealing, Sumii and Pierce (2004) propose a bisimulation for proving contextual equivalences in a language with a dynamic sealing primitive. Roughly, the examples they consider are pairs of expressions—"modules" implementing the same "interface" using sealing—and the question they study is whether those expressions are indistinguishable, even if their internal representations differ. Bengtson et al. (2011) use dynamic sealing to define ideal implementations of cryptographic operations in RCF. They offer no general specifications for dynamic sealing analogous to the specification in §3, but derive *instances* of such a specification as needed. Spiessens and Van Roy (2005); Spiessens (2007) and Murray (2010) use, respectively, model- and refinement-checking tools to establish certain safety and liveness properties of abstract models of object capability systems, including some OCPs. To take one small example, Murray shows that a specific model of an unsealing operation does not reveal a sealed value unless a capability to that sealed value was passed (possibly indirectly) to the unsealing operation by the context. Although useful, such properties cannot be directly used to verify clients of the OCPs. In contrast, our goal is very different: we write compositional specifications for concrete implementations of OCPs and our specifications can be directly used to verify clients.

ACKNOWLEDGMENTS

This research was supported in part by a Microsoft Research PhD Scholarship, and in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

REFERENCES

Martín Abadi. 1999. Secrecy by Typing in Security Protocols. *J. ACM* 46, 5 (Sept. 1999), 749–786.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. 109–122.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (Feb. 2011), 8:1–8:45.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke Models over Recursive Worlds. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. 119–132.

Douglas Crockford. 2008. Making JavaScript Safe for Advertising. (2008). Retrieved April, 2017 from http://www.adsafe.org/

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)*. 147–162.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP '10)*. 504–528.

Sophia Drossopoulou, James Noble, and Mark S. Miller. 2015a. Swapsies on the Internet: First Steps Towards Reasoning About Risk and Trust in an Open World. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security (PLAS'15)*. 2–15.

Sophia Drossopoulou, James Noble, Mark S. Miller, and Toby Murray. 2015b. *Reasoning about Risk and Trust in an Open World.* Technical Report ECSTR-15-08. Victoria University of Wellington.

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.* 103, 2 (Sept. 1992), 235–271.

Google, Inc. 2015. Caja membrane implementation. (Feb. 2015). Retrieved July, 2015 from https://github.com/google/caja/blob/master/src/com/google/caja/plugin/taming-membrane.js

Andrew D. Gordon and Alan Jeffrey. 2001. Authenticity by Typing for Security Protocols. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW '01)*. 145–159.

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 '16)*. 256–269.

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 '15)*. 637–650.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*. 696–723.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. 205–217.

Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '10)*.

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University.

Mark S. Miller. 2008. Sealers and Unsealers. (2008). Retrieved February, 2017 from http://wiki.erights.org/wiki/Walnut/Secure_Distributed_Computing/Capability_Patterns#Sealers_and_Unsealers

Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in Javascript. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. 1–20.

Mark S. Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-Based Financial Instruments. In *Proceedings of the 4th International Conference on Financial Cryptography (FC '00)*. 349–378.

Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Caja: Safe active content in sanitized JavaScript. (June 2008). Retrieved February, 2017 from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.306.6704 Unpublished draft.

Mark S. Miller and Jonathan S. Shapiro. 2003. Paradigm Regained: Abstraction Mechanisms for Access Control. In *Advances in Computing Science – ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference (ASIAN '03)*. 224–242. Springer LNCS 2896.

James H. Morris, Jr. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21.

Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns.* Ph.D. Dissertation. Hertford College.

Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2014. Typed-based verification of Web sandboxes. *J. Comput. Secur.* 22, 4 (July 2014), 511–565.

Alfred Spiessens. 2007. *Patterns of Safe Collaboration.* Ph.D. Dissertation. Université catholique de Louvain.

Fred Spiessens and Peter Van Roy. 2004. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Proceedings of the Second International Conference on Multiparadigm Programming in Mozart/Oz (MOZ '04)*. 21–40.

Fred Spiessens and Peter Van Roy. 2005. A Practical Formal Model for Safety Analysis in Capability-based Systems. In *Proceedings of the 1st International Conference on Trustworthy Global Computing (TGC'05)*. 248–278.

Marc Stiegler and Mark Miller. 2006. *How Emily Tamed the Caml*. Technical Report HPL-2006-116. HP Laboratories.

Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. 161–172.

Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. 363–378.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. 377–390.

A SEALING WITH LOW FUNCTIONS

In this appendix, we point out that, in the dynamic sealing OCP, representing proxy values as low-integrity *functions* renders unsealing non-deterministic, forcing somewhat surprising changes to otherwise "obviously safe" code.

Our implementation of sealing differs from Morris' in one important respect. Ignoring synchronization, Morris sealed a value v by returning a *function* $(\lambda k.\ tbl \leftarrow mapinsertnew\,(!\,tbl)\,k\,v)$ for extending the table.[11] To unseal a value f, Morris allocated a fresh location k, applied $f\ k$, and looked for k in the table. This simple change—returning a function rather than a location—renders

$$\textsc{UnsealAnySpec}\quad \{\mathsf{isunseal}\,\gamma\,u\,\phi\}\ u\,v'\ \{x.\ \mathsf{issealed}\,\gamma\,x\,v'\,\phi\}_?$$

unsound. There are two problems. First, the triple makes no assumptions about the value v'. When v' is a function f, there is no way to *safely* apply $f\ k$ as f might have "unfair" access to otherwise hidden state. An easy fix is to weaken the triple, assuming that v' is low so that, if v' is a function f, we know by assumption that we can apply $f\ k$ (if we can mark k low—we can). Second, the triple returns a value v such that $\mathsf{issealed}\,\gamma\,v\,v'\,\phi$ which means that subsequent applications $unseal\,v'$ must *also* return v. This is absurd! An adversary with values f_1, f_2 obtained by sealing values v_1, v_2 can, for example, construct

$$f \triangleq \mathtt{let}\ r = \mathtt{ref\ false\ in}\ \lambda k.\ \mathtt{let}\ b = \,!r\ \mathtt{in}\ (r \leftarrow (\mathtt{not}\,b); (\mathtt{if}\ b\ \mathtt{then}\ f_1\ \mathtt{else}\ f_2)\,k)$$

that, when unsealed, oscillates between v_1 and v_2. An easy fix is to weaken the triple further, giving up on deterministic unsealing.

We verified Morris' implementation of sealing against a weaker specification, obtained by replacing rule UnsealAnySpec in Fig. 4 with

$$\textsc{UnsealLowSpec}\quad \{\mathsf{isunseal}\,\gamma\,u\,\phi * \mathsf{lowval}\,v'\}\ u\,v'\ \{x.\ \phi\,x\}_?$$

This switch to non-deterministic unsealing of low values comes at some cost. We must weaken the non-progressive triples in our intervals specification; for example, replacing MinAnySpec by

$$\textsc{MinLowSpec}\quad \{\mathsf{ismin}\,\gamma\ imin * \mathsf{lowval}\,v\}\ imin\,v\ \{n, \mathsf{ret}\ n.\ \top\}_?$$

With these weaker rules, we can no longer verify the function $check = \lambda j.\ \mathtt{assert}\ (imin\,j \le imax\,j)$ in our intervals client. Applying *check* to a monster like the oscillating function f can cause the assertion to fail. All is not lost, of course. We can rewrite *check* to read $check = \lambda j.\ \mathtt{let}\ j = snap\,j\ \mathtt{in\ assert}\ (imin\,j \le imax\,j)$ after extending our intervals library with the trivial snapshot function $snap = \lambda i.\ seal\,(unseal\,i)$. The point of *snap* is to replace an unpredictable function by a predictable one:

$$\textsc{SnapSpec}\quad \{\mathsf{issnap}\,\gamma\ snap * \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v\}\ snap\,v\ \{x'.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,x'\}$$

$$\textsc{SnapLowSpec}\quad \{\mathsf{issnap}\,\gamma\ snap * \mathsf{lowval}\,v\}\ snap\,v\ \{v'\,n_1\,n_2, \mathsf{ret}\ v'.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v'\}_? \qquad \textsc{SnapLow}\quad \mathsf{issnap}\,\gamma\ snap \vdash \mathsf{lowval}\ snap$$

With these changes, we can prove the assertion expression in our modified intervals client succeeds and the client is robustly safe.

