

Updatable Security Views

J. Nathan Foster Benjamin C. Pierce Steve Zdancewic
University of Pennsylvania

Draft of Technical Report MS-CIS-09-05
Department of Computer and Information Science
University of Pennsylvania

Abstract

Security views are a flexible and effective means of controlling access to confidential information. Rather than allowing untrusted users to access the source data directly, they can instead be provided with a restricted view, from which all confidential information has been removed. The program that generates the view effectively embodies a confidentiality policy for the underlying source data. However, this approach has a significant drawback: it prevents users from updating the data in the view.

To address the “view update problem” in general, a number of bidirectional languages have been proposed. Programs in these languages—often called lenses—can be run in two directions: read from left to right, they map sources to views; read from right to left, they map updated views back to updated sources. However, existing bidirectional languages do not deal adequately with security issues. In particular, they do not provide a way to ensure the integrity of data in the source as it is manipulated by untrusted users of the view.

We propose a novel framework of secure lenses that addresses these shortcomings. We first enrich the types of basic lenses with equivalence relations capturing notions of confidentiality and integrity and formulate the essential security conditions on source data as non-interference properties. We then offer a concrete instantiation of our framework in the domain of string transformations, developing concrete syntax for security-annotated regular expressions as well as a collection of bidirectional string combinators with annotated expressions as their types.

1. Introduction

Security views are a widely-used mechanism for controlling access to confidential information in databases and other systems for managing structured information. By forcing users to access data via views that only expose public information, administrators ensure that secrets will not be leaked, even if the users mishandle the data or are malicious. Security views are robust, making it impossible for users to leak the data hidden by the view, and they are flexible: being implemented as arbitrary programs, security views can be used to enforce extremely fine-grained access control policies. However, they are not usually updatable—and for good reason! Propagating an update to a view back to the underlying source can, in general, alter the source, including the parts hidden by the view.

Still, there are many applications in which having a mechanism for updating security views reliably would be incredibly useful. As an example, consider Intellipedia, a collaborative data sharing system based on Wikipedia that is used by members of the intelligence community. The data stored in Intellipedia is classified at the granularity of whole documents, but many documents actually contain a mixture of highly-classified and less-classified data. In order to give users with low clearances access to the portions of documents they have sufficient clearance to see, documents often have to be regraded: i.e., the highly-classified parts need to be erased or redacted, leaving behind a residual document—a security view!—that can be reclassified at a lower level of clearance. Regrading provides fine-grained access to the information contained in documents, but naturally (since we are talking about a wiki) we would also like users to be able to make updates—e.g., to correct errors or to add new information—and have their changes be propagated back to the original document.

To support updates, the program that generates the view needs to be *bidirectional*: i.e., it must not only be able to transform sources to views but also to map updated views back to updated sources. In previous work, we, along with many others,

have proposed a family of languages for describing bidirectional transformations, often called *lenses* [17], [7], [6], [18], [19], [31], [34], [23], [8], [21], [30], [15], [20], [26], [24]. Formally, a lens l mapping between a set S of “source” structures and a set V of “views” consists of three functions

$$\begin{aligned} l.get &\in S \longrightarrow V \\ l.put &\in V \longrightarrow S \longrightarrow V \\ l.create &\in V \longrightarrow S \longrightarrow V \end{aligned}$$

satisfying the following “round-tripping” laws for every $s \in S$ and $v \in V$.

$$l.get (l.put v s) = v \quad (\text{PUTGET})$$

$$l.get (l.create v) = v \quad (\text{CREATEGET})$$

$$l.put (l.get s) s = s \quad (\text{GETPUT})$$

The *get* function defines the view and is a total function from S to V . There are two transformations that handle updates: the *put* function takes an updated V and the original S and weaves them together to yield a correspondingly modified S , while the *create* function handles the special case where we need to compute an S from an V but we have no S to use as the original (it fills in any source data not reflected in the view with defaults).

The behavioral laws obeyed by lenses capture fundamental expectations about how these three functions should work together and are closely related to the classical conditions on correct *view update translators* that have been studied in the database literature [1], [11]. The first two laws require that updates to views must be propagated “exactly”—i.e., given a view, the *put* and *create* functions must produce a source that the *get* function maps back to the very same view. The other law states that the *put* function must restore the original source when the update does not change the view.

Languages for describing these basic lenses have been extensively studied in recent years, but none of the languages that have been proposed so far deal adequately with security issues. The critical problem that they fail to address is that the natural ways of propagating many updates to views back to sources alters the hidden source data in ways that violate expectations about its integrity. For example, in the Intellipedia system, a natural way to propagate a deletion of a section from a regraded document is to delete the corresponding section from the original document. But while doing so faithfully reflects the edit made to the view—formally, it satisfies PUTGET—it is not necessarily what we want: if the section in the original document contains additional, highly-classified data in nested subsections, then deleting the whole section is almost surely unacceptable—users should not be allowed to delete data they do not even have sufficient clearance to see!

At this point, we might be tempted to add an additional behavioral law stipulating that updates to the view must not lose *any* hidden data in the source. This idea has been explored in the so-called *constant complement* approach to view update in databases [1]. The idea is that the source S should be isomorphic to $(V \times C)$, a product consisting of the view (V) and a complement (C) that contains the source information not reflected in the view. The *get* function uses one half of this isomorphism to transform the source into a pair, and then projects out the first component. The *put* function pairs up the new view and the original complement and applies the other half of the isomorphism to this pair to obtain a new source. Note that because the backward function is injective, the *put* necessarily propagates *all* of the information contained in the complement back to the source.

One can formalize a notion of lenses that hold a complement constant by imposing an additional law requiring that the effect of two *puts* in a row must be the same as just the second:

$$l.put v' (l.put v s) = l.put v' s \quad (\text{PUTPUT})$$

(For details about the relationship between this law and constant complement approaches see [17].) We call lenses that satisfy PUTPUT *very well behaved*. Unfortunately, requiring very well behavedness universally is a draconian restriction of behaviors that seem indispensable in practice. For example, it disallows certain conditional and iteration operators.

So, since we do want to allow untrusted users to modify hidden source data through the view under some circumstances, we need a simple, declarative way to specify the parts of the source that can be affected by view updates and the parts that cannot. Developing a framework in which it is possible to formulate integrity policies like “these sections in the source can be deleted” or “these sections in the view must not be altered (because doing so would have an unacceptable effect on the source)” and verify that lenses obey them is the goal of this paper.

To this end, we identify a new semantic space of *security lenses*, in which types not only describe the sets of structures that are manipulated by the components of lenses, but also capture the notion that certain parts of the source and view represent endorsed data while other parts are tainted. Semantically, we model these types as sets of structures along with equivalence relations identifying the structures that agree on high-integrity data. Syntactically, we describe them using *annotated regular types*—regular expressions decorated with labels drawn from a set of static levels of integrity. We then formulate a condition ensuring the integrity of source data by stipulating a *non-interference* property of the *put* function as an additional behavioral

law. This law ensures that, if the update to the view does not modify endorsed data in the view, then the *put* function will not modify endorsed source data.

Having presented this semantic space of secure lenses, we demonstrate its applicability by developing a security-aware variant of *Boomerang*, a bidirectional language whose primitives are based on finite-state string transductions [6]. (We choose the domain of string transformations both because it is interesting in itself—the Intellipedia example is one situation out of many where one might want an updatable view of a structured string containing a mixture of public and private information—and because it is a fairly simple setting for exploring the pragmatics of security lenses, while still offering enough structure to raise a host of issues that will also come up in richer settings. In particular, the regular-expression-based type system that we work with is powerful enough to encode non-recursive XML schemas and is closely related to the full-blown schema languages of XML transformation languages such as XQuery [5], which are based on regular tree automata.)

We present refined typing rules for Boomerang’s core string lens primitives—atomic lenses for copying, deleting, and filtering data, and lens combinators for concatenation, union, iteration, and sequential composition. The typing rules for these lenses embody an information-flow analysis that tracks dependencies between data in the source and view and ensures the behavioral laws, including non-interference. There are some interesting details since, compared with the types used in other information-flow type systems, our types can describe data schemas very precisely.

So far, we have been talking only about ensuring the integrity of source data. But confidentiality is also interesting in this context: the very reason for defining a security view is to restrict access to particular parts of the source. To the best of our knowledge, none of the previous work on security views has provided a way to formally verify that information hidden by the view adheres to a declarative confidentiality policy—the query *is* the policy. But, having already developed the technical machinery for tracking integrity, it is easy to extend it to track confidentiality as well, and we do so in our information-flow type system for Boomerang. Thus, the actual type system tracks flows of information in both directions, ensuring confidentiality in the forward direction and integrity in the reverse direction.

Our contributions can be summarized as follows:

1. We propose a semantic space of *secure lenses* that refines our previous work on lenses with a type system ensuring the confidentiality and integrity of data in the source. This provides a framework for building reliable, updatable security views.
2. We develop the syntax and semantics of *annotated regular expressions*, which describe sets of strings along with equivalence relations on those sets that encode confidentiality and integrity policies.
3. We instantiate the semantic space of secure lenses with specific *string lens combinators* based on Boomerang, and we define a bidirectional information-flow type system for these combinators based on annotated regular types.
4. We present an extension to the type system of secure lenses that also ensures the integrity of source data but replaces static constraints on lens types with more permissive dynamic tests.

2. Example

Before diving into technicalities, let’s warm up with a very small example—much smaller than the Intellipedia case study discussed in the introduction.¹

Suppose that we have an electronic calendar in which certain events, indicated by lines beginning with “*”, as well as the locations of all events, indicated by parentheses, are intended to be private.

```
*08:30 Coffee with Sara (Starbucks)
 10:00 Meeting with Brett (My office)
 12:00 PLClub Seminar (Seminar room)
*15:00 Work out (Gym)
```

Next, suppose we want to compute a security view where some of the private data is hidden—e.g., perhaps we want to redact the descriptions of the private events by marking them as *BUSY* and at the same time we want to erase the location of every event.

```
08:30 BUSY
10:00 Meeting with Brett
12:00 PLClub Seminar
```

1. Interested readers can find some prototype code for computing security views of MediaWiki documents in the Boomerang source distribution, but this simpler example makes the same essential points.

15:00 BUSY

Or perhaps we want to go a step further and erase the private events completely.

10:00 Meeting with Brett

12:00 PLClub Seminar

In either case, having generated a security view, we might like to let colleagues make changes to the public version of our schedule, to correct errors or make amendments. For example, here the user of the redacted version of the view has corrected “Brett” to “Brent” and added a meeting with Michael:

08:30 BUSY

10:00 Meeting with Brent

12:00 PLClub

15:00 BUSY

16:00 Meeting with Michael

The *put* function of the lens combines this version with the original source to produce a new view reflecting both changes:

*08:30 Coffee with Sara (Starbucks)
10:00 Meeting with Brent (My office)
12:00 PLClub (Seminar room)
*15:00 Work out (Gym)
16:00 Meeting with Michael

However, although this particular update was handled in a reasonable way, in general propagating updates can violate expectations about how the private data is handled. For example, if the update shortens the view,

08:30 BUSY

10:00 Meeting with Brent

then the source will also be truncated (as it must, to satisfy the PUTGET law):

*08:30 Coffee with Sara (Starbucks)
10:00 Meeting with Brent (My office)

From a certain perspective, this is correct—the updated view was obtained by deleting entries, so the new source should be obtained by deleting corresponding entries. But, if the owner of the source expects the lens to both hide and maintain the integrity of the hidden private data, then it is troubling that a user of the view was able to cause some of that data—the description and location of the 15:00 entry and the location of the 12:00 entry—to be lost.

As similar problem arises when the view updates a private entry to a public one. Consider a private source entry.

*15:00 Work out (Gym)

It maps via *get* to a view.

15:00 BUSY

If user of the view changes it to a public entry (here, they have insisted that an important event should take precedence over whatever private event was at 15:00)

15:00 Distinguished Lecture

then the new source loses the confidential information—the description and location—associated with the original entry, becoming:

15:00 Distinguished Lecture

As these examples demonstrate, to use lenses to reliably manage security views, we need more refined ways of tracking the integrity of source data.

First let us consider an attractive (but impossible) collection of guarantees we might like to have. Ideally, the *get* function of the lens would hide the private data—the descriptions of private events and the locations of all events—and the *put* function would take any updated view and produce an updated source where all of this hidden data is preserved. Sadly, this

is not possible: if we are going to allow the user of the view to make any update they want, including replacing private entries with public ones and deleting entries from the view, then we either need to allow the possibility that certain updates will cause hidden data in the source to be lost, or, if we insist that it must not, then we need to prevent the user from making those edits to the view in the first place.

Both alternatives can be expressed using the framework developed in this paper. To illustrate them precisely, we need a few definitions. The source and view types of the redacting lens are formed out of the regular expressions for timestamps, descriptions, and locations, and a few predefined regular expressions (NUMBER, COLON, SPACE, etc.). These are defined in Boomerang as follows:

```
let TIME : regexp =
  NUMBER{2} . COLON . NUMBER{2} . SPACE
let DESC : regexp =
  [^\n()]* - (ANY . BUSY . ANY)
let LOC : regexp =
  (SPACE . LPAREN . [^()]* . RPAREN)?
```

Boomerang uses standard POSIX notation for character sets (`[^\n()]`) and repetition (`*` and `{2}`). The `(.)` and `(-)` operators denote concatenation and difference.

To specify the policy that prevents the user from applying certain updates to the view, we pick a type (by decorating the regular expressions with annotations) that marks some of the data as endorsed. Here is a type where the private entries are endorsed, as indicated by annotations of the form $(R:E)$, but the public entries may be tainted, indicated by annotations of the form $(R:T)$:

$$\text{redact} \in ((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T \mid (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):E)^* \iff ((\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T \mid (\text{TIME}\cdot\text{BUSY}\cdot\text{NL}):E)^*$$

As described in the next section, before the owner of the source data allows an untrusted user to propagate their update using the `put` function of a secure lens they check that the new and old views agree on endorsed data. With this type, since the private entries are endorsed in the view, the user cannot modify them. They can, however, freely modify the public entries.

Alternatively, to specify the policy that allows a more liberal update policy, we pick a type that labels both kinds of entries as potentially tainted:

$$\text{redact} \in ((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T \mid (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T)^* \iff (((\text{TIME}\cdot\text{DESC}\cdot\text{NL}) \mid (\text{TIME}\cdot\text{BUSY}\cdot\text{NL})):T)^*$$

With this type, the user is allowed to make any update to the view—the view doesn't contain any endorsed data. However, it doesn't guarantee the integrity of either kind of entry in the source—to the contrary, the fact that the entire source is tainted is reflected explicitly in its type.

Here is the Boomerang code that implements these lenses. Note that all the types are unannotated—the current implementation only tracks basic types. We plan to extend it with annotated types in the immediate future.

```
let public : lens =
  del SPACE .
  copy ( TIME . DESC ) .
  del LOC .
  copy NL

let private : lens =
  del ASTERISK .
  copy TIME .
  ( ( DESC . LOC ) <-> "BUSY" ) .
  copy NL

let redact : lens =
  public* . ( private . public* )*

let erase : lens =
```

```

filter ( stype public ) ( stype private );
public*

```

In the *get* direction, the lenses can be read as ordinary string transducers, written in regular expression style. For example, the `public` lens deletes a whitespace character (`del SPACE`), copies the timestamp and event description that follows (`copy (TIME . TEXT)`), deletes an optional location (`del LOCATION?`) and copies a newline character (`copy NEWLINE`). The concatenation operator (`.`) used to combine these lenses works in the obvious way. Similarly, the `private` lens deletes an asterisk (`del ASTERISK`), copies the timestamp (`copy TIME`), redacts the event description and optional location by rewriting them to `BUSY` (`TEXT . LOCATION? <-> "BUSY"`), and copies a newline (`copy NEWLINE`). The top-level `redact` lens works by processing blocks of public entries (`public*`) interspersed with single private entries (`private`). The iteration operator `*` works by splitting the source string into a sequence of substrings and processing each using the *get* of the lens being iterated. The top-level `erase` lens uses sequential composition (`;`) and does its work in two phases: filtering away the private entries (the two uses of `stype` extract the regular expressions representing the source types of the public and private lenses), and then processing the remaining public entries.

