A Principled Approach to Tracking Information Flow in the Presence of Libraries

Daniel Hedin[1,2], Alexander Sjösten[1], Frank Piessens[3], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology
[2] Mälardalen University
[3] imec-DistriNet, KU Leuven

Abstract. There has been encouraging progress on information flow control for programs in increasingly complex programming languages, tracking the propagation of information from input sources to output sinks. Yet, programs are typically deployed in an environment with rich APIs and powerful libraries, posing challenges for information flow control when the code for these APIs and libraries is either unavailable or written in a different language.
This paper presents a principled approach to tracking information flow in the presence of libraries. With the goal to strike the balance between security and precision, we present a framework that explores the middle ground between the "shallow", signature-based modeling of libraries and the "deep", stateful approach, where library models need to be supplied manually. We formalize our approach for a core language, extend it with lists and higher-order functions, and establish soundness results with respect to the security condition of noninterference.

1 Introduction

The prevalent way to extend a language with functionality, e.g., to interact with its execution environment, is via libraries. As an example, consider a library that provides a collection of functions to provide the language with network capabilities. Since the language functionality in such cases is fundamentally extended, these libraries cannot be written in the language itself, but must be provided by some other means such as a *foreign function interface* (e.g. [27] in Java, [34] in Haskell and [30] in node.js) or via the execution environment.

Recently, there has been a growing interest in retrofitting libraries with *dynamic* execution monitors to provide additional runtime checks. One prominent example of this is *monitors for secure information flow* [15,1,18,17,3]. The interest in information flow control lies in the realization that access control is often not enough in cases when it is important what a program does with the information it has access to [31]. As an example, when a user enters credit card information into an application to perform a purchase, information flow control can guarantee that the credit card information is only used for the purpose of enabling the purchase (i.e., by passing the information to the payment provider) and is not being sent or gathered for illicit purposes.

Dynamic monitoring is similar to dynamic type checking, and works by augmenting the semantics of the language, with additional runtime information that provides an abstract view of the execution and enables enforcement of the desired properties. In the case of dynamic types, the additional information is a runtime representation of the types of values, and in the case of information flow control it is the security level.

In the presence of libraries written in another language, dynamic monitors face two important challenges: (i) the library is not able to work with values in the augmented semantics, and, more fundamentally, (ii) is not able to maintain the abstract view of the execution. With respect to the first challenge, some kind of marshaling must take place — this already occurs for the values of the language, but must be extended to first remove any additional runtime information. With respect to the second challenge, it is important that the removed runtime information is kept, in order to be able to reestablish the augmentation, once the library returns.

Thus, the challenges above translate to these pivotal questions:

(i) how should the runtime augmentation be removed when entities are passed from the monitored program into the unmonitored library, and
(ii) how should the runtime augmentation be reinstated when entities are passed from the unmonitored library to the monitored program.

On the surface, those questions may seem fairly straightforward, but prove surprisingly involved in the presence of common programming language features, such as structured data and higher-order functions.

In the work targeting secure information flow, one can identify two extremes with respect to library models [15,6,1,20,18,28,17,3]. On one hand are the *shallow models*, essentially corresponding to providing static boundary types, and on the other hand are the *deep models*, where the information flow inside the library is modeled in detail, frequently requiring a reimplementation of the library in the monitored semantics.

In JavaScript, already the standard API introduces information flow challenges. Consider, for instance, the following example, that makes use of the standard JavaScript function `Array.every` which, given a predicate, returns `true` if every element in the array on which `every` is called, is in the extension of the predicate.

```
[1,2,3,0,4,5].every(function(elem) { return elem > 0; })
```

In both JSFlow [17,16] and FlowFox [13,14], accurate modeling of many library functions, such as `Array.every`, requires hand-written, deep models. This is both labor-intensive and hard to maintain, not scaling to models for a rich set of libraries, as would be needed in a rich execution environment such as a browser or node.js [25,26,24]. For this reason, JSFlow attempts at providing a way of automatically wrapping libraries. However, JSFlow's approach is somewhat ad hoc and lacks formal underpinning. While for simple cases correctness is evident, it is unclear if this approach scales to more complex interactions with libraries such as for promises [22], e.g., when functions are passed to and from the library.

Contribution We investigate how to provide concise library models, in the setting of dynamic information flow control, for a small functional language. We present the development in a gradual way and investigate different programming language constructs in isolation, as extensions of a common core language. The modeling is such, that the results combine with relative ease. For space reasons, we limit ourselves to the treatment of structured data and higher-order functions. The main contributions of this paper are:

- a *split semantics* with *stateful marshaling* for a simple core;
- a split semantics with stateful marshaling for structured data in the form of lists and the concept of *lazy* marshaling;
- a split semantics for higher-order functions that introduces the concept of *abstract names*, enabling the connection between callbacks and *label models*.