[11] We are not so much interested in Morris' implementation but in the fact that it returns a function. Other dynamic sealing implementations return functions and otherwise have attractive properties (*e.g.,* dispensing with the table). Ignoring synchronization, the table-free implementation of sealing (Miller 2008; Taly et al. 2011)

$$\mathtt{let}\ r = \mathtt{ref\ None\ in\ let}\ seal = \lambda x\,\_.\ r \leftarrow \mathtt{Some}\ x\ \mathtt{in\ let}\ unseal = \lambda f.\ (r \leftarrow \mathtt{None}; f\,(); \mathtt{valOf}\,(!\,r))\ \mathtt{in}\ (seal, unseal)$$

shares this drawback with Morris' implementation.

Asymmetric signatures.

MakeSignSpec
$$\frac{\phi \text{ persistent} \qquad (\forall v)\ \phi\, v \vdash \mathsf{lowval}\, v}{\{\top\}\ \ \mathit{makeseal}\ ()\ \ \{v_1\, v_2\, \gamma, \mathsf{ret}\ (v_1, v_2).\ \mathsf{issign}\, \gamma\, v_1\, \phi * \mathsf{isverify}\, \gamma\, v_2\, \phi\}}$$

SignSpec
$$\{\mathsf{issign}\, \gamma\, \mathit{sign}\, \phi * \phi\, v\}\ \mathit{sign}\, v\ \{x'.\ \mathsf{issigned}\, \gamma\, v\, x'\, \phi\}$$

VerifySpec
$$\{\mathsf{isverify}\, \gamma\, \mathit{verify}\, \phi * \mathsf{issigned}\, \gamma\, v\, v'\, \phi\}\ \mathit{verify}\, v'\ \{\mathsf{ret}\ v.\ \top\}$$

VerifyAnySpec
$$\{\mathsf{isverify}\, \gamma\, \mathit{verify}\, \phi\}\ \mathit{verify}\, v'\ \{x.\ \mathsf{issigned}\, \gamma\, x\, v'\, \phi\}_?$$

VerifyLow
$$\mathsf{isverify}\, \gamma\, \mathit{verify}\, \phi \vdash \mathsf{lowval}\, \mathit{verify}$$

SignedLow
$$\mathsf{issigned}\, \gamma\, v\, v'\, \phi \vdash \mathsf{lowval}\, v'$$

SignedInv
$$\mathsf{issigned}\, \gamma\, v\, v'\, \phi \vdash \phi\, v$$

SignedAgree
$$\mathsf{issigned}\, \gamma\, v_1\, v'\, \phi * \mathsf{issigned}\, \gamma\, v_2\, v'\, \phi \vdash v_1 = v_2$$

Asymmetric encryption.

MakeEncryptSpec
$$\{\top\}\ \ \mathit{makeseal}\ ()\ \ \{v_1\, v_2\, \gamma, \mathsf{ret}\ (v_1, v_2).\ \mathsf{isencrypt}\, \gamma\, v_1 * \mathsf{isdecrypt}\, \gamma\, v_2\}$$

EncryptSpec
$$\{\mathsf{isencrypt}\, \gamma\, \mathit{enc}\}\ \mathit{enc}\, v\ \{x'.\ \mathsf{isctext}\, \gamma\, v\, x'\}$$

DecryptSpec
$$\{\mathsf{isdecrypt}\, \gamma\, \mathit{dec} * \mathsf{isctext}\, \gamma\, v\, v'\}\ \mathit{dec}\, v'\ \{\mathsf{ret}\ v.\ \top\}$$

DecryptAnySpec
$$\{\mathsf{isdecrypt}\, \gamma\, \mathit{dec}\}\ \mathit{dec}\, v'\ \{x.\ \mathsf{isctext}\, \gamma\, x\, v'\}_?$$

EncryptLow
$$\mathsf{isencrypt}\, \gamma\, \mathit{enc} \vdash \mathsf{lowval}\, \mathit{enc}$$

CtextLow
$$\mathsf{isctext}\, \gamma\, v\, v' \vdash \mathsf{lowval}\, v'$$

CtextAgree
$$\mathsf{isctext}\, \gamma\, v_1\, v' * \mathsf{isctext}\, \gamma\, v_2\, v' \vdash v_1 = v_2$$

Fig. 11. Public-key interfaces (derived from Fig. 4)

B PUBLIC-KEY INTERFACES FOR SEALING

In this appendix, we show that dynamic sealing can be used to implement *ideal* cryptographic signing/verification as well as encryption/decryption primitives, and that the specification of dynamic sealing (Fig. 4) can be *specialized* to derive very general specifications for these primitives. Such ideal primitives are sometimes used in the verification of cryptographic protocols Bengtson et al. (2011). While the observation that dynamic sealing can implement these cryptogrpahic primitives is not new—Morris (1973) notes this even though he did not use the terms signing and encryption back then—the observation that specifications for the cryptographic primitives can be derived from those for dynamic sealing is new.