In the *put* direction, these lenses combine an updated view with the original source, restoring the information that was hidden by the *get* function. For example, the `(copy)` lens, whose *get* function does not hide any of the source string, simply copies the updated view back to the source. The delete (`del`) and rewriting (`<->`) lenses, whose *get* functions discard the source completely, restore it in the *put* direction. The concatenation (`.`) operator it splits the source and view in two and uses the *put* functions of its sublenses to process these smaller strings. The iteration operator (`*`) is similar.²

3. Semantics

The basic lens laws `PUTGET`, `CREATEGET`, and `GETPUT` ensure some fundamental sanity conditions, but, as we saw in the examples in the previous section, to uses lenses in security settings we need some additional guarantees. This section describes a refined semantic space of *secure lenses* that has new laws ensuring that the *put* function does not taint endorsed (high integrity) source data and that the *get* function does not leak secret (high confidentiality) data.

Let \mathcal{P} (for “privacy”) and \mathcal{Q} (for “quality”) be lattices of security labels representing levels of confidentiality and integrity, respectively. To streamline the presentation, we’ll work with two-point lattices $\mathcal{P} = \{P, S\}$ (for “public” and “secret”) with $P \sqsubseteq S$ and $\mathcal{Q} = \{E, T\}$ (for “endorsed” and “tainted”) with $E \sqsubseteq T$.



Our results generalize straightforwardly to arbitrary finite lattices. (Although we call endorsed data “high integrity” informally, it is actually the least element in \mathcal{Q} . This is standard—intuitively, data that is higher in the lattice requires more careful handling while data that is lower in the lattice can be used more flexibly.)

Fix sets S (of sources) and V (of views). To formalize notions like “these two sources contain the same public information (but possibly differ on their private parts),” we suppose there are equivalence relations on S and V indexed by each of the lattices of security labels. Formally, let $\sim_k^S \subseteq S \times S$ and $\sim_k^V \subseteq V \times V$ be families of equivalence relations indexed by security labels in \mathcal{P} , and let $\approx_k^S \subseteq S \times S$ and $\approx_k^V \subseteq V \times V$ be families of equivalence relations indexed by labels in \mathcal{Q} . The S and V superscripts will be clear from context in what follows, so we will usually suppress them to lighten the notation. Typically, \sim_S and \approx_T will be equality, while \sim_P and \approx_E will be coarser relations identifying sources and views with the same public and endorsed parts respectively and will capture static confidentiality and integrity policies for the data.

A *security lens* l has three components

$$\begin{aligned} l.get &\in S \longrightarrow V \\ l.put &\in V \longrightarrow S \longrightarrow S \\ l.create &\in V \longrightarrow S \end{aligned}$$

obeying the following laws for every s in S , v in V , and k in \mathcal{Q} or \mathcal{P} as appropriate:

2. In the simple version of lenses we are using here, the *put* direction of a lens of the form l^* operates positionally, parsing the source and view into substrings belonging to the domain and codomain of l and applying $l.put$ to pairs of these in order. It is not hard to think of examples where this behavior is not what’s wanted (imagine deleting the first element of the view...), and readers familiar with Boomerang may recall that it is actually based on *dictionary lenses*, which incorporate extra mechanisms for handling ordered data. It would be interesting to investigate secure versions of dictionary lenses, but we leave this extension for future work and only consider the simpler basic string lenses here.

$$\begin{array}{r}
l.get (l.put v s) = v \quad \text{(PUTGET)} \\
l.get (l.create v) = v \quad \text{(CREATEGET)} \\
\frac{v \approx_k (l.get s)}{l.put v s \approx_k s} \quad \text{(GETPUT)} \\
\frac{s \sim_k s'}{l.get s \sim_k l.get s'} \quad \text{(GETNOLEAK)}
\end{array}$$

The PUTGET and CREATEGET laws here are identical to the basic lens version that we saw in the Introduction and express a fundamental constraint on lenses: updates made to the view must be reflected in the source.

The GETPUT law for secure lenses, however, is different. It stipulates a non-interference property which ensures the integrity of source data after the lens propagates an update to the view. Formally, it requires that if the new view and original view (computed from the source) are related by \approx_k , then the original source must also be related by \approx_k to the updated source computed by *put*. For example, if the original and new view are related by \approx_E —they agree on the endorsed data—then GETPUT ensures that the new source and the original will also agree on the endorsed data.

This suggests a protocol for safely using a security lens: before the owner of the source data allows a user of a view to invoke *put*, she checks that the original and updated views are related by \approx_k for k s lower in \mathcal{Q} than the data the user is allowed to edit—e.g., in the two-point lattice, a user whose edits are considered tainted, the check would be done using \approx_E , to ensure that they have not modified any endorsed data—and refuse to perform the *put* if this is not the case. Note that we recover the basic lens law GETPUT as a special case when \approx_k is equality, as it typically is for \approx_T .

Our main concern in this paper is preserving integrity after updates, but it is worth noticing that we also can also tell a slightly improved story about confidentiality. In previous presentations of security views (without update), the confidentiality policy enforced by the view is not stated explicitly—the private information in the source is simply “all the information that is projected away in the view.” Our security lenses, on the other hand, have an explicit representation of confidentiality policies, embodied in the choice of equivalence relations. Thus, we can add the GETNOLEAK law stipulating that the *get* function must not leak confidential information in the source. This law is a standard non-interference policy stating that, whenever two sources are related by \sim_k the results computed by *get* from those sources must also be related by \sim_k . For example, when \sim_P relates two sources, GETNOLEAK ensures that the views computed from those sources also agree on public data. Thus, secure lenses provide a confidentiality guarantee that can be understood without having to look at the program that defines the lens.³ In the next section, we present a declarative language for security annotations that can be used to describe many such equivalences.

4. Annotated Regular Expressions

The types of our secure string lens combinators are built out of regular expressions annotated with labels drawn from two lattices of security labels, one representing confidentiality and one representing integrity. In this section, we define the precise syntax and semantics of these annotated regular expressions.

Let us begin by defining a few pieces of notation. Let Σ be a finite alphabet (e.g., ASCII). A language L is a subset of Σ^* . When L is non-empty, we write $rep(L)$ for an arbitrary element of L . The ϵ symbol denotes the empty string and $u \cdot v$ the concatenation of u and v , which we lift to languages in the obvious way: $L_1 \cdot L_2 \triangleq \{u \cdot v \mid u \in L_1 \text{ and } v \in L_2\}$. The iteration of L is written L^* : i.e., $L^* = \bigcup_{n=0}^{\infty} L^n$ where L^n denotes the n -fold concatenation of L with itself.

Many of our definitions require that every string in the concatenation of two languages have a unique factorization, in the following sense. We say that a pair of languages L_1 and L_2 are unambiguously concatenable, written $L_1 \cdot^! L_2$, if for every u_1, v_1 in L_1 and u_2, v_2 in L_2 if $(u_1 \cdot u_2) = (v_1 \cdot v_2)$ then $u_1 = v_1$ and $u_2 = v_2$. Similarly, a language L is unambiguously iterable, written $L^!*$, if for every $(u_1, \dots, u_m), (v_1, \dots, v_n) \in L$, whenever $(u_1 \cdots u_m) = (v_1 \cdots v_n)$ we have $m = n$ and $u_i = v_i$ for $i \in \{1..n\}$.

4.1 Fact: It is decidable whether two regular languages are unambiguously concatenable and whether a language is unambiguously iterable (see [4, Proposition 4.1.3]).

Now we are ready to define our types. Let $\mathcal{K} = (K, \sqsubseteq)$ be a finite lattice.⁴ The set of *annotated regular expressions* over

3. Since the confidentiality and integrity equivalences are orthogonal, users can also, if they like, choose the \sim_P^S equivalence to be equality; then our laws place no constraints on confidentiality and we obtain the same story as before, where “what the view hides” is simply read off from the view definition.

4. Note that, to streamline the notation, annotations here are drawn from just *one* lattice of labels. Later, when we use these annotated regular expressions to denote the types of secure string lenses, we’ll decorate them with labels from both \mathcal{P} and \mathcal{Q} . When we calculate the semantics of a type—in particular, the equivalence relations it denotes—we will consider each lattice separately, ignoring the labels in the other lattice.

Σ and \mathcal{K} is given by the following grammar:

$$\mathcal{R} ::= \emptyset \mid u \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} | \mathcal{R} \mid \mathcal{R}^* \mid \mathcal{R} : k$$

where $u \in \sigma^*$ and $k \in K$. The semantics of an expression R has two components:

- A regular language $\mathcal{L}(R)$, defined in the standard way (ignoring annotations).
- A family of equivalence relations $\sim_k \subseteq (\mathcal{L}(R) \times \mathcal{L}(R))$ capturing the intuitive notion that two structures that differ only in high-security data cannot be distinguished by low-security observers.

In many languages with this sort of annotations, the type structure of is relatively simple, so the definition of the associated “observability relations” is straightforward. However, annotated regular expressions have features like non-disjoint unions that make the intended semantics less obvious—indeed, there seem to be several reasonable alternatives. We describe here a simple semantics based on a notion of erasing inaccessible substrings that we find natural; some alternatives are discussed toward the end of the section.

Formally, the equivalence relations are defined in terms a function that erases data inaccessible to a k -observer: two strings are equivalent if their erased versions are identical. For ease of exposition, we describe the erasing function itself as the composition of a function that marks the inaccessible regions of a string and another function that erases the marked regions.

Let $\#$ be a fresh symbol not in Σ . The helper function $hash(R)$ maps strings in $\mathcal{L}(R)$ to strings consisting entirely of $\#$ symbols:

$$hash(R)(u) \triangleq \underbrace{\# \dots \#}_{|u| \text{ times}}$$

The relation $mark(R, k)$ obscures regions of its input that are inaccessible to a k -observer by mapping characters to $\#$ (using $hash$):

$$\begin{aligned} mark(\emptyset, k) &\triangleq \{\} \\ mark(u, k) &\triangleq \{(u, u)\} \\ mark(R_1 \cdot R_2, k) &\triangleq mark(R_1, k) \cdot mark(R_2, k) \\ mark(R_1 | R_2, k) &\triangleq mark(R_1, k) \& (\mathcal{L}(R_1) \setminus \mathcal{L}(R_2)) \\ &\quad | mark(R_2, k) \& (\mathcal{L}(R_2) \setminus \mathcal{L}(R_1)) \\ &\quad | mark(R_1, k) \& mark(R_2, k) \\ mark(R_1^*, k) &\triangleq mark(R_1, k)^* \\ mark(R_1 : j, k) &\triangleq \begin{cases} mark(R_1, k) & \text{if } k \sqsupseteq j \\ hash(R_1) & \text{otherwise} \end{cases} \end{aligned}$$

We use the operations of concatenation and iteration, lifted to relations in the usual way. In the case for union, we use intersection with a regular language L , defined as

$$mark(R, k) \& L \triangleq \{(u, v) \mid u \in L \wedge (u, v) \in mark(R, k)\},$$

as well as an intersection operation on relations that marks the characters that either relation marks:

$$\begin{aligned} mark(R_1, k) \& mark(R_2, k) &\triangleq \\ \{(u, merge(v_1, v_2)) \mid & \begin{array}{l} (u, v_1) \in mark(R_1, k) \\ \wedge (u, v_2) \in mark(R_2, k) \end{array} \} \end{aligned}$$

where

$$\begin{aligned} merge(\epsilon, \epsilon) &= \epsilon \\ merge(c \cdot v_1, c \cdot v_2) &= c \cdot merge(v_1, v_2) \\ merge(\# \cdot v_1, _ \cdot v_2) &= \# \cdot merge(v_1, v_2) \\ merge(_ \cdot v_1, \# \cdot v_2) &= \# \cdot merge(v_1, v_2). \end{aligned}$$

Although we defined $mark$ as a relation, we are actually interested in cases where it denotes a function. Unfortunately, the operations of concatenation, and iteration used in the definition of $mark$ do not yield a function in general due to possible ambiguity. Thus, we impose the following condition:

4.2 Definition: R is *well-formed* iff every subexpression of the form $R_1 \cdot R_2$ is uniquely concatenable ($\mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$) and every subexpression of the form R^* is uniquely iterable ($\mathcal{L}(R)^{!*}$).

4.3 Proposition: If R is well formed, then $mark(R, k)$ is a function.

We will tacitly assume that all annotated expressions are well formed. (And when we define typing rules for our secure lens combinators, below, we will be careful to ensure well-formedness.)

Let *erase* be the function on $(\Sigma \cup \{\#\})$ that copies characters in Σ and erases $\#$ symbols. We define \sim_k as the relation induced by marking and then erasing:

$$\begin{aligned} \text{hide}(k) &\triangleq \text{erase} \circ \text{mark}(R, k) \\ \sim_k &\triangleq \{(u, v) \mid \text{hide}_k(u) = \text{hide}_k(v)\} \end{aligned}$$

It is easy to see that \sim_k is an equivalence relation.

4.4 Lemma: Let R_1 and R_2 be well-formed annotated regular expressions over a finite lattice \mathcal{K} . It is decidable whether R_1 and R_2 are equivalent.

Proof sketch: Equivalence for the regular languages $\mathcal{L}(R_1)$ and $\mathcal{L}(R_2)$ is straightforward. Moreover, each relation \sim_k is induced by $\text{hide}(-)_k$, which is definable as a rational function—a class for which equivalence is decidable [3, Chapter IV, Corollary 1.3]. \square

As examples to illustrate the semantics, consider a two-point lattice $(\{P, S\}, \sqsubseteq)$ with $P \sqsubseteq S$ and take R_1 to be the annotated expression $[\mathbf{a-z}]:S$. Then for every string s in $\mathcal{L}(R_1)$ we have $\text{mark}(R_1, P)(s) = \#$, and so $\text{hide}(P)(s) = \epsilon$, and \sim_P is the total relation. For the annotated relation R_1^* , the equivalence \sim_P is again the total relation because every s in $\mathcal{L}(R_1^*)$ maps to a sequence of $\#$ symbols by $\text{mark}(R_1^*, P)$, and so $\text{hide}(P)(s) = \epsilon$. More interestingly, for R_2 defined as

$$\begin{aligned} &([\mathbf{a-z}]:P) \cdot ([0-4]:S) \\ &| ([\mathbf{a-z}]:P) \cdot ([5-9]:S), \end{aligned}$$

and any string $c \cdot n$ in $\mathcal{L}(R_2)$ we have $\text{mark}(R_2, P)(c \cdot n) = c\#$ and so $\text{hide}(P)(c \cdot n) = c$. It follows that $cn \sim_P c'n'$ iff $c = c'$. Finally, for R_2^* the equivalence \sim_P identifies $(c_1 \cdot n_1 \cdots c_i \cdot n_i)$ and $(d_1 \cdot m_1 \cdots d_j \cdot m_j)$ iff $i = j$ and $c_i = d_i$ for i from 1 to n .

As we remarked above, there are other reasonable ways to define \sim_k . For example, we could compose *mark* with a function that compresses sequences of $\#$ symbols into a single $\#$. The equivalence induced by this function would allow low-security observers to determine the presence and location of high-security data, but would obscure its content. We could even take the equivalence induced by the *mark* function itself! This semantics would reveal the presence, location, and length of high-security data to low-security observers. There may well be scenarios where where one of these alternative semantics more accurately models the capabilities of low-security observers. However, we will stick with the erasing semantics for simplicity.

To lighten the notation in the rest of the paper, when it is clear from context we will often conflate $\mathcal{L}(R)$ and R —e.g., we will write $u \in R$ instead of $u \in \mathcal{L}(R)$.