The focus of this paper is on the stateful marshaling, leaving the label models relatively simple. The presented model does, however, allow for more advanced label models including (value) dependent models that harness the power coming from the knowledge of runtime values. We discuss possible extensions beyond the limitations of the provided label model language.

Outline The rest of the paper is laid out as follows. Section 2 introduces the core language and the notion of split semantics with stateful marshaling. Section 3 investigates lists in terms of an extension to the core language and introduces the notion of lazy marshaling. Section 4 investigates higher-order functions in terms of an extension to the core language and introduces the notion of abstract names. Finally, Section 5 discusses related work, and Section 6 discusses future work and concludes.

2 Core language $\mathcal{C}$

We present syntax and split semantics with stateful marshaling for a small core language. The notion of split semantics entails that a program is built up by two distinct parts: 1) the monitored program executing a labeled information flow aware semantics, and 2) the unmonitored library, executing an unlabeled standard semantics. For simplicity, the two parts of the program share syntax and semantics — the labeled semantics is an extension of the unlabeled. This is to keep the exposition small and the value-level marshaling to a minimum and is not a fundamental limitation of the approach.

2.1 Syntax

The syntax of the core language is defined as follows.

$$e ::= n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid f\ \boldsymbol{e} \mid f_{lib}\ \boldsymbol{e} \mid e_1 \oplus e_2$$

Let $\boldsymbol{x}$ denote a list of x, where $[\,]$ is the empty list and $\cdot$ is the cons operator. The top-level definitions, $d ::= f\ \boldsymbol{x} = e$, are restricted to function definitions, and *function models*, $m ::= f :: \boldsymbol{\varphi} \to \gamma$. A function model defines how labeled values

are marshaled to the unlabeled function, $\boldsymbol{\varphi}$, and how the unlabeled return value is marshaled back into the labeled world, γ, see below. All unlabeled functions called from the labeled world must have a corresponding function model.

A *program* is a triple, ($\boldsymbol{d}$, $\boldsymbol{d}$, $\boldsymbol{m}$), where the first component corresponds to the monitored program, the second component corresponds to the unmonitored library, and the third component is the *library model* consisting of function models. Execution starts in the *main* function of the monitored program. In the following, we refer to the monitored part of the program as the program, and the unmonitored library as the library.

The bodies of functions are made up of expressions, consisting of integers n, identifiers x and f (denoting functions), conditional branches, let bindings, function calls, library calls and binary operators $\oplus$. Library calls are not allowed in the library part of the program.

2.2 Semantics

As indicated above, $\mathcal{C}$ has two semantics, one *labeled* and one *unlabeled*. To distinguish between the two, without unnecessary notational burden, we use $\hat{X}$ to denote an entity in the labeled semantics corresponding to X in the unlabeled semantics.

Values The *labeled values*, $\hat{v}$, and *unlabeled values*, v, are defined as labeled and unlabeled integers respectively. The labels, ℓ, are taken from a two-point upper semi-lattice $L \sqsubseteq H$, where L denotes *low* ("public" when modeling confidentiality or "trusted" when modeling integrity) and H denotes *high* ("secret" when modeling confidentiality or "untrusted" when modeling integrity). While we focus on confidentiality throughout the paper, information flow integrity can be modeled dually [5].

$$\hat{v} ::= n^{\ell} \quad v ::= n$$

For labels let $\ell_1 \sqcup \ell_2$ denote the least upper bound of ℓ_1 and ℓ_2, and let $\hat{v}^{\ell_2} = v^{\ell_1 \sqcup \ell_2}$ for $\hat{v} = v^{\ell_1}$.

Stateful marshaling A function model defines how to marshal values between the program and the library in terms of the parameters and the return value, i.e., how to *unlabel* the parameters and *label* the result. Since the result is dependent on the parameters, it follows that the label of the result must be dependent on the labels of the parameters. For this reason, the removed labels must be stored for the duration of the library call in such a way that they can be used when relabeling the result. To achieve this, the unlabel process creates a *model state*[1], $\xi : \alpha \to \ell$, based on identifiers α, given by the unlabel model, φ. This model state is used in the labeling process in the interpretation of the label model, γ. The unlabel and label models follow the structure of the values, and are defined as follows for the core language

$$\varphi ::= \alpha \quad \gamma ::= \kappa$$

[1] Note that here, and in the following, for simplicity, we identify sets with the meta variables ranging over them.