Signature/verification. A signature/verification primitive provides a function *sign* that creates a low proxy for its argument. This proxy is usually called a signature. The proxy (signature) can be verified using a dual function *verify*, which when applied to the value returned by *sign* v, returns v and if the argument was not the output of *sign*, gets stuck. The important point is how *sign* and *verify* are used in practice. In typical use, *sign* is held private by a piece of code that has been verified to apply *sign* only to values that satisfy a representation invariant ϕ. The *verify* function is publicly available to everyone and anyone can use it to verify signatures created by the code that holds *sign*. In this mode of use, signing/verification *transfers* knowledge of the representation invariant from the signer to the verifier: Since the signer only signs values with representation invariant ϕ, and *verify* only returns previously signed values, the verifier always knows that any value returned by *verify* must satisfy ϕ.

It should be clear that *sign* and *verify* are quite similar to *seal* and *unseal* returned by *makeseal* (). The top half of Fig. 11 shows that, in fact, *seal* and *unseal* *are* an implementation of *sign* and *verify*. The specification shown in this figure is a specific instance of the general sealing specification from Figure 4, derived by defining the abstract predicates issign, isverify, and issigned to be isseal, isunseal, and issealed, respectively; using UnsealLow to prove VerifyLow; and dropping SealLow. The derived specification says that *makeseal* () returns a pair of functions that behave like *sign* and *verify* if the representation invariant ϕ implies lowval. This condition is necessary since *verify* returns previously signed values to untrusted code, so signed values should always be low—in cryptographic terms, signatures provide no secrecy. Additionally, the specification says that the value returned by *verify* always satisfies ϕ (rule VerifyAnySpec), that *verify* itself is low (rule VerifyLow) so *verify* can be shared safely with untrusted code, and that the output of *sign* is low (rule SignedLow) so signatures can also be shared safely with untrusted code. On the other hand, *sign* itself is not low, so it should not be shared with untrusted code.

Encryption/decryption. An encryption/decryption primitive provides a function *enc* that creates a low proxy for its argument. A dual function *dec* converts the proxy back to the original argument. Typically, encryption/decryption is used by holding *dec* private in some piece of code, and making *enc* public. This allows anyone to encrypt a secret using *enc* and share it freely, with the guarantee that only the code holding *dec* can ever access the secret.

Again, *enc* and *dec* look similar to *seal* and *unseal* returned by *makeseal* () and the bottom half of Fig. 11 formalizes this intuition by specializing the specification of Fig. 4 to match the intuitive description of encryption/decryption. The abstract predicates isencrypt, isdecrypt, and isctext are defined to be isseal, isunseal, and issealed, respectively. Importantly, the specification says that *makeseal* () returns functions that behave like *enc* and *dec* (rule MakeEncryptSpec), that decrypting a previously encrypted value v returns v (rules EncryptSpec and DecryptSpec), and that the encryption function and ciphertexts are low (rules EncryptLow and CtextLow) so they can be shared with untrusted code. However, the decryption function is not low, so it should not be shared with untrusted code.

C INTERVALS INTERFACE

The following rules constitute the full specification of the intervals library in §3.

Progressive triples.

IntervalsSpec

$$\{\top\}\ \mathit{intervals}\,()\ \{v_1\,v_2\,v_3\,v_4\,\gamma, \mathsf{ret}\ (v_1, v_2, v_3, v_4).\ \mathsf{ismakeint}\,\gamma\,v_1 * \mathsf{ismin}\,\gamma\,v_2 * \mathsf{ismax}\,\gamma\,v_3 * \mathsf{issum}\,\gamma\,v_4\}$$

MakeintSpec

$$\{\mathsf{ismakeint}\,\gamma\,\mathit{mk}\}\ \mathit{mk}\,n_1\ \{f.\ \mathsf{ismakeint}'\,\gamma\,n_1\,f\}$$