5. Secure String Lens Combinators

Having identified a semantic space of secure lenses and defined the syntax and semantics of annotated regular expressions, we now turn to our attention defining secure versions of the core lenses in Boomerang [6].

Copy The simplest lens, *copy* E , takes a well-formed annotated regular expression as an argument. It copies strings belonging to E in both directions. Its components are defined precisely in the box below.

| |
|---|
| $\frac{E \text{ well-formed}}{\text{copy } E \in E \iff \bar{E}}$ $\begin{aligned} \text{get } s &= s \\ \text{put } v \ s &= v \\ \text{create } v &= v \end{aligned}$ |
|---|

The inference rule at the top should be read as a lemma that, assuming that E is well-formed, $(\text{copy } E)$ is a secure lens at $E \iff \bar{E}$: i.e., its components are total and obey PUTGET, CREATEGET, GETNOLEAK, and GETPUT. The proofs of all these properties can be found in the appendix.

Const The next lens, *const*, takes as arguments a well-formed annotated regular expressions E and F with F a singleton, and a string d that must belong to E . It maps every string in E to the unique element of F in the *get* direction and restores the discarded source in the reverse. The d argument is used as a default by *create* when no original source is available.

$$\boxed{
\begin{array}{c}
\frac{E, F \text{ well-formed} \quad |F| = 1 \quad d \in E}{\text{const } E \ F \ d \in E \iff F} \\
\\
\text{get } s \quad = \text{rep}(F) \\
\text{put } v \ s \ = s \\
\text{create } v \ = d
\end{array}
}$$

Usually F will be a bare string u , but occasionally it will be useful to decorate it with integrity labels (see the discussion around the union combinator below). The typing rule for const places no additional label on the source type (E) or view type (F). This is safe: the get function maps every string in E to $\text{rep}(F)$, so it discards any secrets and GETNOLEAK holds trivially. The put restores the source exactly—including any endorsed data—so GETPUT also holds. Using const , we can define a few additional lenses as derived forms:

$$\begin{array}{l}
E \leftrightarrow F \triangleq \text{const } E \ F \ \text{rep}(E) \in E \iff F \\
\text{del } E \triangleq E \leftrightarrow \epsilon \in E \iff \epsilon \\
\text{ins } F \triangleq \epsilon \leftrightarrow F \in \epsilon \iff F
\end{array}$$

$E \leftrightarrow F$ is like const but chooses an arbitrary element of E as the default; $\text{del } E$ deletes a source string belonging to E in the get direction and restores it in the put direction; $\text{ins } F$ inserts the fixed string $\text{rep}(F)$ in the get direction and removes it in the put direction.

The next few combinators build bigger lenses from smaller ones using the familiar regular operators.

Union The union combinator behaves like a conditional operator on lenses (the typing rule uses some new notation, explained below):

$$\boxed{
\begin{array}{c}
(S_1 \cap S_2) = \emptyset \\
l_1 \in S_1 \iff V_1 \quad l_2 \in S_2 \iff V_2 \\
q = \bigvee \{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \ \& \ V_2 \text{ agree}\} \\
p = \bigvee \{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\} \\
\hline
l_1 \mid l_2 \in (S_1 \mid S_2):q \iff (V_1 \mid V_2):p \\
\\
\text{get } s \quad = \begin{cases} l_1.\text{get } s & \text{if } s \in S_1 \\ l_2.\text{get } s & \text{if } s \in S_2 \end{cases} \\
\\
\text{put } v \ s \ = \begin{cases} l_1.\text{put } v \ s & \text{if } s \in S_1 \wedge v \in V_1 \\ l_2.\text{put } v \ s & \text{if } s \in S_2 \wedge v \in V_2 \\ l_1.\text{create } v & \text{if } s \in S_2 \wedge v \in (V_1 \setminus V_2) \\ l_2.\text{create } v & \text{if } s \in S_1 \wedge v \in (V_2 \setminus V_1) \end{cases} \\
\\
\text{create } v \ = \begin{cases} l_1.\text{create } v & \text{if } v \in V_1 \\ l_2.\text{create } v & \text{if } v \in (V_2 \setminus V_1) \end{cases}
\end{array}
}$$

It uses membership tests to select a lens in each direction. As is usual with conditionals, the typing rule for union needs to be designed carefully to take implicit data flows into account. In the forward direction, we need to handle implicit flows of confidential data. For example, it is not the case that the union of

$$\begin{array}{l}
l_1 \triangleq [0-4]:S \leftrightarrow A \in ([0-4]:S) \iff A \\
l_2 \triangleq [5-9]:S \leftrightarrow B \in ([5-9]:S) \iff B
\end{array}$$

has the type:

$$l_1 \mid l_2 \in ([0-4]:S \mid [5-9]:S) \iff (A \mid B)$$

The get function leaks information about which branch was selected, as demonstrated by a counterexample to GETNOLEAK:

$$0 \sim_P 5 \quad \text{but} \quad (l_1 \mid l_2).\text{get } 0 = A \not\sim_P B = (l_1 \mid l_2).\text{get } 5$$

The usual way to deal with these implicit information flows is to escalate the label on the branches with the label of the data used in the conditional test. Our typing rule for union is based on this idea, although the computation of the label is

somewhat complicated because the conditional test is membership in S_1 or S_2 ! Thus, “the label of the data used in the conditional test” is the smallest label that can distinguish strings in S_1 from those in S_2 .

Fortunately, we can compute this label from annotated regular expressions. Let k be a label. We say k *observes* $(S_1 \cap S_2 = \emptyset)$ iff for every string $s_1 \in S_1$ and $s_2 \in S_2$ we have $s_1 \not\sim_k s_2$. Note that k observes $(S_1 \cap S_2) = \emptyset$ iff the codomains of the rational function $hide(k)$ for S_1 and S_2 are disjoint. As the codomain of a rational function is computable and regular, it follows that we can decide whether k observes disjointness.

In a general finite lattice, there may be several labels that observe disjointness. The label p we compute for the view type is the join of the minimal set of such labels. For example, in the two point lattice and the lens $l_1 | l_2$, we compute S for p .

In the reverse direction, we need to consider the integrity of the source data. The key issue here is that modifying the view can cause l_2 to be used for *put* even though l_1 was used for *get* (or vice versa). As a result, we need to escalate the integrity label on the type of the source. To see why, consider the union of:

$$\begin{aligned} l_1 &\triangleq (del [0-4]:E) \cdot (copy [A-Q]:T) \\ &\in ([0-4]:E \cdot [A-Q]:T) \iff ([A-Q]:T) \\ l_2 &\triangleq (del [5-9]:E) \cdot (copy [F-Z]:T) \\ &\in ([5-9]:E \cdot [F-Z]:T) \iff ([F-Z]:T) \end{aligned}$$

This lens does not have secure lens type obtained by taking the union of the source and view types

$$\begin{aligned} (l_1 | l_2) &\in ([0-4]:E \cdot [A-Q]:T) | ([5-9]:E \cdot [F-Z]:T) \\ &\iff ([A-Q]:T | [F-Z]:T) \end{aligned}$$

because *put* sometimes fail to maintain the integrity of the number in the source, as demonstrated by the following counterexample to GETPUT:

$$\begin{aligned} Z &\approx_E A = (l_1.l_2).get\ 0A \\ \text{but } (l_1 | l_2).put\ Z\ 0A &= 5Z \not\approx_E 0A \end{aligned}$$

To fix this hole and obtain a correct typing rule for union, we need to escalate the integrity label on the source side. We compute q as the join of the minimal set of labels that observe that (the languages denoted by) V_1 and V_2 are not identical—e.g., for the lens above, T . For technical reasons—i.e., to ensure that $v \in V_1$ and $s \in S_1$ and $v \approx_k^{S_1|S_2} (l_1 | l_2).get\ s$ implies $v \approx_k^{S_1} l_1.get\ s$ —we also require that q observe that V_1 and V_2 denote the same equivalence relations on strings in their intersection, written as V_1 & V_2 agree. This property can also be decided by an elementary construction.

An important special case arises when V_1 and V_2 coincide. Since both lenses handle the entire view type, the same lens is always selected for *put* that was selected for *get*. For example when

$$\begin{aligned} l_1 &\triangleq (del [0-4]:E) \cdot (copy [A-Z]:T) \\ &\in ([0-4]:E \cdot [A-Q]:T) \iff ([A-Z]:T) \\ l_2 &\triangleq (del [5-9]:E) \cdot (copy [Z-Z]:T) \\ &\in ([5-9]:E \cdot [F-Z]:T) \iff ([A-Z]:T) \end{aligned}$$

then the union has type

$$\begin{aligned} (l_1 | l_2) &\in ([0-4]:E \cdot [A-Z]:T) | ([5-9]:E \cdot [A-Z]:T) \\ &\iff ([A-Z]:T | [A-Z]:T) \\ \text{i.e., } &([0-4]:E \cdot [A-Z]:T) | ([5-9]:E \cdot [A-Z]:T) \\ &\iff [A-Z]:T \end{aligned}$$

Our typing rule captures this case—if $V_1 = V_2$ then q is the join of the empty set, i.e., the minimal element E , which is semantically equivalent to having no additional integrity annotation on the source.

Concatenation The concatenation operator takes two smaller lenses and forms a bigger lens that operates on the concatenations of their types.

$$\begin{array}{c}
l_1 \in S_1 \iff V_1 \quad S_1 \cdot^! S_2 \\
l_2 \in S_2 \iff V_2 \quad V_1 \cdot^! V_2 \\
q = \bigvee \{k \mid k \text{ min obs. } V_1 \cdot^! V_2\} \\
p = \bigvee \{k \mid k \text{ min obs. } S_1 \cdot^! S_2\} \\
\hline
l_1 \cdot l_2 \in (S_1 \cdot S_2) : q \iff (V_1 \cdot V_2) : p \\
\\
\text{get } (s_1 \cdot s_2) \quad \quad \quad = (l_1.\text{get } s_1) \cdot (l_2.\text{get } s_2) \\
\text{put } (v_1 \cdot v_2) (s_1 \cdot s_2) = (l_1.\text{put } v_1 \ s_1) \cdot (l_2.\text{put } v_2 \ s_2) \\
\text{create } (v_1 \cdot v_2) \quad \quad \quad = (l_1.\text{create } v_1) \cdot (l_2.\text{create } v_2)
\end{array}$$

When we write $(s_1 \cdot s_2)$ we mean that s_1 and s_2 are the unique substrings (as the source types are unambiguously concatenable) belonging to S_1 and S_2 respectively. We use this convention throughout the paper.

The typing rule for concatenation needs to be designed carefully to take implicit flows of information due to the way that the concatenation lens splits strings into account. As an example, consider a lens l_1 that maps $a0$ to A and $a1$ to a , and a similar lens l_2 that $b0$ to B and $b1$ to b , where the source data in each lens is all private, except for the 1 , which is public:

$$\begin{array}{l}
l_1 \triangleq ((a:S) \cdot (1:P) \leftrightarrow A) \mid ((a:S) \cdot (0:S) \leftrightarrow a) \in (a:S \cdot (0:S \mid 1:P)) \iff (A \mid a) \\
l_2 \triangleq ((b:S) \cdot (1:P) \leftrightarrow B) \mid ((b:S) \cdot (0:S) \leftrightarrow b) \in (b:S \cdot (0:S \mid 1:P)) \iff (B \mid b)
\end{array}$$

The concatenation of l_1 and l_2 does not have a type obtained by concatenating their source and view types

$$l_1 \cdot l_2 \in ((a:S \cdot (0:S \mid 1:P)) \cdot (b:S \cdot (0:S \mid 1:P))) \iff ((A \mid a) \cdot (B \mid b))$$

because the *get* function exposes how the source string is split, as demonstrated by a counterexample to GETNOLEAK:

$$a1b0 \sim_P a0b1 \quad \text{but} \quad (l_1 \cdot l_2).\text{get } a1b0 = Ab \not\sim_P aB = (l_1 \cdot l_2).\text{get } a0b1$$

The way we deal with this information flow is to escalate the label on the view type with the label that observes the unambiguous concatenation of the source data. Formally, we say that k observes $S_1 \cdot^! S_2$ if and only if the codomains of the rational function $hide(k)$ for S_1 and S_2 are unambiguously concatenable.

In the reverse direction, an analogous problem arises with integrity. We escalate the label on the source type with a label that observes the unambiguous concatenation of the data in the view.

Iteration The next combinator iterates a lens:

$$\begin{array}{c}
l \in S \iff V \quad S^{!*} \quad V^{!*} \\
q = \bigvee \{k \mid k \text{ min obs. } V^{!*}\} \\
p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \\
\hline
l^* \in (S^*) : q \iff (V^*) : p \\
\\
\text{get } (s_1 \cdots s_n) \quad \quad \quad = (l.\text{get } s_1) \cdots (l.\text{get } s_n) \\
\text{put } (v_1 \cdots v_n) (s_1 \cdots s_m) = s'_1 \cdots s'_n \\
\quad \text{where } s'_i = \begin{cases} l.\text{put } v_i \ s_i & i \in \{1, \dots, \min(m, n)\} \\ l.\text{create } v_i & i \in \{m+1, \dots, n\} \end{cases} \\
\text{create } (v_1 \cdots v_n) \quad \quad \quad = (l.\text{create } v_1) \cdots (l.\text{create } v_n)
\end{array}$$

As with union, we need to escalate the confidentiality label (p) on the view side and the integrity label (q) on the source side. As an example, consider:

$$l \triangleq A:S \leftrightarrow B:P \in A:S \iff B:P$$

It is not the case that

$$l^* \in (A:S)^* \iff (B:P)^*$$

as demonstrated by the following counterexample to GETNOLEAK

$$AAA \sim_P AA \quad \text{but} \quad l^*.\text{get } AAA = BBB \not\sim_P BB = l^*.\text{get } BB$$

The problem is that *get* leaks the length of the source string, but the source type asserts that it is secret. We can fix this hole by escalating the confidentiality label by the join of the minimal label that observes that S is unambiguously iterable.

Formally, we say that k observes $R^{!*}$ if and only if the codomain of the rational function $hide(k)$ for R is unambiguously iterable.

For similar reasons, it is not the case that the iteration of

$$l \triangleq [0-9]:E \leftrightarrow A: T \in [0-9]:E \iff A: T$$

has type

$$l^* \in ([0-9]:E)^* \iff (A: T)^*$$

as demonstrated by the following counterexample to GETPUT

$$A \sim_E AAA = *.get\ 123 \quad \text{but} \quad l^*.put\ A\ 123 = 1 \not\sim_P 123$$

Here the problem is that the update shortens the length of the view, which causes the iteration operator to discard endorsed data in the source. As above, we can fix this hole by escalating the integrity label by the join of the minimal label that observes that S is unambiguously iterable.

Sequential Composition The composition operator places two lenses in sequence:

| |
|---|
| $\frac{l_1 \in S \iff T \quad l_2 \in T \iff V}{l_1; l_2 \in S \iff V}$ |
| $get\ s = l_2.get\ (l_1.get\ s)$ |
| $put\ v\ s = l_1.put\ (l_2.put\ v\ (l_1.get\ s))\ s$ |
| $create\ v = l_1.create\ (l_2.create\ v)$ |

In the forward direction it applies the *get* of l_1 followed by the *get* of l_2 , and in the reverse direction it applies the *put* of l_2 followed by the *put* of l_1 (using l_1 's *get* to generate a T for l_2 's *put*). The typing rule requires that the view type of the first lens and the source type of the second be identical. This is essential for ensuring the secure lens laws.

Filter The *filter* lens takes as arguments two well-formed annotated regular expressions E and F , which must be disjoint.