$$\text{int}\frac{}{\delta \models n \rightsquigarrow n} \qquad \text{var}\frac{\delta[x] = v}{\delta \models x \rightsquigarrow v} \qquad \text{op}\frac{\delta \models e_1 \rightsquigarrow v_1 \quad \delta \models e_2 \rightsquigarrow v_2}{\delta \models e_1 \oplus e_2 \rightsquigarrow v_1 \oplus v_2}$$

$$\text{if}_1\frac{\delta \models e_1 \rightsquigarrow v \quad v \neq 0 \quad \delta \models e_2 \rightsquigarrow v}{\delta \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rightsquigarrow v} \qquad \text{if}_2\frac{\delta \models e_1 \rightsquigarrow v \quad v = 0 \quad \delta \models e_3 \rightsquigarrow v}{\delta \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rightsquigarrow v}$$

$$\text{let}\frac{\delta \models e_1 \rightsquigarrow v_1 \quad \delta[x \mapsto v_1] \models e_2 \rightsquigarrow v_2}{\delta \models \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \rightsquigarrow v_2} \qquad \text{app}\frac{\Delta[f] = (\boldsymbol{x}, e_f) \quad \delta \models \boldsymbol{e} \rightsquigarrow \boldsymbol{v} \quad [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e_f \rightsquigarrow v}{\delta \models f\ \boldsymbol{e} \rightsquigarrow v}$$

Fig. 1. Unlabeled semantics

where $\kappa ::= \alpha \mid \kappa_1 \sqcup \kappa_2 \mid \ell$ and the interpretation of κ in a model state ξ is given by

$$[\![\alpha]\!]_\xi = \begin{cases} L & \text{if } \xi[\alpha] \text{ is undefined} \\ \xi[\alpha] & \text{otherwise} \end{cases} \qquad [\![\ell]\!]_\xi = \ell \qquad [\![\kappa_1 \sqcup \kappa_2]\!]_\xi = [\![\kappa_1]\!]_\xi \sqcup [\![\kappa_2]\!]_\xi$$

From this, we define an unlabel operation, $v^\ell \downarrow \alpha$, and a label operation, $v \uparrow_\xi \kappa$, as follows

$$v^\ell \downarrow \alpha = (v, [\alpha \mapsto \ell]) \quad v \uparrow_\xi \kappa = v^{[\![\kappa]\!]_\xi}$$

The label operation takes an unlabeled value, v, a label model $\gamma = \kappa$ and a model state, ξ and labels the value in accordance with the interpretation of the label model in the model state. The unlabel operation takes a labeled value, $\hat{v}$, and an unlabel model, $\varphi = \alpha$, and returns an unlabeled value and a model state, ξ. The unlabel operation is lifted to sequences of values by chaining, in the following way, where $\amalg$ denotes disjoint union.

$$[\,] \downarrow [\,] = ([\,], [\,])$$
$$\hat{v} \cdot \hat{\boldsymbol{v}} \downarrow \varphi \cdot \boldsymbol{\varphi} = (v \cdot \boldsymbol{v}, \xi_1 \amalg \xi_2) \text{ where } \hat{v} \downarrow \varphi = (v, \xi_1) \text{ and } \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_2)$$

Unlabeled semantics Let the unlabeled variable environments, $\delta : x \to v$, be maps from identifiers to values, and let $\Delta : f \to (\boldsymbol{x}, e)$ be a map from identifiers to function definitions representing the unmonitored library. For simplicity we leave Δ implicit, since it is unmodified by the execution.

The unlabeled semantics, defined in Figure 1, is of the form $\delta \models e \rightsquigarrow v$, read, expression e evaluates to v in the unlabeled variable environment δ. For space reasons, since the unlabeled semantics is entirely standard, it is not explained further.

Labeled semantics Let the labeled variable environments, $\hat{\delta} : x \to \hat{v}$, be maps from identifiers to labeled values, let $\hat{\Delta} : f \to (\boldsymbol{x}, e)$ be a map from identifiers to function definitions representing the monitored program, and let $\Lambda : f \to (\boldsymbol{\varphi}, \gamma)$ represent the library model. The labeled semantics, defined in Figure 2, is of the form $\hat{\delta} \models e \to \hat{v}$, read, expression e evaluates to $\hat{v}$ in the labeled variable

$$\text{app}\frac{\hat{\Delta}[f] = (\boldsymbol{x}, e_f) \quad \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \quad [\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e_f \to \hat{v}}{\hat{\delta} \models f\ \boldsymbol{e} \to \hat{v}} \qquad \text{lib}\frac{\Delta[f] = (\boldsymbol{x}, e_f) \quad \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \quad \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \quad \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \quad [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e_f \rightsquigarrow v \quad v \uparrow_\xi \gamma = \hat{v}}{\hat{\delta} \models f_{lib}\ \boldsymbol{e} \to \hat{v}}$$

Fig. 2. Labeled semantics

environment $\hat{\delta}$. For space reasons, only the rules that differ from the unlabeled semantics in a non-standard way are included. The remaining rules propagate and compute with labels to reflect the dynamic information flow of the program and can be found in the full version of the paper [19]. Similarly to the unlabeled semantics we leave Δ, $\hat{\Delta}$, and Λ implicit.