Makeint′Spec

$$\{\mathsf{ismakeint}'\,\gamma\,n_1\,f\}\ \ f\,n_2\ \ \{i.\ \mathsf{isinterval}\,\gamma\,(\min n_1\,n_2)\,(\max n_1\,n_2)\,i\}$$

MinSpec

$$\{\mathsf{ismin}\,\gamma\,\mathit{imin} * \mathsf{isinterval}\,\gamma\,n_1\,n_2\,i\}\ \mathit{imin}\,i\ \{\mathsf{ret}\ n_1.\ \top\}$$

MaxSpec

$$\{\mathsf{ismax}\,\gamma\,\mathit{imax} * \mathsf{isinterval}\,\gamma\,n_1\,n_2\,i\}\ \mathit{imax}\,i\ \{\mathsf{ret}\ n_2.\ \top\}$$

SumSpec

$$\{\mathsf{issum}\,\gamma\,\mathit{isum} * \mathsf{isinterval}\,\gamma\,n_1\,n_2\,i\}\ \mathit{isum}\,i\ \{f.\ \mathsf{issum}'\,\gamma\,n_1\,n_2\,f\}$$

Sum′Spec

$$\{\mathsf{issum}'\,\gamma\,n_1\,n_2\,f * \mathsf{isinterval}\,\gamma\,n_1'\,n_2'\,i'\}\ \ f\,i'\ \ \{j.\ \mathsf{isinterval}\,\gamma\,(n_1 + n_1')\,(n_2 + n_2')\,j\}$$

IntervalInv

$$\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i \vdash n_1 \le n_2$$

IntervalAgree

$$\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i * \mathsf{isinterval}\,\gamma\,n_1'\,n_2'\,i \vdash n_1 = n_1' * n_2 = n_2'$$

Non-progressive triples.

MinAnySpec
$$\{\mathsf{ismin}\,\gamma\,\mathit{imin}\}\ \mathit{imin}\,v\ \{n_1\,n_2, \mathsf{ret}\ n_1.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v\}_?$$

MaxAnySpec
$$\{\mathsf{ismax}\,\gamma\,\mathit{imax}\}\ \mathit{imax}\,v\ \{n_1\,n_2, \mathsf{ret}\ n_2.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v\}_?$$

SumAnySpec
$$\{\mathsf{issum}\,\gamma\,\mathit{isum}\}\ \mathit{isum}\,v_1\ \{f\,n_1\,n_2, \mathsf{ret}\ f.\ \mathsf{isinterval}\,\gamma\,n_1\,n_2\,v_1 * \mathsf{issum}'\,\gamma\,n_1\,n_2\,f\}_?$$

Sum′AnySpec
$$\{\mathsf{issum}'\,\gamma\,n_1\,n_2\,f\}\ f\,v_2\ \{i\,n_1'\,n_2', \mathsf{ret}\ i.\ \mathsf{isinterval}\,\gamma\,n_1'\,n_2'\,v_2 * \mathsf{isinterval}\,\gamma\,(n_1+n_1')\,(n_2+n_2')\,i\}_?$$

Low values.

IntervalLow
$$\mathsf{isinterval}\,\gamma\,n_1\,n_2\,i \vdash \mathsf{lowval}\,i$$

Makeint′Low
$$\mathsf{ismakeint}'\,\gamma\,n_1\,f \vdash \mathsf{lowval}\,f$$

MakeintLow
$$\mathsf{ismakeint}\,\gamma\,\mathit{mk} \vdash \mathsf{lowval}\,\mathit{mk}$$

MinLow
$$\mathsf{ismin}\,\gamma\,\mathit{imin} \vdash \mathsf{lowval}\,\mathit{imin}$$

MaxLow
$$\mathsf{ismax}\,\gamma\,\mathit{imax} \vdash \mathsf{lowval}\,\mathit{imax}$$

Sum′Low
$$\mathsf{issum}'\,\gamma\,n_1\,n_2\,f \vdash \mathsf{lowval}\,f$$

SumLow
$$\mathsf{issum}\,\gamma\,\mathit{isum} \vdash \mathsf{lowval}\,\mathit{isum}$$