Its *get* function is a standard filtering function, which takes a string of E s and F s and discards the F s, and its *put* function is an unfiltering function that weaves together a view consisting of E s with the original E s and F s from the source, using the positions of the F s to determine where to restore F s:

$$\begin{aligned} \text{let rec filter } E\ xs &= \text{match } xs \text{ with} \\ &| \epsilon \rightarrow \epsilon \\ &| x \cdot xs' \rightarrow \text{if } x \in E \text{ then } x \cdot (\text{filter } E\ xs') \text{ else } (\text{filter } E\ xs') \\ \\ \text{let rec unfilter } F\ es\ xs &= \text{match } es, xs \text{ with} \\ &| \epsilon, _ \rightarrow \text{filter } F\ es \\ &| e \cdot es', x \cdot xs' \rightarrow \\ &\quad \text{if } x \in F \text{ then } x \cdot (\text{unfilter } es\ xs') \\ &\quad \text{else } e \cdot (\text{unfilter } es'\ xs') \end{aligned}$$

It is tempting to want to define *filter* as $(copy\ E \mid del\ F)^*$ but our type system disallows this grouping of combinators—the view type of lens being iterated is not unambiguously iterable as it contains ϵ —and, moreover, its *put* function would not have the same behavior as *unfilter*, which always restores all the F s from the source. But it is no problem to have it as a primitive lens:

| |
|--|
| $\begin{aligned} E, F \text{ well-formed} \quad E \cap F = \emptyset \quad (E \mid F)^{!*} \\ q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\} \\ p \sqsupseteq \bigvee \{k \mid k \text{ observes } E.^!F \text{ and } F.^!E\} \\ \hline filter\ E\ F \in (E:q \mid F:p)^* \iff E^* \end{aligned}$ |
| $get\ (s_1 \cdots s_n) = \text{filter } E\ (s_1 \cdots s_n)$ |
| $put\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) = \text{unfilter } F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)$ |
| $create\ (v_1 \cdots v_n) = (v_1 \cdots v_n)$ |

The typing rule for *filter* captures the fact that none of the F s are leaked to the view and so can be assigned any confidentiality label we like—e.g., S . Since observers with clearance lower than p cannot distinguish source strings that differ only in the F s, the proof of GETNOLEAK is simple: two source strings are related by \sim_P exactly when their filterings—i.e., the views computed by *get*—are related by \sim_P . In the reverse direction, we need to escalate the integrity label on the E s by the join of the minimal label that observes that E is unambiguously iterable. However, the F s are restored exactly, so their label does not need to be escalated.

Subsumption Finally, we note that secure lenses admit a rules of subsumption:

$$\boxed{\frac{l \in S \iff V \quad q \in Q \quad p \in \mathcal{P}}{l \in (S:q) \iff (V:p)}}$$

This allows us to escalate the integrity label on the source type and the confidentiality label on the view type. While it may seem silly to escalate labels arbitrarily, this can be useful—e.g., to make the source and view types agree when forming the sequential composition of two lenses.

6. Dynamic Very Well Behavedness

Recall the PUTPUT law, discussed in the Introduction:

$$l.put\ v' (l.put\ v\ s) = l.put\ v'\ s \quad (\text{PUTPUT})$$

It provides a strong integrity guarantee, ensuring that the *put* function will not discard hidden source data. However, it rules out many useful transformations—e.g., lenses built using union and iteration do not obey it in general. Secure lenses obey a variant of PUTPUT, which can be derived from the GETPUT and PUTGET laws:

$$\frac{v' \approx_k l.get\ s \approx_k v}{l.put\ v' (l.put\ v\ s) \approx_k l.put\ v'\ s} \quad (\text{PUTPUTENDORSED})$$

Rather than insisting that *all* hidden source data be preserved, PUTPUTENDORSED only requires that the high-integrity parts, as identified by \approx_k , be preserved. This allows the *put* functions of lenses to produce tainted results, as long as they indicate that they do so in their type.

Using the static type system to track tainting of source data is effective, but coarse—it can require marking large regions of the source type as tainted. In this section, we explore a different idea: augmenting lenses with dynamic tests that check, for a particular view and source, whether the lens can preserve the integrity of the source data. This will ensure the integrity of source data while letting us assign more relaxed typing rules to many lenses.

Formally, we fix a set of $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{Q}$ representing static clearances for accessing and editing data. A clearance (j, k) represents the capabilities of a user of the view who may access data at confidentiality level j and whose edits taint data at level k . We extend security lenses with an new function

$$l.safe \in \mathcal{C} \longrightarrow \mathcal{L}(V) \longrightarrow \mathcal{L}(S) \longrightarrow \mathbb{B}$$

that returns *true* when a user with a given clearance can safely *put* a view and source back together and replace the hypothesis that $v \approx_k s$ in GETPUT with a version that has *safe* as a hypothesis instead, requiring that forall $(j, k) \in \mathcal{C}$ and $s \in S$ and $v \in V$:

$$\frac{l.safe\ (j, k)\ v\ s}{l.put\ v\ s \approx_k s} \quad (\text{GETPUT})$$

The revised protocol for making updates to the view is as follows: before the owner of the source data allows a user of a view to invoke *put*, she checks that the original and updated views are safe.

Adding arbitrary *safe* functions gives us some extra flexibility in designing lens primitives, but it raises an interesting problem: the *safe* function itself can leak information about source data.⁵ We deal with this by adding another law stating *safe* does not leak confidential information, formulated as a non-interference property for all $(j, k) \in \mathcal{C}$ and s and s' in S and v and $v' \in V$:

$$\frac{v \sim_j v' \quad s \sim_j s'}{l.safe\ (j, k)\ v\ s = l.safe\ (j, k)\ v'\ s'} \quad (\text{SAFENOLEAK})$$

5. In fact, the same problem arises with the secure lenses presented in the previous sections—testing whether an updated view preserves the endorsed data in the original view can also leak confidential data in the source. We address the interaction between confidentiality and integrity here.

For technical reasons (to prove that the *safe* component of the sequential composition operator, which is defined in terms of the *put* function of one of its sublenses, satisfies SAFENOLEAK) we also need a law stating that the *put* function is non-interfering for all $(j, k) \in \mathcal{C}$ and s and $s' \in S$ and v and $v' \in V$:

$$\frac{v \sim_j v' \quad s \sim_j s' \quad l.\mathit{safe}(j, k) v s \quad l.\mathit{safe}(j, k) v s'}{l.\mathit{put} v s \sim_j l.\mathit{put} v' s'} \quad (\text{PUTNOLEAK})$$

With these refinements, we can now state revised types for our lenses.

For *copy* we pick a *safe* function that—just like the original version—checks whether the new view and original source agree on k -integrity data.

| |
|---|
| $\frac{E \text{ well-formed} \quad \forall (j, k) \in \mathcal{C}. \sim_j \subseteq \approx_k}{\mathit{copy} E \in E \iff E}$ $\mathit{safe}(j, k) v s = v \approx_k s$ |
|---|

To ensure that *safe* does not leak information, we require that \sim_j refine \approx_k for every $(j, k) \in \mathcal{C}$.

For *const*, since the view type is a singleton set, there is only one possible update: a no-op. Hence, we can choose a *safe* function that always returns *true*.

| |
|--|
| $\frac{E, F \text{ well-formed} \quad F = 1 \quad d \in E}{\mathit{const} E F d \in E \iff F}$ $\mathit{safe}(j, k) v s = \mathit{true}$ |
|--|

For the concatenation operator, the *safe* function tests if the two substrings of the source and view are safe for l_1 and l_2 respectively.

| |
|--|
| $\frac{l_1 \in S_1 \iff V_1 \quad S_1 \cdot^! S_2 \quad l_2 \in S_2 \iff V_2 \quad V_1 \cdot^! S_2 \quad p = \bigvee \{k \mid k \text{ min obs. } S_1 \cdot^! S_2\}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \iff (V_1 \cdot V_2) : p}$ $\mathit{safe}(j, k) v_1 \cdot v_2 s_1 \cdot s_2 =$ $j \text{ observes } S_1 \cdot^! S_2 \text{ and } V_1 \cdot^! V_2$ $\wedge l_1.\mathit{safe}(j, k) v_1 s_1 \wedge l_2.\mathit{safe}(j, k) v_2 s_2$ |
|--|

To ensure that *safe* does not leak information, we require that it observe the unambiguous concatenation of the source and view types.

For union, we test if the source and view can be processed by the same lens. Additionally, because *safe* can be used to determine whether the source came from S_1 or S_2 , we require that j observe their disjointness (and the fact that V_1 & V_2).

| |
|---|
| $\frac{(S_1 \cap S_2) = \emptyset \quad l_1 \in S_1 \iff V_1 \quad l_2 \in S_2 \iff V_2 \quad p = \bigvee \{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\}}{l_1 \mid l_2 \in (S_1 \mid S_2) \iff (V_1 \mid V_2) : p}$ $\mathit{safe}(j, k) v s =$ $j \text{ observes } (S_1 \cap S_2) = \emptyset \text{ and } V_1 \text{ \& } V_2 \text{ agree}$ $\wedge \begin{cases} l_1.\mathit{safe}(j, k) v s & \text{if } v \in V_1 \wedge s \in S_1 \\ l_2.\mathit{safe}(j, k) v s & \text{if } v \in V_2 \wedge s \in S_2 \\ \mathit{false} & \text{otherwise} \end{cases}$ |
|---|

For iteration, *safe* checks that the view is the same length as the one generated from the source. Because *safe* can be used to determine the length of the source we require that j observe the unambiguous concatenation of S and V .

$$\begin{array}{c}
l \in S \iff V \\
\frac{p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\}}{l^* \in S^* \iff (V^*):p} \\
\text{safe } (v_1 \cdots v_n) (s_1 \cdots s_m) = \\
j \text{ observes } S^{!*} \text{ and } V^{!*} \\
\wedge n = m \wedge l.\text{safe } v_i s_i \text{ for } i \in \{1, \dots, n\}
\end{array}$$

For sequential composition, we only require the safety conditions implied by l_1 on the view computed by l_2 's *put* and the original source.

$$\begin{array}{c}
\frac{l_1 \in S \iff T \quad l_2 \in T \iff V}{l_1; l_2 \in S \iff V} \\
\text{safe } s v = l_1.\text{safe } (l_2.\text{put } v (l_1.\text{get } s)) s
\end{array}$$

Note that the composition operator is the reason that we stipulate **PUTNOLEAK** and that **SAFENOLEAK** requires that the *safe* function be non-interfering in both its source and view arguments (rather than just its source argument). We could relax these by only requiring **PUTNOLEAK** of lenses used as the second argument to a composition operator and the full version of **SAFENOLEAK** of lenses used as the first argument. This would give us yet more flexibility in designing *safe* functions (at the cost of a more complicated type system—we need have several different kinds of lens types). We leave it as future work.

Finally, the *safe* function for the filter lens checks that the new view and filtered source agree on k -endorsed data.

$$\begin{array}{c}
E, F \text{ well-formed} \quad E \cap F = \emptyset \quad (E \mid F)^{!*} \\
p \sqsupseteq \bigvee \{k \mid k \text{ observes } E.^!F \text{ and } F.^!E\} \\
\forall (j, k) \in \mathcal{C}. \sim_j^E \subseteq \approx_k^E \\
\hline
\text{filter } E F \in (E \mid F:p)^* \iff E^* \\
\text{safe } (j, k) (v_1 \cdots v_n) (s_1 \cdots s_m) = \\
j \text{ observes } E.^!F \text{ and } F.^!E \\
\wedge j \text{ and } k \text{ observe } E^{!*} \\
\wedge (v_1 \cdots v_n) \approx_k (\text{filter } E (s_1 \cdots s_m))
\end{array}$$

To ensure that *safe* does not leak information about the source, j is required to observe the way the way that E s and F s are split in the source, and the unambiguous iterability of E .

These typing rules illustrate how dynamic tests can be incorporated into the secure lens framework, providing relaxed typing rules for many of our secure string lens combinators. We plan to investigate this idea further in future work by exploring other combinations of *safe* functions and static types for our combinators.

7. Related Work

Views have long been used to enforce security boundaries in systems based on relational databases. They were first discussed in the context of XML data by Stoica and Farkas [32], and they were later studied extensively by Fan and his colleagues [12], [13], [14]. The main difference between previous work on security views and the ones developed in this paper is, of course, that previous systems do not support updates. Less critically, they also do not come with any way to formally characterize the data kept confidential by the view—the query that defines the view hides whatever data it does and exposes everything else. Finally, their views are virtual whereas the views constructed using lenses are materialized.

This paper extends previous work by the authors and others on lenses [17], [7], [6], [18]. A number of other bidirectional languages have also been proposed [19], [31], [34], [23], [8], [21], [30], [15], [20], [26], [24]. The original lens paper includes an extensive survey of the relationship between lenses and the view update problem in the database literature [17].

Many general-purpose languages with information-flow type systems have been proposed, famously including Flow-Caml [29] and Jif [27]. Very little work, however, has focused on applying information flow in the context of languages specifically designed for querying and transforming data. The developers of CDuce proposed a system for tracking information flow that uses a semantics that propagates security labels dynamically [2]. Cheney, Ahmed, and Ucar have introduced

a general framework for comparing static and dynamic approaches to many different kinds of dependency information including information flow [9]. Recent work by Foster, Green, and Tannen proposes a dynamic access control mechanism based on provenance metadata [16].

Another security-oriented language that propagates labels dynamically is Fable [33], [10]. Fable does not fix a particular semantics for label propagation, but instead provides a general framework that enforces a strict boundary between ordinary program code, which must treat labels opaquely, and security code, which may manipulate labels freely. Thus, it can be used to implement a variety of static and dynamic approaches to tracking information flows in programs.

Integrity issues have been studied in the context of data integration by Miklau and Suciu [25].

XACL and XACML are specification languages for access control policies for XML data [22]. We plan to investigate the relationship between these policy languages and our annotated types in the future.

Finally, Galois Inc. has developed a system for managing “tearline” security views of MediaWiki documents [28]. This system has helped motivate and inform our thinking about the Intellipedia example discussed in the introduction.

8. Future Work

The ideas we have described can be extended in several directions.

First, we are in the process of adding all of the features described in this paper to the Boomerang implementation. We can already use Boomerang to develop lens programs that define updatable security views, since the functional components of the secure lens combinators and their basic lens versions are identical. However, the type system in the current implementation only tracks regular types. To bring it up to speed, we need to implement annotated regular types, which in turn will require implementing a library for representing rational functions and deciding properties like equivalence.

We have presented both a static type system for secure lenses and an approach using dynamic very well behaved lenses. We would like to explore connections with other dynamic approaches—e.g., languages that propagate dynamic labels and provenance metadata. We hope that these languages will suggest mechanisms for enforcing security properties at finer levels of granularity than our current, static, approach can track.

We would also like to further explore some of the alternative semantics for annotated regular types that we mentioned in Section 3. We have chosen to work with the erasing semantics in this paper because it seems simplest, but we expect there will be applications where the redacting semantics is a more natural fit. In particular, we are interested in relations for integrity that capture the notion that a modified string *extends* another.

Fan [12] has argued that it is not practical to materialize security views, due to the large number of views that need to be constructed when the security policy is complicated. We believe that materialized views will also be practical for some applications, like the regrading example with the Intellipedia system, where views must be materialized since they need to be sent over the network and displayed in a web browser—but we are also interested in developing variants of secure lenses for non-materialized views.

Finally, we would like to develop security-annotated type systems and secure lenses in other settings besides strings—e.g., trees, relations, and graphs. For relations, the bidirectional language proposed by Bohannon et al. [7] should be a good starting point.

Acknowledgments We are grateful to Kathleen Fisher, Mike Hicks, Andrew Myers, Jeff Vaughan, and Geoff Washburn for stimulating discussions and to Philip Wadler for suggesting that we investigate dynamic very well behavedness. Our work is supported by the National Science Foundation under grants IIS-0534592 *Linguistic Foundations for XML View Update*, and CT-0716469 *Manifest Security*.