Of the rules for the core language, `lib` is the only non-standard. It corresponds to the situation, where an unmonitored library function is called from the monitored semantics. Execution proceeds as follows. First, the function definition, $(\boldsymbol{x}, e_f)$, and the function model, $(\boldsymbol{\varphi}, \gamma)$, are found, then the parameters, $\boldsymbol{e}$, are evaluated to labeled values, $\hat{\boldsymbol{v}}$. Before being passed to the library, the labeled values are first unlabeled in accordance with the function model, resulting in unlabeled values, $\boldsymbol{v}$, and a model state, ξ. The body of the library function is evaluated in an environment $[\boldsymbol{x} \mapsto \boldsymbol{v}]$, where the formal parameters of the function maps to the corresponding arguments, and the result, v, is labeled in accordance with the function model, interpreted in the model state, ξ, produced by the previous unlabeling.

2.3 Correctness

We prove correctness under the assumption that the library model correctly models the library, i.e., that every modeled function in the library respects its function model. Semantically, we express this in terms of the execution of the library, the unlabeling of the parameters and the labeling of the result.

Definition 1 (Correctness of the library models) *A library model correctly models a library if every function, f, in the library, $\Delta[f] = (\boldsymbol{x}, e)$, respects the associated function model, $\Lambda[f] = (\boldsymbol{\varphi}, \gamma)$, if present.*

$$\begin{aligned}\forall f\ .\ \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \wedge \Delta[f] = (\boldsymbol{x}, e) \\ \wedge\ \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi) \\ \wedge\ [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e \rightsquigarrow v \wedge [\boldsymbol{x} \mapsto \boldsymbol{v}'] \models e \rightsquigarrow v' \Rightarrow v \uparrow_\xi \gamma \simeq v' \uparrow_\xi \gamma\end{aligned}$$

As is standard, we prove noninterference as the preservation of a low-equivalence relation under execution, defined as follows for values and labeled variable environments.

$$\frac{}{n^L \simeq n^L} \qquad \frac{}{n_1^H \simeq n_2^H} \qquad \frac{dom(\hat{\delta}) = dom(\hat{\delta}') \quad \forall x \in dom(\hat{\delta})\ .\ \hat{\delta}[x] \simeq \hat{\delta}'[x]}{\hat{\delta} \simeq \hat{\delta}'}$$

Under the assumption that Definition 1 holds, we can prove noninterference for labeled execution.

Theorem 1 (Noninterference for labeled execution)

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

Proof. By induction on the height of the derivation tree $\hat{\delta} \models e \to \hat{v}$. The proof of this and the other theorems is reported in the full version of this paper [19].

2.4 Examples

To illustrate how $\mathcal{C}$ can be used, we give two examples. The first example is the identity function.

```
id :: α → α
id x = x
```

The function model for `id` expresses that the label of the result should be the label of the parameter. This is computed by storing the label under the name α in the model state, when `id` is called, and then interpreting the α in the resulting model state, when the function returns.

The second example is the `min` function, which illustrates how more than one label can be stored into the model state.

```
min :: α1 α2 → α1 ⊔ α2
min x y = if x < y then x else y
```

Since the result of the `min` function is dependent on both parameters, the result should be the least upper bound of the labels of the parameters. To achieve this, both labels are stored in the model state on the call; the first label as α_1 and the second as α_2. The function model uses the label expression $\alpha_1 \sqcup \alpha_2$, which, when interpreted in the model state results in the least upper bound of the labels.

2.5 A note on the policy language

While we, in this work, strive to keep the model language simple, to enable us to study the processes of labeling and unlabeling vis-à-vis different language constructs, it is worthwhile to mention a few possible avenues for extensions. First, consider the following example, where the library function `f` calls the library function `min`. Instead of forcing the model of `f` to repeat the model of `min` it would be possible to add some form of *model application*, where the model of `min` is instantiated with the labels from `f`.

```
f :: α1 α2 → min α1 α2
f x y = min x y
```

This allows for a systematic construction of more complex models (nothing prevents us from introducing models that don't correspond to library functions).