D BLOCKING CARETAKERS

In this appendix, we briefly present a simpler implementation of API caretakers (Fig. 6):

$$\mathit{makecaretaker} \triangleq \mathit{makelock}' \qquad \mathit{enable} \triangleq \mathit{release}$$
$$\mathit{wrap} \triangleq \lambda ct\,f\,x.\ \mathit{syncwith}\,ct\,(\lambda\_.\ f\,x) \qquad \mathit{disable} \triangleq \mathit{acquire}$$

The idea is that, if we are happy to let wrappers block until a caretaker is enabled, we can dispense with the *enabled* flag used in §4. In this implementation, a caretaker is a lock. The function *makelock′* allocates a fresh lock, initially locked. The function *syncwith* is analogous to *sync*, except that it takes the lock to use as an initial argument. To disable and enable wrappers, we simply acquire and release the lock they need. This implementation also satisfies the API caretaker specification (Fig. 6).

E PUBLIC MEMBRANE CLIENT

In this appendix, we further illustrate the public membrane by describing a simple client program that uses it. First, we define the following auxiliary function, *getint*. This function presupposes that we have some invariant on a private integer location, *r*, and that we want to periodically update *r* by "merging" its contents with those of its shadow location, using some merge function *f* that preserves the invariant.

```
getint ≜ λm f r. let $n_1$ = !r in let $x_2$ = shadowread m r in
                 if isint $x_2$ then
                     let $n_3$ = f $n_1$ $x_2$ in
                     let _ = if $n_1 \neq n_3$ then r ← $n_3$ else () in
                     let _ = if $x_2 \neq n_3$ then shadowwrite m r $n_3$ else () in
                     $n_3$
                 else (shadowwrite m r $n_1$; $n_1$)
```

If the private location r and its shadow contain integers n_1 and n_2, the code applies $f\ n_1\ n_2$ to obtain the "merged" integer n_3, updates r and its shadow, and returns n_3; otherwise, the code overwrites the shadow's contents with (and returns) n_1.

Our illustrative client program is a private up/down counter with increment and decrement functions, and private lower and upper limits. The invariant we have in mind is that the counter always remains within these limits. However, the counter also exposes shadow locations for the lower and upper limits to its untrusted context. The context can freely update these shadow lower and upper limits. However, these changes are propagated to the private locations (using *getint*) only if the changes maintain the intended invariant.

```
client ≜ let m = makepub () in
         let lo = pubref m 0 in let c = ref 0 in let hi = pubref m 0 in
         let sync = makesync () in
         let use = λ_. sync(λ_. assert !lo ≤ !c; assert !c ≤ !hi)
         let getlimits = λ_. let n = !c in
                             let a = getint m (λ$n_1$ $n_2$. if $n_2$ ≤ n then $n_2$ else $n_1$) lo in
                             let b = getint m (λ$n_1$ $n_2$. if n ≤ $n_2$ then $n_2$ else $n_1$) hi in
                             (a, b)
         let decr = λ_. sync(λ_. let n = (!c) − 1 in
                                 let b = fst (getlimits ()) ≤ n in
                                 if b then c ← n else (); b)
         let incr = λ_. sync(λ_. let n = (!c) + 1 in
                                 let b = n ≤ snd (getlimits ()) in
                                 if b then c ← n else (); b)
         let lo′ = pubwrap m lo in let hi′ = pubwrap m hi in
         (use, lo′, hi′, incr, decr)
```

The *client*'s private state comprises a public membrane m, a private counter c and private limits lo and hi, subject to the representation invariant $!\,lo \leq\ !c \leq\ !\,hi$ (*i.e.,* the counter respects its limits). The *client* returns (i) a simple function *use* that, when applied, simply asserts the representation invariant (and that could be extended to offer some service to the counter's clients), (ii) the shadow locations lo' and hi' that substitute lo and hi for the surrounding untrusted code (the untrusted code may freely change these at any time) and (iii) increment and decrement functions, *incr* and *decr*, that either change the counter c and return `true` or (when changing the counter would violate its current limits) return `false`. Prior to changing the counter, *incr* and *decr* copy the shadow locations lo' and hi' into lo and hi using the function *getint*, if doing so will not break the representation invariant.

Using the rules of OCPL, we prove that *client* is robustly safe, meaning that no context can exploit the returned values *use*, *incr*, *decr*, lo' and hi' in a way that violates the assertions about the representation invariant in *use*. To do this, we first prove that *client* satisfies the following triple: $\{\top\}$ *client* $\{v.\ \mathsf{lowval}\, v\}$. One key step in this proof is showing that lo' and hi' are both low. This follows because lo' and hi' are obtained from *pubwrap* and the postcondition of *pubwrap* (rule PubWrapSpec, Fig. 10) says that the value it returns is low. Robust safety follows immediately from Theorem RobustSafety applied to this triple.