References

- [1] François Bancilhon and Nicolas Spyrtos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.
- [2] Véronique Benzaken, Marwan Burelle, and Giuseppe Castagna. Information flow security for XML transformations. In *Advances in Computing Science: Programming Languages and Distributed Computation (ASIAN), Mumbai, India*, volume 2896 of *Lecture Notes in Computer Science*, pages 33–53, 2003.
- [3] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Verlag, 1979.
- [4] Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*. Cambridge University Press, 2009. To appear. Manuscript available from <http://www-igm.univ-mlv.fr/~berstel/LivreCodes/>.

- [5] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C, January 2007. Available from <http://www.w3.org/TR/xquery>.
- [6] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, San Francisco, CA, pages 407–419, January 2008.
- [7] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS)*, Chicago, IL, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.
- [8] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. Short version in DBPL '05.
- [9] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In *Symposium on Database Programming Languages (DBPL)*, Vienna, Austria, pages 138–152, 2007.
- [10] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Combining provenance and security policies in a web-based document management system. In *On-line Proceedings of the Workshop on Principles of Provenance (PrOPr)*, Edinburgh, Scotland, November 2007. <http://homepages.inf.ed.ac.uk/jcheney/propr/>.
- [11] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, September 1982.
- [12] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure XML querying with security views. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Paris, France, pages 587–598, 2004.
- [13] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. SMOQE: A system for providing secure access to XML. In *International Conference on Very Large Data Bases (VLDB)*, Seoul, Korea, pages 1227–1230, September 2006.
- [14] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting regular XPath queries on XML views. In *International Conference on Data Engineering (ICDE)*, Istanbul, Turkey, pages 666–675, April 2007.
- [15] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, pages 295–304, 2005.
- [16] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS)*, Vancouver, BC, pages 271–280, June 2008.
- [17] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. Short version in POPL '05.
- [18] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Victoria, BC, pages 383–395, September 2008.
- [19] Michael Greenberg and Shriram Krishnamurthi. Declarative Composable Views, May 2007. Undergraduate Honors Thesis, Brown University.
- [20] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2), June 2008. Short version in PEPM '04.
- [21] Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Portland, Oregon, pages 201–214, 2006.
- [22] Michiharu Kudo and Satoshi Hada. Xml document security based on provisional authorization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Athens, Greece, pages 87–96, November 2000.
- [23] David Lutterkort. Augeas—A configuration API. In *Linux Symposium*, Ottawa, ON, pages 47–56, 2008.
- [24] Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from <ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps>.
- [25] Gerome Miklau and Dan Suciu. Managing integrity for data exchanged on the web. In *Workshop on the Web & Databases (WebDB)*, Baltimore, Maryland, pages 13–18.

- [26] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.
- [27] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, San Antonio, TX, pages 228–241, 1999.
- [28] Isaac Potoczny-Jones. Tearline Wiki: Information collaboration across security domains, 2008. Whitepaper. Presented at the Navy Opportunity Forum.
- [29] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
- [30] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *International Workshop Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
- [31] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Nashville, TN, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007.
- [32] Andrei Stoica and Csilla Farkas. Secure XML views. In *IFIP WG 11.3 International Conference on Data and Applications Security (DBSEC)*, Cambridge, UK, pages 133–146, 2002.
- [33] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 369–383, May 2008.
- [34] Janis Voigtländer. Bidirectionalization for free! In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, pages 165–176, January 2009.

Appendix

This appendix contains proofs that secure lens presented in this paper inhabits its declared type. For each lens, we repeat its type, state the lemma explicitly, and then give the proof.

1. Electronic Calendar Lens Proofs

A.1 Lemma:

$$\text{redact} \in ((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T \mid (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):E)^* \\ \iff ((\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T \mid (\text{TIME}\cdot\text{BUSY}\cdot\text{NL}):E)^*$$

Proof: Suppose that we have annotated some of the regular expressions in the `redact` lens with security labels indicating that the data handled by the `public` lens is tainted:

```
let public : lens =
  del ( SPACE:T ) .
  copy ( ( TIME . DESC ):T ) .
  del ( LOC:T ) .
  copy ( NL:T )

let private : lens =
  del ASTERISK .
  copy ( TIME ) .
  ( ( DESC . LOC ) <-> "BUSY" ) .
  copy NL

let redact : lens =
  public* . ( private . public* )*
```

We do not explicitly add annotations for E data since every regular expression R is equivalent to $R:E$ in the two-point integrity lattice. By the typing rules for `del`, `copy`, `<->`, and concatenation we have:

$$\text{public} \in (\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T \\ \iff (\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T \\ \text{private} \in (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}) \\ \iff (\text{TIME}\cdot\text{BUSY}\cdot\text{NL})$$

The syntactic type that would be computed mechanically using our typing rules is slightly more complicated but semantically equivalent; we use such equivalences throughout this proof—e.g., between $((R:T)\cdot(S:T)):T$ and $(R\cdot S):T$.

By the typing rule for iteration, using the equivalence between $(R:T)^*:T$ and $(R\cdot T)^*$, we have:

$$\text{public*} \in (((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T))^* \\ \iff (((\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T))^*$$

Next, by the typing rule for concatenation, and as E observes the unambiguous concatenation of the types in the view, we have:

$$\text{private} . \text{public*} \in (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL})\cdot(((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T))^* \\ \iff (\text{TIME}\cdot\text{BUSY}\cdot\text{NL})\cdot(((\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T))^*$$

Then, by the typing rule for iteration, as E observes the unambiguous iteration of the types in the view, we have:

$$(\text{private} . \text{public*})^* \in (((\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL})\cdot(((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T))^*))^* \\ \iff (((\text{TIME}\cdot\text{BUSY}\cdot\text{NL})\cdot(((\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T))^*))^*$$

Finally, by the typing rule for concatenation, and again as E observes the unambiguous concatenation of the types in the view, we have:

$$\text{public*} . (\text{private} . \text{public*})^* \in (((((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL})^*):T))^* \cdot \\ (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL})\cdot(((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOC}\cdot\text{NL}):T))^*)^* \\ \iff (((((\text{TIME}\cdot\text{DESC}\cdot\text{NL})^*):T))^* \cdot \\ (\text{TIME}\cdot\text{BUSY}\cdot\text{NL})\cdot(((\text{TIME}\cdot\text{DESC}\cdot\text{NL}):T))^*))^*$$

The equivalent type stated in the lemma can be obtained using the equivalence between $(R:T)^*\cdot(S\cdot(R:T))^*$ and $((R:T) \mid S)^*$ that holds when R and S are disjoint and unambiguously iterable. \square

2. Secure String Lens Combinators Proofs

$$\boxed{\frac{E \text{ well-formed}}{\text{copy } E \in E \iff E}}$$

A.2 Lemma: $\text{copy } E \in E \iff E$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $e, e' \in E$ with $e' \approx_k (\text{copy } E).\text{get } e$. We calculate as follows

$$\begin{aligned} & (\text{copy } E).\text{put } e' e \\ &= e' && \text{by definition of } (\text{copy } E).\text{put} \\ \approx_k & (\text{copy } E).\text{get } e && \text{by assumption} \\ &= e && \text{by definition of } (\text{copy } E).\text{get} \end{aligned}$$

and obtain the required equivalence.

► **PutGet:** Let $e, e' \in E$. We calculate as follows

$$\begin{aligned} & (\text{copy } E).\text{get } ((\text{copy } E).\text{put } e' e) \\ &= (\text{copy } E).\text{get } e' && \text{by definition of } (\text{copy } E).\text{put} \\ &= e' && \text{by definition of } (\text{copy } E).\text{get} \end{aligned}$$

and obtain the required equality.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $e, e' \in E$ with $e' \sim_j e$. We calculate as follows

$$\begin{aligned} & (\text{copy } E).\text{get } e \\ &= e && \text{by definition of } (\text{copy } E).\text{get} \\ \sim_j & e' && \text{by assumption} \\ &= (\text{copy } E).\text{get } e' && \text{by definition of } (\text{copy } E).\text{get} \end{aligned}$$

and obtain the required equivalence, which completes the proof. \square

$$\boxed{\frac{E, F \text{ well-formed} \quad |F| = 1 \quad d \in E}{\text{const } E F d \in E \iff F}}$$

A.3 Lemma: $\text{const } E F v \in E \iff F$

Proof:

► **GetPut:** Let $q \in \mathcal{Q}$ and $e \in E$ and $u \in F$ with $u \approx_q (\text{const } E F v).\text{get } e$. By the definition of $(\text{const } E F v).\text{put}$ we have $(\text{const } E F v).\text{put } u e = e$. The required equivalence follows as \approx_k is reflexive.

► **PutGet:** Let $u \in F$ and $e \in E$. We calculate as follows

$$\begin{aligned} & (\text{const } E F v).\text{get } ((\text{const } E F v).\text{put } u e) \\ &= (\text{const } E F v).\text{get } e && \text{by definition of } (\text{const } E F v).\text{put} \\ &= \text{rep}(F) && \text{by definition of } (\text{const } E F v).\text{get} \\ &= u && \text{as } |F| = 1 \end{aligned}$$

and obtain the required equality.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $e, e' \in E$ with $e \sim_j e'$. Let u be the unique element of F . By the definition of $(\text{const } E F v).\text{get}$ we have $(\text{const } E F v).\text{get } e = u = (\text{const } E F v).\text{get } e'$. The required equivalence follows as \sim_j is reflexive, which completes the proof. \square

$$\boxed{\frac{\begin{aligned} & (S_1 \cap S_2) = \emptyset \\ & l_1 \in S_1 \iff V_1 \quad l_2 \in S_2 \iff V_2 \\ & q = \bigvee \{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \& V_2 \text{ agree}\} \\ & p = \bigvee \{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\} \end{aligned}}{l_1 \mid l_2 \in (S_1 \mid S_2):q \iff (V_1 \mid V_2):p}}$$

A.4 Lemma: $l_1 \mid l_2 \in (S_1 \mid S_2):q \iff (V_1 \mid V_2):p$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $s \in (S_1 \mid S_2):q$ and $v \in (V_1 \mid V_2):p$ with $v \approx_k \text{get } s$. We analyze two cases.

Case $k \not\sqsupseteq q$: Then \approx_k is the total relation on $(S_1 \mid S_2):q$ and $(l_1 \mid l_2).\text{put } v s \approx_k s$ trivially.

Case $k \sqsupseteq q$ and $v \in V_1$ and $s \in S_1$: From

$$v \in V_1 \quad v \approx_k^{(V_1 \mid V_2):p} (l_1 \mid l_2).\text{get } s \quad k \text{ observes } V_1 \& V_2 \text{ agree}$$

we have $v \approx_k^{V_1} l_1.get\ s$. Using this fact, we calculate as follows

$$\begin{aligned} & (l_1 \mid l_2).put\ v\ s \\ &= l_1.put\ v\ s && \text{by definition of } (l_1 \mid l_2).get \\ &\approx_k s && \text{by GETPUT for } l_1 \end{aligned}$$

and obtain the required equivalence.

Case $k \sqsupseteq q$ and $v \in V_2$ and $s \in S_2$: Symmetric to the previous case.

Case $k \sqsupseteq q$ and $v \in V_2 \setminus V_1$ and $s \in S_1$: Can't happen. The assumptions $v \approx_k (l_1 \mid l_2).get\ s$ and k observes $V_1 \neq V_2$ lead to a contradiction.

Case $k \sqsupseteq q$ and $v \in V_1 \setminus V_2$ and $s \in S_2$: Symmetric to the previous case.

► **PutGet:** Let $v \in (V_1 \mid V_2):p$ and $s \in (S_1 \mid S_2):q$. We analyze several cases.

Case $v \in V_1$ and $s \in S_1$: We calculate as follows ($ran(-)$ denotes the codomain of a function)

$$\begin{aligned} & (l_1 \mid l_2).get\ ((l_1 \mid l_2).put\ v\ s) \\ &= (l_1 \mid l_2).get\ (l_1.put\ v\ s) && \text{by definition of } (l_1 \mid l_2).put \\ &= l_1.get\ (l_1.put\ v\ s) && \text{by definition of } (l_1 \mid l_2).get \text{ as } ran(l_1.put) = S_1 \\ &= v && \text{by PUTGET for } l_1 \end{aligned}$$

where the last line follows by PUTGET for l_1 .

Case $v \in V_2$ and $s \in S_2$: Symmetric to the previous case.

Case $v \in (V_1 \setminus V_2)$ and $s \in S_2$: We calculate as follows

$$\begin{aligned} & (l_1 \mid l_2).get\ ((l_1 \mid l_2).put\ v\ s) \\ &= (l_1 \mid l_2).get\ (l_1.create\ v) && \text{by definition of } (l_1 \mid l_2).put \\ &= l_1.get\ (l_1.create\ v) && \text{by definition of } (l_1 \mid l_2).get \text{ as } ran(l_1.create) = S_1 \\ &= v && \text{by CREATEGET for } l_1 \end{aligned}$$

and obtain the required equality.

Case $v \in (V_2 \setminus V_1)$ and $s \in S_1$: Symmetric to the previous case.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $s, s' \in (S_1 \mid S_2):q$ with $s \sim_j s'$. We analyze two cases.

Case $j \not\sqsupseteq p$: Then \sim_j is the total relation on $(V_1 \mid V_2):p$ so $(l_1 \mid l_2).get\ s \approx_j (l_1 \mid l_2).get\ s'$ trivially.

Case $j \sqsupseteq p$ and $s \in S_1$: From

$$s \in S_1 \quad s \sim_j s' \quad k \text{ observes } (S_1 \cap S_2) = \emptyset$$

we have $s' \in S_1$ and $s \sim_k^{S_1} s'$. Using these facts, we calculate as follows

$$\begin{aligned} & (l_1 \mid l_2).get\ s \\ &= l_1.get\ s && \text{by definition of } (l_1 \mid l_2).get \\ &\sim_k l_1.get\ s' && \text{by GETNOLEAK for } l_1 \\ &= (l_1 \mid l_2).get\ s' && \text{by definition of } (l_1 \mid l_2).get \end{aligned}$$

and obtain the required equivalence.

Case $j \sqsupseteq p$ and $s \in S_2$: Symmetric to the previous case, which completes the proof. □

| |
|--|
| $\begin{aligned} l_1 \in S_1 &\iff V_1 \quad S_1.^!S_2 \\ l_2 \in S_2 &\iff V_2 \quad V_1.^!V_2 \\ q &= \bigvee \{k \mid k \text{ min obs. } V_1.^!V_2\} \\ p &= \bigvee \{k \mid k \text{ min obs. } S_1.^!S_2\} \\ \hline l_1 \cdot l_2 &\in (S_1 \cdot S_2):q \iff (V_1 \cdot V_2):p \end{aligned}$ |
|--|

A.5 Lemma: $l_1 \cdot l_2 \in (S_1 \cdot S_2):q \iff (V_1 \cdot V_2):p$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $(s_1 \cdot s_2) \in (S_1 \cdot S_2):q$ and $(v_1 \cdot v_2) \in (V_1 \cdot V_2):p$ with $(v_1 \cdot v_2) \approx_k (l_1 \cdot l_2).get\ (s_1 \cdot s_2)$. We analyze two cases.

Case $k \not\sqsupseteq q$: Then \approx_k is the total relation on $(S_1 \cdot S_2):q$ and so $(l_1 \cdot l_2).put\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \approx_k (s_1 \cdot s_2)$ trivially.

Case $k \sqsupseteq q$: From the assumption of the case we have that k observes $V_1.^!V_2$ and so we also have

$$v_1 \approx_k^{V_1} l_1.get\ s_1 \quad v_2 \approx_k^{V_2} l_2.get\ s_2$$

using the definition of $(l_1 \cdot l_2).get$. Using these equivalences, we calculate as follows

$$\begin{aligned} & (l_1 \cdot l_2).put\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \\ &= (l_1.put\ v_1\ s_1) \cdot (l_2.put\ v_2\ s_2) && \text{by definition of } (l_1 \cdot l_2).put \\ &\approx_k s_1 \cdot s_2 && \text{by GETPUT for } l_1 \text{ and } l_2 \end{aligned}$$

and obtain the required equivalence.