Further, since the models are evaluated at runtime, they could be extended to have access to the *values* of the parameters in addition to the labels. This would allow for *dependent models*, where different labels are computed depending on the value of the parameters. Consider, for instance, the following library function.

```
f :: α1 α2 → x?α1 ⊔ α2 : α1
f x y = if x then y else 0
```

In this example the model uses the value of the parameter (stored in the model state under the parameter name) in order to select between two labels. In a language more complex than $\mathcal{C}$, those additions provide important expressiveness to the model language.

3 Lists $\mathcal{L}$

Structured data pose interesting challenges in relation to marshaling between the monitored and unmonitored semantics. While the unlabel and label processes must follow the structure of the values passed, structured data offer more freedom in the design of the unlabel and label models. In addition, fundamental questions pertaining to the time and extent of labeling and unlabeling arise. When passing a labeled list to the library, should the list be marshaled in a strict or a lazy fashion? For library functions that only use parts of the passed data, strict marshaling can be both expensive and potentially imprecise, in particular when large object graphs are passed to or from the library (cf., getting an object from the DOM, where strict marshaling would be prohibitively expensive).

For this reason, we explore the notion of lazy marshaling. The idea is to marshal only when the opposite program part actually makes use of the data that has been passed. Unlabeling (or labeling in the dual setting) occurs only when the library (dually, program) actually uses the data, and only the part of the data that was used is unlabeled. This requires us to be able to pass data in such a manner that we can trap any interaction and unlabel or relabel on the fly. To this end, we opt for a solution that is inspired by the Proxy objects of JavaScript [23] but cast in terms of lists, and use a representation of lists that allow for proxying. The approach is general in the sense that it scales well to other types of structural data and that it can be implemented in different ways, e.g., proxies and accessor methods, both available in a range of languages, including JavaScript, Python and Objective C. One limitation of the approach is that some form of programming language support, that allows for trapping the read and write interaction of the library with given objects, is needed. If such support is not available, one can always resort to strict marshaling, which corresponds to a relatively immediate lifting of the label and unlabel functions of the core language to structured data. Most of the ideas presented in this paper should carry over to strict marshaling with little effort at the cost of efficiency and precision of the marshaling.

3.1 Syntax

From a syntactic standpoint the extension of $\mathcal{C}$ to support lists is small; the empty list, [ ], the cons operation, :, and operations for getting the head, *head*, and tail, *tail*, of lists are added.

$$e ::= \ldots \mid [\,] \mid e : e \mid \mathit{head}\ e \mid \mathit{tail}\ e$$

3.2 Semantics

In JavaScript, a Proxy is an object that forwards all interactions to a set of user defined functions, provided at the creation time of the Proxy. Once the Proxy object has been created, it can be interacted with like a normal object. Thus, e.g., by defining a function corresponding to *get*, all property reads of the proxy object can be trapped and modified — the return value of the function will be the result of the read. The fundamental property that makes Proxies suitable for lazy marshaling is that they allow the functions to modify all possible interactions with the object.

Unlike the strict marshaling of the core language, where the model state is computed before entering the library, the introduction of lazy marshaling requires the model state to be updated during the execution of the library function (in case the function interacts with the passed data). In a practical setting, the monitored program and the unmonitored library would share memory (they are different parts of the same program). This means that it is easy to maintain the model state in the presence of lazy marshaling. In an operational semantics, mutable state is modeled by threading the state through the evaluation.

Values We model proxyable lists as pairs of functions $(\hat{H}, \hat{T})$ and (H, T) respectively.

$$\hat{v} ::= n^{\ell} \mid (\hat{H}, \hat{T})^{\ell} \mid [\,]^{\ell} \quad v ::= n \mid (H, T) \mid [\,]$$

The idea is that $\hat{H}$ and H return the head of the list, and $\hat{T}$ and T return the tail (which can be the empty list). This representation allows for an elegant lazy marshaling of lists, when they are passed between the program and the library, by wrapping the head and tail functions. The actual marshaling takes place only when the function is called, i.e., when the respective value is read.