Initially locked locks.

LockedExclusive
$$\mathsf{locked}\,\gamma * \mathsf{locked}\,\gamma \vdash \bot$$

MakeLock′Spec
$$\{\top\}\ \mathit{makelock'}\,()\ \{\mathit{lk}\,\gamma, \mathsf{ret}\ \mathit{lk}.\ \mathsf{islock}\,\gamma\ \mathit{lk}\,R * \mathsf{locked}\,\gamma\}$$

ReleaseSpec
$$\{\mathsf{islock}\,\gamma\ \mathit{lk}\,R * \mathsf{locked}\,\gamma * R\}\ \mathit{release}\ \mathit{lk}\ \{\mathsf{ret}\ ().\ \top\}$$

AcquireSpec
$$\{\mathsf{islock}\,\gamma\ \mathit{lk}\,R\}\ \mathit{acquire}\ \mathit{lk}\ \{\mathsf{ret}\ ().\ \mathsf{locked}\,\gamma * R\}$$

Initially unlocked locks.

MakeLockSpec
$$\{R\}\ \mathit{makelock}\,()\ \{\mathit{lk}\,\gamma, \mathsf{ret}\ \mathit{lk}.\ \mathsf{islock}\,\gamma\ \mathit{lk}\,R\}$$

Synchronization.

IsSync
$$\mathsf{issync}\ \mathit{sync}\,R \triangleq \Box\forall Q.\ \{R\}\ e\ \{v.\ R * Q\} \twoheadrightarrow \{\top\}\ \mathit{sync}\,(\lambda\_.\ e)\ \{v.\ Q\}$$

SyncWithSpec
$$\{\mathsf{islock}\,\gamma\ \mathit{lk}\,R\}\ \mathit{syncwith}\ \mathit{lk}\ \{v.\ \mathsf{issync}\,v\,R\}$$

MakeSyncSpec
$$\{R\}\ \mathit{makesync}\,()\ \{v.\ \mathsf{issync}\,v\,R\}$$

Fig. 12. Lock and synchronization interface

F LIBRARY FUNCTIONS

In this appendix, we specify the implementations of locks, finite maps, and finite bijections used in our examples. As nothing here is original or terribly interesting, we relegate many implementation and verification details to our Coq development.

Locks and synchronization. We specify locks and synchronization in Fig. 12. Rules LockedExclusive–AcquireSpec constitute a CAP-style lock specification (Dinsdale-Young et al. 2010) satisfied by, for example, the evident implementation of spin locks using CAS. The assertion $\mathsf{islock}\,\gamma\ \mathit{lk}\,R$ represents knowledge that the value lk is a lock with abstract name γ protecting resource R. The assertion $\mathsf{locked}\,\gamma$ represents ownership of the lock named γ. Thus, for any resource R,

- $\mathit{makelock'}\,()$ returns a new, locked lock lk protecting R (MakeLock′Spec);
- $\mathit{release}\ \mathit{lk}$ gives up lock lk and resource R, returning unit (ReleaseSpec); and
- $\mathit{acquire}\ \mathit{lk}$ grabs lock lk and the resource R it protects, returning unit (AcquireSpec).

This specification is atypical in that $\mathit{makelock'}$ constructs a locked lock, a feature needed only to argue that our caretaker specification is just a specialized lock spec (see the implementation of “blocking caretakers” in Appendix D). MakeLockSpec embodies a typical CAP-style lock specification: it specifies the evident wrapper around $\mathit{makelock'}$ and $\mathit{release}$ that consumes resource R in order to construct a new, unlocked lock lk protecting resource R.