► **PutGet:** Let $(v_1 \cdot v_2) \in (V_1 \cdot V_2):p$ and $(s_1 \cdot s_2) \in (S_1 \cdot S_2):q$. We calculate as follows

$$\begin{aligned}
& (l_1 \cdot l_2).get((l_1 \cdot l_2).put(v_1 \cdot v_2)(s_1 \cdot s_2)) \\
&= (l_1 \cdot l_2).get((l_1.put\ v_1\ s_1) \cdot (l_2.put\ v_2\ s_2)) \quad \text{by definition of } (l_1 \cdot l_2).put \\
&= (l_1.get(l_1.put\ v_1\ s_1)) \cdot (l_2.get(l_2.put\ v_2\ s_2)) \quad \text{by definition of } (l_1 \cdot l_2).get \text{ with } S_1 \cdot S_2 \\
&= v_1 \cdot v_2 \quad \text{by PUTGET for } l_1 \text{ and } l_2
\end{aligned}$$

and obtain the required equality.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $(s_1 \cdot s_2), (s'_1 \cdot s'_2) \in (S_1 \cdot S_2):q$ with $(s_1 \cdot s_2) \sim_j (s'_1 \cdot s'_2)$. We analyze two cases.

Case $j \not\sqsupseteq p$: Then \sim_j is the total relation on $(V_1 \cdot V_2):p$ and so $(l_1 \cdot l_2).get(s_1 \cdot s_2) \sim_j (l_1 \cdot l_2).get(s'_1 \cdot s'_2)$ trivially.

Case $j \sqsupseteq p$: From the assumption of the case we have that j observes $S_1 \cdot S_2$ and so:

$$s_1 \sim_j^{S_1} s'_1 \quad s_2 \sim_j^{S_2} s'_2$$

Using these equivalences, we calculate as follows

$$\begin{aligned}
& (l_1 \cdot l_2).get(s_1 \cdot s_2) \\
&= (l_1.get\ s_1) \cdot (l_2.get\ s_2) \quad \text{by definition of } (l_1 \cdot l_2).get \\
&\sim_k (l_1.get\ s'_1) \cdot (l_2.get\ s'_2) \quad \text{by GETNOLEAK for } l_1 \text{ and } l_2 \text{ and the definition of } \sim_k \\
&= (l_1 \cdot l_2).get(s'_1 \cdot s'_2) \quad \text{by definition of } (l_1 \cdot l_2).get
\end{aligned}$$

and obtain the required equivalence, which completes the proof. □

| |
|---|
| $ \begin{aligned} l \in S &\iff V \quad S^{!*} \quad V^{!*} \\ q &= \bigvee \{k \mid k \text{ min obs. } V^{!*}\} \\ p &= \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \\ \hline l^* \in (S^*):q &\iff (V^*):p \end{aligned} $ |
|---|

A.6 Lemma: $l^* \in (S^*):q \iff (V^*):p$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $(s_1 \cdots s_m) \in (S^*):q$ and $(v_1 \cdots v_n) \in (V^*):p$ with $(v_1 \cdots v_n) \approx_k l^*.get(s_1 \cdots s_m)$. We analyze two cases.

Case $k \not\sqsupseteq q$: Then \approx_k is the total relation on $(S^*):q$ and so $l^*.put(v_1 \cdots v_n)(s_1 \cdots s_m) \approx_k (s_1 \cdots s_m)$ trivially.

Case $k \sqsupseteq q$: From

$$k \text{ observes } V^{!*} \quad (v_1 \cdots v_n) \approx_k l^*.get(s_1 \cdots s_m)$$

we have

$$m = n \quad v_i \approx_k l.get\ s_i \text{ for } i \in \{1, \dots, n\}$$

Using these facts, we calculate as follows

$$\begin{aligned}
& l^*.put(v_1 \cdots v_n)(s_1 \cdots s_m) \\
&= (l.put\ v_1\ s_1) \cdots (l.put\ v_n\ s_n) \quad \text{by the definition of } l^*.put \\
&\approx_k (s_1 \cdots s_m) \quad \text{by GETPUT for } l \text{ (} n \text{ times)}
\end{aligned}$$

and obtain the required equivalence.

► **PutGet:** Let $(v_1 \cdots v_n) \in (V^*):p$ and $(s_1 \cdots s_m) \in (S^*):q$. We calculate as follows

$$\begin{aligned}
& l^*.get(l^*.put(v_1 \cdots v_n)(s_1 \cdots s_m)) \\
&= l^*.get(s'_1 \cdots s'_n) \\
&\text{where } s'_i = \begin{cases} l.put\ v_i\ s_i & \text{for } i \in \{1, \dots, \min m, n\} \\ l.create\ v_i & \text{for } i \in \{m+1, \dots, n\} \end{cases} \quad \text{by definition } l^*.put \\
&= (l.get\ s'_1) \cdots (l.get\ s'_n) \quad \text{by definition of } l^*.get \text{ as } S^{!*} \\
&= (v_1 \cdots v_n) \quad \text{by PUTGET for } l \text{ (} m \text{ times) and CREATEGET for } l \text{ (} n - m \text{ times)}
\end{aligned}$$

and obtain the required equality.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $(s_1 \cdots s_m), (s'_1 \cdots s'_n) \in (S^*):q$ with $(s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$. We analyze two cases.

Case $j \not\sqsupseteq p$: Then \sim_j is the total relation on $(V^*):p$ and so $l^*.get(s_1 \cdots s_m) \sim_j l^*.get(s'_1 \cdots s'_n)$ trivially.

Case $j \sqsupseteq p$: From

$$j \text{ observes } S^{!*} \quad (s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$$

we have

$$m = n \quad s_i \sim_j^S s'_i \text{ for } i \in \{1, \dots, n\}$$

Using these facts, calculate as follows

$$\begin{aligned}
& l^*.get(s_1 \cdots s_n) \\
&= (l.get\ s_1) \cdots (l.get\ s_n) && \text{by definition of } l^*.get \\
&\sim_j (l.get\ s'_1) \cdots (l.get\ s'_n) && \text{by GETNOLEAK for } l \text{ (} n \text{ times)} \\
&= l^*.get(s'_1 \cdots s'_n) && \text{by definition of } l^*.get
\end{aligned}$$

and obtain the required equivalence, which completes the proof. \square

$$\boxed{\frac{l_1 \in S \iff T \quad l_2 \in T \iff V}{l_1; l_2 \in S \iff V}}$$

A.7 Lemma: $l_1; l_2 \in S \iff V$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $s \in S$ and $v \in V$ with $v \approx_k (l_1; l_2).get\ s$. By the definition of $(l_1; l_2).get$ we have:

$$v \approx_k l_2.get(l_1.get\ s)$$

By GETPUT for l_2 we also have

$$l_2.put\ v\ (l_1.get\ s) \approx_k (l_1.get\ s)$$

Using this fact and GETPUT for l_1 we obtain

$$(l_1; l_2).put\ v\ s = l_1.put\ (l_2.put\ v\ (l_1.get\ s))\ s \approx_k s$$

as required.

► **PutGet:** Let $v \in V$ and $s \in S$. We calculate as follows

$$\begin{aligned}
& (l_1; l_2).get\ ((l_1; l_2).put\ v\ s) \\
&= (l_1; l_2).get\ (l_1.put\ (l_2.put\ v\ (l_1.get\ s))\ s) && \text{by definition of } (l_1; l_2).put \\
&= l_2.get\ (l_1.get\ (l_1.put\ (l_2.put\ v\ (l_1.get\ s))\ s)) && \text{by definition of } (l_1; l_2).get \\
&= l_2.get\ (l_2.put\ v\ (l_1.get\ s)) && \text{by PUTGET for } l_1 \\
&= v && \text{by PUTGET for } l_2
\end{aligned}$$

and obtain the required equality.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $s, s' \in S$ with $s \sim_j s'$. We calculate as follows:

$$\begin{aligned}
& (l_1; l_2).get\ s \\
&= l_2.get\ (l_1.get\ s) && \text{by definition of } (l_1; l_2).get \\
&\sim_j l_2.get\ (l_1.get\ s') && \text{by GETNOLEAK for } l_1 \text{ and } l_2 \\
&= (l_1; l_2).get\ s' && \text{by definition of } (l_1; l_2).get
\end{aligned}$$

and obtain the required equivalence. \square

$$\boxed{\frac{\begin{array}{l} E, F \text{ well-formed} \quad E \cap F = \emptyset \quad (E \mid F)^{!*} \\ q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\} \\ p \sqsupseteq \bigvee \{k \mid k \text{ observes } E.^!F \text{ and } F.^!E\} \end{array}}{\text{filter } E\ F \in (E:q \mid F:p)^* \iff E^*}}$$

A.8 Lemma: $\text{filter } E\ F \in (E:q \mid F:p)^* \iff E^*$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $(s_1 \cdots s_m) \in (E:q \mid F:p)^*$ and $(v_1 \cdots v_n) \in E^*$ with $(v_1 \cdots v_n) \approx_k ((\text{filter } E\ F).get\ (s_1 \cdots s_m))$. We analyze two cases.

Case $k \not\sqsupseteq q$: From the assumption of the case we have that $\text{hide}(k)$ maps elements of E to ϵ . Let (f_1, \dots, f_o) be the elements of F in $(s_1 \cdots s_m)$. By the definition of unfilter, the elements of F in $\text{unfilter } F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)$ are also (f_1, \dots, f_o) . Using these facts, we calculate as follows

$$\begin{aligned}
& \text{hide}(k)\ ((\text{filter } E\ F).put\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)) \\
&= \text{hide}(k)\ (\text{unfilter } F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)) && \text{by definition of } (\text{filter } E\ F).put \\
&= \text{hide}(k)\ (f_1 \cdots f_o) && \text{as } \text{hide}(k) \text{ maps elements of } E \text{ to } \epsilon \\
&= \text{hide}(k)\ (s_1, \dots, s_m) && \text{also as } \text{hide}(k) \text{ maps elements of } E \text{ to } \epsilon
\end{aligned}$$

and obtain the required equivalence.

Case $k \sqsupseteq q$: Let (t_1, \dots, t_o) be the elements of $(E : q)$ in $(s_1 \cdots s_m)$. From

$$k \text{ observes } E^{!*} \quad (v_1 \cdots v_n) \approx_k (\text{filter } E\ F).get\ (s_1 \cdots s_m)$$

we have:

$$n = o \quad v_i \approx_k^E t_i \text{ for } i \in \{1, \dots, n\}$$

Using these facts, we calculate as follows:

$$\begin{aligned} & \text{hide}(k) ((\text{filter } E \ F).put (v_1 \cdots v_n) (s_1 \cdots s_m)) \\ & \text{hide}(k) (\text{unfilter } F (v_1 \cdots v_n) (s_1 \cdots s_m)) && \text{by definition of } (\text{filter } E \ F).put \\ = & \text{hide}(k) (s'_1 \cdots s'_m) \\ & \text{where } s'_i = \begin{cases} v_j & \text{if } s_i \text{ is the } j^{\text{th}} \text{ element of } E \text{ in } (s_1 \cdots s_m) \\ s_i & \text{if } s_i \in F \end{cases} && \text{by definition of unfilter} \\ = & \text{hide}(k) (s_1 \cdots s_m) && \text{as } \text{hide}(k) (v_i) = \text{hide}(k) (t_i) \text{ for } i \in \{1, \dots, o\} \end{aligned}$$

and obtain the required equivalence.

► **PutGet:** Let $(v_1 \cdots v_n) \in E^*$ and $(s_1 \cdots s_m) \in (E:q \mid F:p)^*$. Observe that filter preserves the elements of E in its argument and that unfilter propagates all of the elements in its first argument, which all belong to E . Using these facts, we calculate as follows

$$\begin{aligned} & (\text{filter } E \ F).get ((\text{filter } E \ F).put (v_1 \cdots v_n) (s_1 \cdots s_m)) \\ = & (\text{filter } E \ F).get (\text{unfilter } F (v_1 \cdots v_n) (s_1 \cdots s_m)) && \text{by definition of } (\text{filter } E \ F).put \\ = & \text{filter } E (\text{unfilter } F (v_1 \cdots v_n) (s_1 \cdots s_m)) && \text{by definition of } (\text{filter } E \ F).get \\ = & (v_1 \cdots v_n) && \text{by above facts about filter and unfilter} \end{aligned}$$

and obtain the required equality.

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $(s_1 \cdots s_m), (s'_1 \cdots s'_n) \in (E:q \mid F:p)^*$ with $(s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$. We analyze two cases

Case $j \not\sqsupseteq p$: Then $\text{hide}(j)$ maps elements of F to ϵ . Let (e_1, \dots, e_i) and (e'_1, \dots, e'_j) be the elements of $(s_1 \cdots s_m)$ and $(s'_1 \cdots s'_n)$ that belong to E . From $(s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$ we calculate as follows

$$\begin{aligned} & \text{hide}(j) (s_1 \cdots s_m) = \text{hide}(j) (s_1 \cdots s_n) \\ \text{i.e., } & \text{hide}(j) (e_1 \cdots e_i) = \text{hide}(j) (e'_1 \cdots e'_j) && \text{as } \text{hide}(j) \text{ maps elements of } F \text{ to } \epsilon \\ \text{i.e., } & \text{hide}(j) ((\text{filter } E \ F).get (s_1 \cdots s_m)) = \text{hide}(j) ((\text{filter } E \ F).get (s'_1 \cdots s'_n)) && \text{by definition of } (\text{filter } E \ F).get \\ \text{i.e., } & (\text{filter } E \ F).get (s_1 \cdots s_m) \sim_j (\text{filter } E \ F).get (s'_1 \cdots s'_n) && \text{by definition of } \sim_j \end{aligned}$$

and obtain the required equivalence.

Case $j \sqsupseteq p$: Let $(e_{11} \cdots e_{1i}), \dots, (e_{o1} \cdots e_{oj})$ and $(e'_{11} \cdots e'_{1q}), \dots, (e'_{p1}, \dots, e'_{pr})$ be the *contiguous*—i.e., those not separated by an F —elements of E in $(s_1 \cdots s_m)$ and $(s'_1 \cdots s'_n)$. From $(s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$ and j observes $E.^1F$ and $F.^1E$ we have

$$o = p \quad \text{hide}(j) (e_{11} \cdots e_{1i}) = \text{hide}(j) (e'_{11} \cdots e'_{1q}) \quad \dots \quad \text{hide}(j) (e_{o1} \cdots e_{oj}) = \text{hide}(j) (e'_{p1} \cdots e'_{pr})$$

Using these facts, we calculate as follows

$$\begin{aligned} & \text{hide}(j) (e_{11} \cdots e_{1i}) \cdots \text{hide}(j) (e_{o1} \cdots e_{oj}) = \text{hide}(j) (e'_{11} \cdots e'_{1q}) \cdots \text{hide}(j) (e'_{p1} \cdots e'_{pr}) \\ \text{i.e., } & \text{hide}(j) ((e_{11} \cdots e_{1i}) \cdots (e_{o1} \cdots e_{oj})) = \text{hide}(j) ((e'_{11} \cdots e'_{1q}) \cdots (e'_{p1} \cdots e'_{pr})) && \text{by definition } \text{hide}(j) \\ \text{i.e., } & \text{hide}(j) (\text{filter } E (s_1 \cdots s_m)) = \text{hide}(j) (\text{filter } E (s'_1 \cdots s'_n)) && \text{by definition of filter} \\ \text{i.e., } & \text{hide}(j) ((\text{filter } E \ F).get (s_1 \cdots s_m)) = \text{hide}(j) ((\text{filter } E \ F).get (s'_1 \cdots s'_n)) && \text{by definition of } (\text{filter } E \ F).get \\ \text{i.e., } & (\text{filter } E \ F).get (s_1 \cdots s_m) \sim_j (\text{filter } E \ F).get (s'_1 \cdots s'_n) && \text{by definition of } \sim_j \end{aligned}$$

and obtain the required equivalence, which completes the proof. \square

$$\boxed{\frac{l \in S \iff V \quad q \in Q \quad p \in \mathcal{P}}{l \in (S:q) \iff (V:p)}}$$

A.9 Lemma: $l \in (S:q) \iff (V:p)$

Proof:

► **GetPut:** Let $k \in \mathcal{Q}$ and $s \in (C:q)$ and $s \in (V:p)$ with $v \approx_k (l).get \ s$. We analyze two cases:

Case $k \not\sqsupseteq q$: Then \approx_k is the total relation on $(C:q)$ and so and so $(l).put \ v \ s \approx_k \ s$ trivially.