Stateful marshaling In order to support unlabeling and labeling of lists we must extend the unlabel and label models. Since we are mainly interested in the stateful marshaling, we use a simple extension that differentiates between the labels of the values and the label of the structure of the lists [18]. See Section 3.5 for a discussion on possible extensions.

$$\varphi ::= \alpha \mid [\ \varphi\ ]_{\alpha} \quad \gamma ::= \kappa \mid [\ \gamma\ ]_{\kappa}$$

The intuition for unlabel models is that, whenever a value is read from the list, the model state is updated accordingly. This means that the model state can be changed during the execution of the library, which must be reflected in the unlabeled semantics. The same is not true for the labeled semantics; any value passed from the unlabeled world will be labeled with respect to the model state at the time of return, even if the labeling is lazy. This leads to a seeming asymmetry in the semantics reflected by the definition of the head and tail functions for lists.

$$\hat{H} : () \to \hat{v} \quad \hat{T} : () \to \hat{v} \qquad H : \xi \to (\xi, v) \quad T : \xi \to (\xi, v)$$

The way to interpret this asymmetry is not that the unlabeled semantics has to be changed to enable marshaling — as described above, mutable state is modeled

by threading the state through the computation. Rather, the asymmetry arises from the fact that the model state is only important for the evaluation of library functions called from the monitored semantics.

With respect to the unlabel and label operations, they must be updated to handle the extended unlabel and label models.

$$\begin{aligned}[\ ]^{\ell} \downarrow [\ \varphi \ ]_{\alpha} &= ([\ ], [\alpha \mapsto \ell]) \\ (\hat{H}, \hat{T})^{\ell} \downarrow [\ \varphi \ ]_{\alpha} &= ((\mathrm{unlabel}(\hat{H}, \varphi), \mathrm{unlabel}(\hat{T}, [\ \varphi \ ]_{\alpha})), [\alpha \mapsto \ell])\end{aligned}$$

The unlabeling of lists updates the structure label and wraps the head and tail of the list (if present) with unlabeling wrappers, that unlabel with respect to the unlabel model. On access the wrapper receives the model state (of the current call to the library), after which it uses $\hat{H}$ to get the labeled value, and φ to unlabel. The unlabeled value is returned together with an updated model state, where $\xi \sqcup \xi'$ is defined as the union of ξ and ξ' under least upper bound of shared mappings. The wrapper for the tail of the list works analogously, but with respect to the full unlabel model of the list $[\ \varphi \ ]_{\alpha}$.

$$\begin{aligned}&\mathrm{unlabel}(\hat{H}, \varphi) = \lambda\xi \ . \ (\xi \sqcup \xi', v), \\ &\quad \text{where } \hat{H}() = \hat{v} \text{ and } \hat{v} \downarrow \varphi = (v, \xi')\end{aligned} \qquad \begin{aligned}&\mathrm{unlabel}(\hat{T}, [\ \varphi \ ]_{\alpha}) = \lambda\xi \ . \ (\xi \sqcup \xi', v), \\ &\quad \text{where } \hat{T}() = \hat{v} \text{ and } \hat{v} \downarrow [\ \varphi \ ]_{\alpha} = (v, \xi')\end{aligned}$$

The labeling of lists is similar, with the difference that the labeling is done with respect to the final model state. Once evaluation has returned, nothing can change the model state corresponding to the call.

$$\begin{aligned}[\ ] \uparrow_{\xi} [\ \gamma \ ]_{\kappa} &= [\ ]^{[\![\kappa]\!]_{\xi}} \\ (H, T) \uparrow_{\xi} [\ \gamma \ ]_{\kappa} &= (\mathrm{label}(H, \xi, \gamma), \mathrm{label}(T, \xi, [\ \gamma \ ]_{\kappa}))^{[\![\kappa]\!]_{\xi}}\end{aligned}$$

The wrappers are given the model state, ξ, and the label model, γ. On access the wrapper uses H to get the unlabeled value, v. Notice, how this may actually extend the model state to ξ' (it could be the case that H is an unlabel wrapper) and that ξ' is used together with γ to compute a label for v. This new model state does not have to be propagated, though. If the value was used by the unlabeled world in the creation of the tail of the list its label is already included in ξ.

The relabeling of the tail of the list works analogously, but with respect to the label model of the list $[\ \gamma \ ]_{\kappa}$. Any extension of the model state is passed to the wrapping of the tail.

$$\begin{aligned}&\mathrm{label}(H, \xi, \gamma) = \lambda() \ . \ \hat{v}, \\ &\quad \text{where } H(\xi) = (\xi', v) \text{ and } v \uparrow_{\xi'} \gamma = \hat{v}\end{aligned} \qquad \begin{aligned}&\mathrm{label}(T, \xi, [\ \gamma \ ]_{\kappa}) = \lambda() \ . \ \hat{v}, \\ &\quad \text{where } T(\xi) = (\xi', v) \text{ and } v \uparrow_{\xi'} [\ \gamma \ ]_{\kappa} = \hat{v}\end{aligned}$$