Rules IsSync–MakeSyncSpec specify the following simple library, readily verified against any lock implementation.

$$\mathit{syncwith} \triangleq \lambda \mathit{lk}\,f.\ \mathit{acquire}\ \mathit{lk}; \mathtt{let}\ r = f\,()\ \mathtt{in}\ \mathit{release}\ \mathit{lk}; r$$

$$\mathit{makesync} \triangleq \lambda\_.\ \mathtt{let}\ \mathit{lk} = \mathit{makelock}\,()\ \mathtt{in}\ \mathit{syncwith}\ \mathit{lk}$$

Given a lock lk and a thunk f, the function $\mathit{syncwith}$ applies f while holding the lock, returning whatever $f()$ returns. The function $\mathit{makesync}$ returns $\mathit{syncwith}\ \mathit{lk}$ for a fresh lock lk. In specifying $\mathit{makesync}$, we follow Turon et al. (2013). Definition IsSync is the key. It says that to prove $\mathit{sync}\,(\lambda\_.\ e)$ returns a value satisfying Q, it suffices

$$\textsc{MapEmptySpec} \quad \mathsf{ismap}\ \mathit{mapempty}\ \emptyset$$

$$\textsc{MapInsertNewSpec} \quad \{\mathsf{ismap}\ \mathit{map}\ f * x \notin \mathsf{dom}\ f\}\ \mathit{mapinsertnew}\ \mathit{map}\ (\iota\, x)\ v\ \{\mathit{map}'.\ \mathsf{ismap}\ \mathit{map}'\ f[x \mapsto v]\}$$

$$\textsc{MapLookupPartialSpec} \quad \{\mathsf{ismap}\ \mathit{map}\ f\}\ \mathit{maplookup}\ \mathit{map}\ (\iota\, x)\ \{y.\ f\ x = y\}_?$$

Fig. 13. Finite map interface

$$\textsc{BijEmptySpec} \quad \mathsf{isbij}\ \mathit{bijempty}\ \emptyset\ \emptyset$$

$$\textsc{BijInvertSpec} \quad \{\mathsf{isbij}\ \mathit{bij}\ f\ f'\}\ \mathit{bijinvert}\ \mathit{bij}\ \{\mathit{bij}'.\ \mathsf{isbij}\ \mathit{bij}'\ f'\ f\}$$

$$\textsc{BijInsertNewSpec} \quad \{\mathsf{isbij}\ \mathit{bij}\ f\ f' * x \notin \mathsf{dom}\ f * x' \notin \mathsf{dom}\ f'\}\ \mathit{bijinsertnew}\ \mathit{bij}\ (\iota\, x)\ (\iota\, x')\ \{\mathit{bij}'.\ \mathsf{isbij}\ \mathit{bij}'\ f[x \mapsto \iota\, x']\ f'[x' \mapsto \iota\, x]\}$$

$$\textsc{BijLookupPartialSpec} \quad \{\mathsf{isbij}\ \mathit{bij}\ f\ f'\}\ \mathit{bijlookup}\ \mathit{bij}\ (\iota\, x)\ \{x', \mathsf{ret}\ \iota\, x'.\ f\ x = \iota\, x' * f'\ x' = \iota\, x\}_?$$

Fig. 14. Finite bijection interface

to show that e, when given access to the resource R protected by *sync*'s lock, (i) returns a value satisfying Q and (ii) preserves R.

Finite maps. We specify functional finite maps in Fig. 13. The spec is parameterized by an injection $\iota : X \rightarrow \mathit{Val}$ from some type X (our examples use locations) to values. The assertion

$$\mathsf{ismap}\ \mathit{map}\ f$$

denotes knowledge that value *map* represents the finite partial function $f : X \overset{\text{fin}}{\rightharpoonup} \mathit{Val}$. MapEmptySpec says that value *mapempty* represents the empty map. MapInsertNewSpec specifies a special case of insertion. MapLookupPartialSpec specifies a partial lookup operation. The evident implementation of finite maps using association lists satisfies this spec when X enjoys decidable equality.

Finite bijections. We specify functional finite partial bijections in Fig. 14. As with finite maps, the spec is parameterized by an injection $\iota : X \rightarrow \mathit{Val}$ from some type X to values. The assertion

$$\mathsf{isbij}\ \mathit{bij}\ f\ f'$$

denotes knowledge that value *bij* represents the finite partial bijection $f, f' : X \overset{\text{fin}}{\rightharpoonup} \mathit{Val}$ in the sense that for every $x \in X$ and value v, if $f\ x = v'$, then there exists $x' \in X$ such that $v' = \iota\, x'$ and $f'\ x' = \iota\, x$. Representing a bijection as a pair of finite maps (as specified in Fig. 13) leads to a natural implementation of this specification.