Case $k \sqsupseteq q$: By the definition of the semantics of $V:p$ we have $v \approx_k^V (l).get \ s$. By GETPUT for l we have $(l).put \ v \ s \approx_k^S \ s$. Finally, by the definition of the semantics of $S:q$ we have $(l).put \ v \ s \approx_k^{(S:q)} \ s$, as required.

► **PutGet:** Follows directly from PUTGET for l .

► **GetNoLeak:** Let $j \in \mathcal{P}$ and $s, s' \in (S:q)$ with $s \sim_k s'$. We analyze two cases:

Case $j \not\sqsupseteq p$: Then \sim_j is the total relation on $V:p$ and so and so $(l).get \ s \approx_j (l).get \ s'$ trivially.

Case $j \sqsupseteq p$: By the definition of the semantics of $S:q$ we have $s \sim_j^S s'$. Then, by GETNOLEAK for l we have $(l).get \ s \sim_k^V (l).get \ s'$. By the definition of the semantics of $V:p$ we also have $(l).get \ s \sim_j (l).get \ s'$ as required, which completes the proof. \square

3. Dynamic Very Well Behavedness Proofs

A.10 Lemma: The following rule of inference is admissible:

$$\frac{v' \approx_k l.get\ s \approx_k v}{l.put\ v'\ (l.put\ v\ s) \approx_k l.put\ v'\ s} \quad (\text{PUTPUTENDORSED})$$

Proof: Let $k \in \mathcal{Q}$ and $v, v' \in V$ and $s \in S$ with $v \approx_k l.get\ s$ and $v' \approx_k l.get\ s$. By PUTGET for l we have:

$$l.get\ (l.put\ v\ s) = v$$

By the reflexivity of \approx_k it follows that:

$$l.get\ (l.put\ v\ s) \approx_k v$$

Using this fact, we calculate as follows:

$$\begin{array}{ll} l.put\ v'\ (l.put\ v\ s) & \\ \approx_k l.put\ v\ s & \text{by GETPUT for } l \\ \approx_k s & \text{by GETPUT for } l \end{array}$$

Also by GETPUT for l we have

$$l.put\ v'\ s \approx_k s$$

The required equivalence follows from the transitivity of \approx_k . □

A.11 Lemma: $copy\ E \in E \iff E$

Proof:

► **GetPut:** Let $(j, k) \in \mathcal{C}$ and $s \in E$ and $v \in E$ with $(copy\ E).safe\ (j, k)\ v\ s$. By the definition of $(copy\ E).safe$ we have $v \approx_k s$. Using this fact, we calculate as follows

$$\begin{array}{ll} (copy\ E).put\ v\ s & \\ = v & \text{by the definition of } (copy\ E).put \\ \approx_k s & \end{array}$$

and obtain the required equivalence.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $s, s' \in E$ and $v, v' \in E$ with $s \sim_j s'$ and $v \sim_j v'$ and $(copy\ E).safe\ (j, k)\ v\ s$ and $(copy\ E).safe\ (j, k)\ v'\ s'$. We calculate as follows

$$\begin{array}{ll} (copy\ E).put\ v\ s & \\ = v & \text{by definition of } (copy\ E).put \\ \sim_j v' & \text{by assumption} \\ = (copy\ E).put\ v'\ s' & \text{by definition of } (copy\ E).put \end{array}$$

and obtain the required equivalence.

► **SafeNoLeak:** Let $(j, k) \in \mathcal{C}$ and $s, s' \in E$ and $v, v' \in E$ with $v \sim_j v'$ and $s \sim_j s'$. Then as $\sim_j \subseteq \approx_k$ we have $s \approx_k s'$ and $v \approx_k v'$. Using these equivalences, we calculate as follows

$$\begin{array}{ll} (copy\ E).safe\ (j, k)\ v\ s & \\ \text{iff } v \approx_k s & \text{by definition of } (copy\ E).safe \\ \text{iff } v' \approx_k s' & \text{by symmetry and transitivity of } \approx_k \\ \text{iff } (copy\ E).safe\ (j, k)\ v'\ s' & \text{by definition of } (copy\ E).safe \end{array}$$

and obtain the required equality, which completes the proof. □

$$\boxed{\frac{E, F \text{ well-formed} \quad |F| = 1 \quad d \in E}{const\ E\ F\ d \in E \iff F}}$$

A.12 Lemma: $const\ E\ F\ d \in E \iff F$

Proof:

► **GetPut:** Identical to previous proof of GETPUT for $const$.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $s, s' \in E$ and $v, v' \in F$ with $s \sim_j s'$ and $v \sim_j v'$ and $(const\ E\ F\ d).safe\ (j, k)\ v\ s$ and $(const\ E\ F\ d).safe\ (j, k)\ v'\ s'$. We calculate as follows

$$\begin{array}{ll} (const\ E\ F\ d).put\ v\ s & \\ = s & \text{by definition of } (const\ E\ F\ d).put \\ \sim_j s' & \text{by assumption} \\ = (const\ E\ F\ d).put\ v'\ s' & \text{by definition of } (const\ E\ F\ d).put \end{array}$$

and obtain the required equivalence.

► **SafeNoLeak:** Let $s, s' \in E$ and $v, v' \in E$ with $s \sim_j s'$ and $v \sim_j v'$. We have

$$(const\ E\ F\ d).safe\ (j, k)\ v\ s = true = (const\ E\ F\ d).safe\ (j, k)\ v\ s$$

immediately, which completes the proof. \square

$$\boxed{\begin{array}{l} l_1 \in S_1 \iff V_1 \quad S_1.^!S_2 \\ l_2 \in S_2 \iff V_2 \quad V_1.^!S_2 \\ p = \bigvee \{k \mid k \text{ min obs. } S_1.^!S_2\} \\ \hline l_1 \cdot l_2 \in (S_1 \cdot S_2) \iff (V_1 \cdot V_2) : p \end{array}}$$

A.13 Lemma: $l_1 \cdot l_2 \in (S_1 \cdot S_2) \iff (V_1 \cdot V_2) : p$

Proof:

► **GetPut:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \cdot v_2) \in (V_1 \cdot V_2) : p$ and $(s_1 \cdot s_2) \in (S_1 \cdot S_2)$ with $(l_1 \cdot l_2).safe\ (j, k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2)$. By the definition of $(l_1 \cdot l_2).safe$ we have:

$$l_1.safe\ (j, k)\ v_1\ s_1 \quad l_2.safe\ (j, k)\ v_2\ s_2$$

Using these facts, we calculate as follows

$$\begin{aligned} & (l_1 \cdot l_2).put\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \\ &= (l_1.put\ v_1\ s_1) \cdot (l_2.put\ v_2\ s_2) \quad \text{by definition of } (l_1 \cdot l_2).put \\ &\approx_k\ s_1 \cdot s_2 \quad \text{by GETPUT for } l_1 \text{ and } l_2 \end{aligned}$$

and obtain the required equivalence.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $(s_1 \cdot s_2), (s'_1 \cdot s'_2) \in (S_1 \cdot S_2)$ and $(v_1 \cdot v_2), (v'_1 \cdot v'_2) \in (V_1 \cdot V_2) : p$ with:

$$(s_1 \cdot s_2) \sim_j (s'_1 \cdot s'_2) \quad (v_1 \cdot v_2) \sim_j (v'_1 \cdot v'_2) \\ (l_1 \cdot l_2).safe\ (j, k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \quad (l_1 \cdot l_2).safe\ (j, k)\ (v'_1 \cdot v'_2)\ (s'_1 \cdot s'_2)$$

From the definition of $(l_1 \cdot l_2).safe$, we have that j observes $S_1.^!S_2$ and $V_1.^!V_2$ and hence:

$$\begin{array}{cc} s_1 \sim_j^{S_1} s'_1 & s_2 \sim_j^{S_2} s'_2 \\ v_1 \sim_j^{V_1} v'_1 & v_2 \sim_j^{V_2} v'_2 \end{array}$$

Also by the definition of $(l_1 \cdot l_2).safe$ we have

$$\begin{array}{cc} l_1.safe\ (j, k)\ v_1\ s_1 & l_2.safe\ (j, k)\ v_2\ s_2 \\ l_1.safe\ (j, k)\ v_1\ s'_1 & l_2.safe\ (j, k)\ v_2\ s'_2 \end{array}$$

Using all these facts, we calculate as follows

$$\begin{aligned} & (l_1 \cdot l_2).put\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \\ &= (l_1.put\ v_1\ s_1) \cdot (l_2.put\ v_2\ s'_2) \quad \text{by definition of } (l_1 \cdot l_2).put \\ &\sim_j (l_1.put\ v'_1\ s'_1) \cdot (l_2.put\ v'_2\ s'_2) \quad \text{by PUTNOLEAK for } l_1 \text{ and } l_2 \\ &= (l_1 \cdot l_2).put\ (v'_1 \cdot v'_2)\ (s'_1 \cdot s'_2) \quad \text{by definition of } (l_1 \cdot l_2).put \end{aligned}$$

and obtain the required equivalence.

► **SafeNoLeak:** Let $(j, k) \in \mathcal{C}$ and $(s_1 \cdot s_2), (s'_1 \cdot s'_2) \in (S_1 \cdot S_2)$ and $(v_1 \cdot v_2), (v'_1 \cdot v'_2) \in (V_1 \cdot V_2) : p$ with $(s_1 \cdot s_2) \sim_j (s'_1 \cdot s'_2)$ and $(v_1 \cdot v_2) \sim_j (v'_1 \cdot v'_2)$. We analyze two cases.

Case j observes $S_1.^!S_2$ and $V_1.^!V_2$: From the assumptions of the case and the definition of \sim_j we have:

$$\begin{array}{cc} s_1 \sim_j^{S_1} s'_1 & s_2 \sim_j^{S_2} s'_2 \\ v_1 \sim_j^{V_1} v'_1 & v_2 \sim_j^{V_2} v'_2 \end{array}$$

Using these facts, we calculate as follows

$$\begin{aligned} & (l_1 \cdot l_2).safe\ (j, k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \\ &\text{iff } (l_1.safe\ (j, k)\ v_1\ s_1) \wedge (l_2.safe\ (j, k)\ v_2\ s_2) \quad \text{by definition of } (l_1 \cdot l_2).safe \\ &\text{iff } (l_1.safe\ (j, k)\ v'_1\ s'_1) \wedge (l_2.safe\ (j, k)\ v'_2\ s'_2) \quad \text{by SAFENOLEAK for } l_1 \text{ and } l_2 \\ &\text{iff } (l_1 \cdot l_2).safe\ (j, k)\ (v'_1 \cdot v'_2)\ (s'_1 \cdot s'_2) \end{aligned}$$

and obtain the required equality.

Case j does not observe $S_1.^!S_2$ and $V_1.^!V_2$: Then

$$(l_1 \cdot l_2).safe\ (j, k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) = false = (l_1 \cdot l_2).safe\ (j, k)\ (v'_1 \cdot v'_2)\ (s'_1 \cdot s'_2)$$

immediately, which completes the case and the proof. \square

$$\boxed{\begin{array}{c} (S_1 \cap S_2) = \emptyset \\ l_1 \in S_1 \iff V_1 \quad l_2 \in S_2 \iff V_2 \\ p = \bigvee \{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\} \\ \hline l_1 \mid l_2 \in (S_1 \mid S_2) \iff (V_1 \mid V_2):p \end{array}}$$

A.14 Lemma: $l_1 \mid l_2 \in (S_1 \mid S_2) \iff (V_1 \mid V_2):p$

Proof:

► **GetPut:** Let $(j, k) \in \mathcal{C}$ and $s \in (S_1 \mid S_2)$ and $v \in (V_1 \mid V_2):p$ with $(l_1 \mid l_2).safe(j, k) v s$. We analyze two cases.

Case $s \in S_1$: From the assumption of the case and $(l_1 \mid l_2).safe(j, k) v s$ we have:

$$v \in V_1 \quad l_1.safe(j, k) v s$$

Using these facts, we calculate as follows

$$\begin{aligned} & (l_1 \mid l_2).put v s \\ &= l_1.put v s && \text{by definition of } (l_1 \mid l_2).put \\ &\approx_k s && \text{by GETPUT for } l_1 \end{aligned}$$

and obtain the required equivalence.

Case $s \in S_2$: Symmetric to the previous case.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $s, s' \in (S_1 \mid S_2)$ and $v, v' \in (V_1 \mid V_2):p$ with:

$$\begin{array}{l} s \sim_j s' \quad (l_1 \mid l_2).safe(j, k) v s \\ v \sim_j v' \quad (l_1 \mid l_2).safe(j, k) v' s' \end{array}$$

We analyze two cases.

Case $s \in S_1$: From the definition of $(l_1 \mid l_2).safe$ we have that j observes $(S_1 \cap S_2) = \emptyset$ and V_1 & V_2 agree. Using this fact and the assumption that $s \sim_j s'$ we have that $s' \in S_1$ and $s \sim_j^{S_1} s'$. Next, from $(l_1 \mid l_2).safe(j, k) v s$ and $(l_1 \mid l_2).safe(j, k) v' s'$ we have:

$$\begin{array}{l} v \in V_1 \quad l_1.safe(j, k) v s \\ v' \in V_1 \quad l_1.safe(j, k) v' s' \end{array}$$

Finally, from j observes V_1 & V_2 agree we also have $v \sim_j^{V_1} v'$. Putting all these facts together, we calculate as follows

$$\begin{aligned} & (l_1 \mid l_2).put v s \\ &= l_1.put v s && \text{by definition of } (l_1 \mid l_2).put \\ &\sim_j l_1.put v s' && \text{by PUTNOLEAK for } l_1 \\ &= (l_1 \mid l_2).put v s' && \text{by definition of } (l_1 \mid l_2).put \end{aligned}$$

and obtain the required equivalence.

Case $s \in S_2$: Symmetric to the previous case.

► **SafeNoLeak:** Let $(j, k) \in \mathcal{C}$ and $v, v' \in (V_1 \mid V_2):p$ and $s, s' \in (S_1 \mid S_2)$ with $v \sim_j v'$ and $s \sim_j s'$ and $(l_1 \mid l_2).safe v s$ and $(l_1 \mid l_2).safe v' s'$. We analyze several cases.

Case j observes $(S_1 \cap S_2) = \emptyset$ and V_1 & V_2 agree and $s \in S_1$: From the assumptions of the case we have $s' \in S_1$ and $s \sim_j^{S_1} s'$. We also have $v \in V_1$ if and only if $v' \in V_1$ and $v \sim_j^{V_1} v'$. Using these facts we calculate as follows

$$\begin{array}{l} (l_1 \mid l_2).safe(j, k) v s \\ \text{iff } v \in V_1 \wedge l_1.safe(j, k) v s && \text{by definition of } (l_1 \mid l_2).safe \\ \text{iff } v' \in V_1 \wedge l_1.safe(j, k) v' s' && \text{by SAFENOLEAK for } l_1 \\ \text{iff } (l_1 \mid l_2).safe(j, k) v' s' && \text{by definition of } (l_1 \mid l_2).safe \end{array}$$

Case j observes $(S_1 \cap S_2) = \emptyset$ and V_1 & V_2 agree and $s \in S_2$: Symetric to the previous case.