Unlabeled and labeled semantics The additions to the labeled semantics, found in Figure 3, are straightforward given the above modeling. Let $\mathrm{lcons}(\hat{v}_1, \hat{v}_2) = (\lambda(). \ \hat{v}_1, \lambda(). \ \hat{v}_2)$ be the creation of labeled cons cells[2], used in the evaluation of the : operator (`cons`). The evaluation of head and tail (`head`, and `tail`) uses the

[2] The term originates from Lisp. In addition, cons is used as the name for the list-forming operator in many functional languages.

$$\text{empty}\frac{}{\hat{\delta} \models [\,] \to [\,]^L} \qquad \text{cons}\frac{\hat{\delta} \models e_1 \to \hat{v}_1 \quad \hat{\delta} \models e_2 \to \hat{v}_2}{\hat{\delta} \models e_1 : e_2 \to \text{lcons}(\hat{v}_1, \hat{v}_2)^L}$$

$$\text{head}\frac{\hat{\delta} \models e \to (\hat{H}, \hat{T}) \quad \hat{H}() = \hat{v}}{\hat{\delta} \models \mathit{head}\ e \to \hat{v}} \qquad \text{tail}\frac{\hat{\delta} \models e \to (\hat{H}, \hat{T}) \quad \hat{T}() = \hat{v}}{\hat{\delta} \models \mathit{tail}\ e \to \hat{v}}$$

$$\text{lib}\frac{\begin{array}{c}\Delta[f] = (\boldsymbol{x}, e_f) \qquad \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \qquad \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \\ \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \quad [\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi, e_f \rangle \rightsquigarrow \langle \xi', v \rangle \quad v \uparrow_{\xi'} \gamma = \hat{v}\end{array}}{\hat{\delta} \models f_{lib}\ \boldsymbol{e} \to \hat{v}}$$

Fig. 3. Labeled semantics of lists

$$\text{empty}\frac{}{\delta \models \langle \xi, [\,] \rangle \rightsquigarrow \langle \xi, [\,] \rangle} \qquad \text{cons}\frac{\begin{array}{c}\delta \models \langle \xi_1, e_1 \rangle \rightsquigarrow \langle \xi_2, v_1 \rangle \\ \delta \models \langle \xi_2, e_2 \rangle \rightsquigarrow \langle \xi_3, v_2 \rangle\end{array}}{\delta \models \langle \xi_1, e_1 : e_2 \rangle \rightsquigarrow \langle \xi_3, \text{ucons}(v_1, v_2) \rangle}$$

$$\text{head}\frac{\begin{array}{c}\delta \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, (H, T) \rangle \\ H(\xi_2) = (\xi_3, v)\end{array}}{\delta \models \langle \xi_1, \mathit{head}\ e \rangle \rightsquigarrow \langle \xi_3, v \rangle} \qquad \text{tail}\frac{\begin{array}{c}\delta \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, (H, T) \rangle \\ T(\xi_2) = (\xi_3, v)\end{array}}{\delta \models \langle \xi_1, \mathit{tail}\ e \rangle \rightsquigarrow \langle \xi_3, v \rangle}$$

Fig. 4. Unlabeled semantics of lists

head and the tail function respectively to get the value. Notice, how the model state may be modified during the execution of the library, and how the return value is labeled in the modified state (`lib`).

With respect to the unlabeled semantic, the entire semantics must be lifted to thread the model state, $\delta \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, v \rangle$. This modification is straightforward and omitted for space reasons but can be found in the full version [19]. The additions to the unlabeled semantics are found in Figure 4. Let $\text{ucons}(v_1, v_2) = (\lambda\xi\ .\ (\xi, v_1), \lambda\xi\ .\ (\xi, v_2))$ be the creation of unlabeled cons cells, used in the evaluation of the : operator (`cons`). The evaluation of head and tail (`head`, and `tail`) uses the head and tail function respectively to get the value. Notice that the model state is threaded in this case — this is what allows for the lazy unlabeling. In case the head or tail function is an unlabel wrapper, the state will be updated.

3.3 Correctness

Definition 2 (Correctness of the library models) *A library model correctly models a library if every function, f, in the library, $\Delta[f] = (\boldsymbol{x}, e)$, respects the associated function model, $\Lambda[f] = (\boldsymbol{\varphi}, \gamma)$, if present. Notice that, even though the final model states may differ (due to different interactions with marshaled labeled values in the two runs), a correct library model must ensure that the label is independent on the differences and that the values are low-equivalent with*

respect to the labeling.

$$\begin{aligned}\forall f \;.\; \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \wedge \Delta[f] = (\boldsymbol{x}, e) \\ \wedge\, \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_1) \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi_1) \\ \wedge\, [\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, v \rangle \wedge [\boldsymbol{x} \mapsto \boldsymbol{v}'] \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2', v' \rangle \Rightarrow \\ v \uparrow_{\xi_2} \gamma \simeq v' \uparrow_{\xi_2'} \gamma\end{aligned}$$