Case j does not observe $(S_1 \cap S_2) = \emptyset$ and V_1 & V_2 agree: Then

$$(l_1 \mid l_2).safe(j, k) v s = false = (l_1 \mid l_2).safe(j, k) v' s'$$

immediately, which completes the case and the proof. □

$$\boxed{\begin{array}{c} l \in S \iff V \\ p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \\ \hline l^* \in S^* \iff (V^*):p \end{array}}$$

A.15 Lemma: $l^* \in S^* \iff (V^*):p$

Proof:

► **GetPut:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \cdots v_n) \in (V^*):p$ and $(s_1 \cdots s_m) \in C^*$ with $(l^*).safe(j, k)(v_1 \cdots v_n)(s_1 \cdots s_m)$. By the definition of $(l^*).safe$ we have $n = m$ and $l.safe(j, k) v_i s_i$ for every i from 1 to n . Using these facts, we calculate as follows

$$\begin{aligned} & (l^*).put(v_1 \cdots v_n)(s_1 \cdots s_m) \\ &= (l.put\ v_1\ s_1) \cdots (l.put\ v_n\ s_n) \quad \text{by definition of } (l^*).put \\ &\approx_k s_1 \cdots s_m \quad \text{by GETPUT for } l \text{ (} n \text{ times)} \end{aligned}$$

and obtain the required equivalence.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \cdots v_m), (v'_1 \cdots v'_n) \in (V^*):p$ and $(s_1 \cdots s_o), (s'_1 \cdots s'_p) \in S^*$ with

$$\begin{aligned} (v_1 \cdots v_m) \sim_j (v'_1 \cdots v'_n) & \quad (l^*).safe(j, k)(v_1 \cdots v_m)(s_1 \cdots s_o) \\ (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p) & \quad (l^*).safe(j, k)(v'_1 \cdots v'_n)(s_1 \cdots s'_p) \end{aligned}$$

By the definition of $(l^*).safe$ we have that j observes S^{l^*} and V^{l^*} and so

$$m = n = o = p \quad l.safe(j, k) v_i s_i \quad l.safe(j, k) v'_i s'_i \quad s_i \sim_j^S s'_i \quad v_i \sim_j^V v'_i \text{ for } i \in \{1, \dots, n\}$$

Using these facts, we calculate as follows

$$\begin{aligned} & (l^*).put(v_1 \cdots v_n)(s_1 \cdots s_o) \\ &= (l.put\ v_1\ s_1) \cdots (l.put\ v_n\ s_n) \quad \text{by definition of } (l^*).put \\ &\sim_j (l.put\ v'_1\ s'_1) \cdots (l.put\ v'_m\ s'_m) \quad \text{by PUTNOLEAK for } l \text{ (} n \text{ times)} \\ &= (l^*).put(v'_1 \cdots v'_m)(s'_1 \cdots s'_p) \quad \text{by definition of } (l^*).put \end{aligned}$$

and obtain the required equivalence.

► **SafeNoLeak:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \cdots v_n), (v'_1 \cdots v'_m) \in (V^*):p$ and $(s_1 \cdots s_o), (s'_1 \cdots s'_p) \in S^*$ with:

$$(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_m) \quad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)$$

We analyze several cases.

Case j observes S^{l^*} and V^{l^*} and $n = o$: From the assumptions and the definition of $(l^*).safe$ we have that

$$m = n = o = p \quad v_i \sim_k^V v'_i \quad s_i \sim_j^S s'_i \text{ for } i \in \{1 \dots n\}$$

Using these facts we calculate as follows

$$\begin{aligned} & (l^*).safe(j, k)(v_1 \cdots v_n)(s_1 \cdots s_n) \\ & \text{iff } l.safe(j, k) v_1 s_1 \wedge \dots \wedge l.safe(j, k) v_n s_n \quad \text{by definition } (l^*).safe \\ & \text{iff } l.safe(j, k) v'_1 s'_1 \wedge \dots \wedge l.safe(j, k) v'_n s'_n \quad \text{by SAFENOLEAK for } l \text{ (} n \text{ times)} \\ & \text{iff } (l^*).safe(j, k)(v'_1 \cdots v'_n)(s'_1 \cdots s'_n) \quad \text{by definition } (l^*).safe \end{aligned}$$

and obtain the required equality.

Case j observes S^{l^*} and V^{l^*} and $n \neq o$: From the assumptions of the case and

$$(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_m) \quad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)$$

we have $n = m$ and $o = p$. It follows that

$$(l^*).safe(j, k)(v_1 \cdots v_n)(s_1 \cdots s_o) = false = (l^*).safe(v'_1 \cdots v'_m)(s'_1 \cdots s'_p)$$

Case j does not observe S^{l^*} and V^{l^*} : Then

$$(l^*).safe(j, k)(v_1 \cdots v_n)(s_1 \cdots s_o) = false = (l^*).safe(v'_1 \cdots v'_m)(s'_1 \cdots s'_p)$$

immediately, which completes the case and the proof. □

| |
|---|
| $l_1 \in S \iff T \quad l_2 \in T \iff V$ |
| $l_1; l_2 \in S \iff V$ |

A.16 Lemma: $l_1; l_2 \in S \iff V$

Proof:

► **GetPut:** Let $(j, k) \in \mathcal{C}$ and $s \in S$ and $v \in V$ with $(l_1; l_2).safe(j, k) v s$. By the definition of $(l_1; l_2).safe$ we have

$$l_1.safe(l_2.put\ v\ (l_1.get\ s))\ s$$

Using this fact, we calculate as follows

$$\begin{aligned} & (l_1; l_2).put\ v\ s \\ &= l_1.put\ (l_2.put\ v\ (l_1.get\ s))\ s \quad \text{by definition of } (l_1; l_2).put \\ &\approx_k s \quad \text{by GETPUT for } l_1 \end{aligned}$$

and obtain the required equivalence.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $v, v' \in V$ and $s, s' \in S$ with $s \sim_j s'$ and $v \sim_j v'$ and $(l_1; l_2).safe\ v\ s$ and $(l_1; l_2).safe\ v'\ s'$. By the definition of $(l_1; l_2).safe$ we have

$$l_1.safe\ (l_2.put\ v\ (l_1.get\ s))\ s \quad l_1.safe\ (l_2.put\ v'\ (l_1.get\ s'))\ s'$$

Using these facts, we calculate as follows

$$\begin{aligned} & (l_1; l_2).put\ v\ s \\ &= l_1.put\ (l_2.put\ v\ (l_1.get\ s))\ s \quad \text{by definition of } (l_1; l_2).put \\ &\sim_j l_1.put\ (l_2.put\ v'\ (l_1.get\ s'))\ s' \quad \text{by PUTNOLEAK for } l_1 \\ &= (l_1; l_2).put\ v'\ s' \quad \text{by definition of } (l_1; l_2).put \end{aligned}$$

and obtain the required equivalence.

► **SafeNoLeak:** Let $(j, k) \in \mathcal{C}$ and $s, s' \in S$ and $v, v' \in V$ with $s \sim_j s'$ and $v \sim_j v'$. By GETNOLEAK for l_1 we have that:

$$(l_1.get\ s) \sim_j (l_1.get\ s')$$

By PUTNOLEAK for l_2 we have

$$l_2.put\ v\ (l_1.get\ s) \sim_j l_2.put\ v'\ (l_1.get\ s')$$

Using SAFENOLEAK for l_1 we obtain the required equality

$$l_1.safe\ (l_2.put\ v\ (l_1.get\ s))\ s = l_1.safe\ (l_2.put\ v'\ (l_1.get\ s'))\ s'$$

which completes the proof. □

| |
|---|
| $\begin{array}{l} E, F \text{ well-formed} \quad E \cap F = \emptyset \quad (E \mid F)^{!*} \\ p \sqsupseteq \bigvee \{k \mid k \text{ observes } E.^!F \text{ and } F.^!E\} \\ \forall (j, k) \in \mathcal{C}. \sim_j^E \subseteq \approx_k^E \\ \hline \text{filter } E\ F \in (E \mid F:p)^* \iff E^* \end{array}$ |
|---|

A.17 Lemma: $\text{filter } E\ F \in (E \mid F:p)^* \iff E^*$

Proof:

► **GetPut:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \cdots v_n) \in E^*$ and $(s_1 \cdots s_m) \in (E \mid F:p)^*$ with $(\text{filter } E\ F).safe\ (j, k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)$. Let (t_1, \dots, t_o) be the elements of E in $(s_1 \cdots s_m)$. By the definition of $(\text{filter } E\ F).safe$ we have that

$$k \text{ observes } E^{!*} \quad (v_1 \cdots v_n) \approx_k \text{filter } E\ (s_1 \cdots s_m)$$

and hence:

$$n = o \quad v_i \approx_k^E t_i \text{ for } i \in \{1 \dots n\}$$

Using these facts, we calculate as follows

$$\begin{aligned} & \text{hide}(k)\ ((\text{filter } E\ F).put\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)) \\ &= \text{hide}(k)\ (\text{unfilter } F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)) \quad \text{by definition of } (\text{filter } E\ F).put \\ &= \text{hide}(k)\ (s'_1 \cdots s'_m) \\ & \quad \text{where } s'_i = \begin{cases} v_j & \text{if } s_i \text{ is the } j^{\text{th}} \text{ element of } E \text{ in } (s_1 \cdots s_m) \\ s_i & \text{if } s_i \in F \end{cases} \quad \text{by definition of unfilter} \\ &= \text{hide}(k)\ (s_1 \cdots s_m) \quad \text{as } v_i \approx_k t_i \text{ for } i \in \{1, \dots, o\} \\ & \text{i.e. } (\text{filter } E\ F).put\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) \approx_k (s_1 \cdots s_m) \end{aligned}$$

and obtain the required equivalence.

► **PutNoLeak:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \cdots v_m), (v'_1 \cdots v'_n) \in E^*$ and $(s_1 \cdots s_o), (s'_1 \cdots s'_p) \in (E \mid F:p)^*$ with:

$$\begin{aligned} (v_1 \cdots v_m) \sim_j (v'_1 \cdots v'_n) & \quad (\text{filter } E\ F).safe\ (j, k)\ (v_1 \cdots v_m)\ (s_1 \cdots s_o) \\ (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p) & \quad (\text{filter } E\ F).safe\ (j, k)\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_p) \end{aligned}$$

Let (t_1, \dots, t_w) and (t'_1, \dots, t'_x) be the elements of E in $(s_1 \cdots s_o)$ and $(s'_1 \cdots s'_p)$. As j observes $E^{!*}$ we have that

$$m = n \text{ and } v_i \sim_j v'_i \text{ for } i \in \{1, \dots, m\}$$

Let $(f_{11} \cdots f_{1i}), \dots, (f_{y1} \cdots f_{yj})$ and $(f'_{11} \cdots f'_{1k}), \dots, (f'_{z1}, \dots, f'_{zl})$ be the contiguous elements of F in $(s_1 \cdots s_o)$ and $(s'_1 \cdots s'_p)$. As $(s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)$ and j observes $E.^!F$ and $F.^!E$ we have that $y = z$ and

$$(f_{11} \cdots f_{1i}) \sim_j (f'_{11} \cdots f'_{1k}) \wedge \dots \wedge (f_{y1} \cdots f_{yj}) \sim_j (f'_{z1} \cdots f'_{zl})$$

Using these facts, we calculate as follows

$$\begin{aligned}
& \text{hide}(j) ((\text{filter } E \ F).\text{put } (v_1 \dots v_m) (s_1 \dots s_o)) \\
&= \text{hide}(j) (\text{unfilter } (v_1 \dots v_m) (s_1 \dots s_o)) && \text{by definition of } (\text{filter } E \ F).\text{put} \\
&= \text{hide}(j) (s'_1 \dots s'_{o+(m-w)}) \\
&\quad \text{where } s'_i = \begin{cases} v_{i-w} & \text{if } i > o \\ v_j & \text{if } i \leq o \text{ and } s_i \text{ is the } j^{\text{th}} \text{ element of } \in E \text{ in } (s_1 \dots s_o) \\ s_i & \text{if } s_i \in F \end{cases} && \text{by definition of unfilter} \\
&= \text{hide}(j) (s''_1 \dots s''_{p+(n-x)}) \\
&\quad \text{where } s'_i = \begin{cases} v'_{i-x} & \text{if } i > p \\ v'_j & \text{if } i \leq p \text{ and } s'_i \text{ is the } j^{\text{th}} \text{ element of } \in E \text{ in } (s'_1 \dots s'_p) \\ s'_i & \text{if } s'_i \in F \end{cases} && \text{by above facts} \\
&= \text{hide}(j) (\text{unfilter } (v'_1 \dots v'_n) (s'_1 \dots s'_p)) && \text{by definition of unfilter} \\
&= \text{hide}(j) ((\text{filter } E \ F).\text{put } (v'_1 \dots v'_n) (s'_1 \dots s'_p)) && \text{by definition of } (\text{filter } E \ F).\text{put} \\
&\text{i.e., } (\text{filter } E \ F).\text{put } (v_1 \dots v_n) (s_1 \dots s_o) \sim_j (\text{filter } E \ F).\text{put } (v'_1 \dots v'_n) (s'_1 \dots s'_p)
\end{aligned}$$

and obtain the required equivalence.

► **SafeNoLeak:** Let $(j, k) \in \mathcal{C}$ and $(v_1 \dots v_n), (v'_1 \dots v'_m) \in E^*$ and $(s_1 \dots s_o), (s'_1 \dots s'_p) \in (E \mid F:p)^*$ with:

$$(v_1 \dots v_n) \sim_j (v'_1 \dots v'_m) \quad (s_1 \dots s_o) \sim_j (s'_1 \dots s'_p)$$

We analyze two cases.

Case j observes $E \cdot F$ and $F \cdot E$ and j and k observe $E \cdot F$: Let (t_1, \dots, t_w) and (t'_1, \dots, t'_x) be the elements of E in $(s_1 \dots s_o)$ and $(s'_1 \dots s'_p)$. From assumptions of the case we have that $w = x$ and also

$$\text{hide}(j) (t_i) = \text{hide}(j) (t'_i) \text{ for } i \in \{1, \dots, w\}$$

As $\sim_j^E \subseteq \approx_k^E$ we also have:

$$\text{hide}(k) (t_i) = \text{hide}(k) (t'_i) \text{ for } i \in \{1, \dots, w\}$$

Using this fact, we calculate as follows

$$\begin{aligned}
& (\text{filter } E \ F).\text{safe } (j, k) (v_1 \dots v_m) (s_1 \dots s_o) \\
& \text{iff } (v_1 \dots v_n) \approx_k \text{filter } E (s_1 \dots s_o) && \text{by definition of } (\text{filter } E \ F).\text{safe} \\
& \text{iff } \text{hide}(k) (v_1 \dots v_m) = \text{hide}(k) (t_1 \dots t_x) && \text{by definition of filter} \\
& \text{iff } \text{hide}(k) (v'_1 \dots v'_n) = \text{hide}(k) (t'_1 \dots t'_w) && \text{by symmetry and transitivity of } = \\
& \text{iff } (v_1 \dots v_n) \approx_k \text{filter } E (s'_1 \dots s'_p) && \text{by definition of filter} \\
& \text{iff } (\text{filter } E \ F).\text{safe } (j, k) (v'_1 \dots v'_n) (s'_1 \dots s'_p) && \text{by definition of } (\text{filter } E \ F).\text{safe}
\end{aligned}$$

and obtain the required equivalence.

Case j does not observe $E \cdot F$ and $F \cdot E$ or j or k do not observe $E \cdot F$: Then

$$(\text{filter } E \ F).\text{safe } (j, k) (v_1 \dots v_m) (s_1 \dots s_o) = \text{false} = (\text{filter } E \ F).\text{safe } (j, k) (v'_1 \dots v'_n) (s'_1 \dots s'_p)$$

immediately, which completes the proof. □