As is standard we prove noninterference as the preservation of a low-equivalence relation under execution, extended from Section 2.3 with lists as follows.

$$\frac{}{[\,]^L \simeq [\,]^L} \qquad \frac{}{v_1^H \simeq v_2^H} \qquad \frac{\hat{H}() \simeq \hat{H}'() \quad \hat{T}() \simeq \hat{T}'()}{(\hat{H}, \hat{T})^L \simeq (\hat{H}', \hat{T}')^L}$$

Under the assumption that Definition 2 holds, we can prove noninterference for labeled execution.

Theorem 2 (Noninterference for labeled execution)

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

3.4 Examples

We present a selection of examples to illustrate different aspects of our models. Consider first the `length` function, that recursively computes the length of the given list.

```
length :: [ α1 ]α2 → α2
length l = if l == [] then 0 else 1 + length (tail l)
```

The function traverses the list until the empty list is found without looking at the elements. During this traversal, the security labels corresponding to the cons cells are accumulated into the label variable α_2, which is used to label the result. This corresponds precisely to the structure security label of lists in [18]. It is, thus, possible to have functions that are dependent on the structure of a list, but not the content.

The other way, however, is not possible. Getting an element from a list always reveals information about the structure of the list. Thus, the `sum` function, which sums the element of the list must also take the labels of the cons cells into account.

```
sum :: [ α1 ]α2 → α1 ⊔ α2
sum l = if l == [] then 0 else head l + sum (tail l)
```

Consider the function `replicate`, that creates a list by replicating a given element, x, n times. The length of the list is given by the label of n and the label of the elements by the label of x. Notice the limitation in the current label models. By giving the second argument the unlabel model α_2, we force `replicate` to take integers — lists cannot be unlabeled by α_2. In such cases, *polymorphic models* are needed, see below in Section 3.5.

```
replicate :: α1 α2 → [ α2 ]α1
replicate n x = if n == 0 then []
                else x : replicate (n - 1) x
```

Related to both `sum` and `replicate` consider the function `take`, that takes an integer, n, and a list, l, and returns the n first elements of l. Clearly, the length of the list is dependent on both the label of n, α_1, and the structure of the list α_3. Notice, that the label of the structure of the list is accumulated into α_3 as the function traverses the list. This means that, given a list, where the first k cons cells are public, followed by some number of secret cons cells, `take` will yield lists with public structure, as long as no more than k elements are taken. Once more than k elements are taken, however, the labels of all cons cells will be secret. Unfortunately, this is the same for the labels of the values, which are all joined into α_2, see Section 3.5.

```
take :: α1 [ α2 ]α3 → [ α2 ]α1⊔α3
take n l = if l == [] || n == 0 then []
           else head l : take (n - 1) (tail l)
```

Finally, consider the function `takeUntilZero`, that takes an unknown number of elements from the list. In this function, the length of the list is dependent on the labels of the values of the list, as well as the labels of the traversed cons cells. As before, only the labels of the cons cells that actually take part in the computation are part of the accumulated label for α_2.

```
takeUntilZero :: [ α1 ]α2 →[ α1 ]α1⊔α2
takeUntilZero l = if l == [] || head l == 0 then []
                  else head l : takeUntilZero (tail l)
```

3.5 A note on the policy language

With respect to the policy language, there are a number of possible paths to explore. First, consider a form of polymorphic models, where we add variables, x, to the policy language. Unlike α, the intention is that x can map to structured labels (potentially in combination with the values, see Section 2.5). This would enable the following.

```
replicate :: α x → [ x ]α
replicate n x = if n == 0 then []
                else x : replicate (n - 1) x
```

where x would allow any type of value to be repeated. It is also possible to envision other operations on such variables, such as $@x$, the computation of the least upper bound of the labels reachable from x.

Additionally, it is natural to extend the model language with some form of pattern matching on lists, as follows.

```
f :: (α1 : α2 : [ α3 ]α4) → α3 ⊔ α4
f ls = sum (drop 2 ls)
```

In this case, the first two elements are dropped before the remainder is summed together. An interesting avenue of research is to explore this in combination with dependent models and richer models for building structured data.

4 Higher-order functions $\mathcal{F}$

After having investigated how to pass structured and unstructured data between the program and the library, we turn the attention to the passing of computations, in terms of higher-order functions. The passing of functions between programs and libraries is commonplace, used in the presence of, e.g., asynchronous operations. Examples of this are *callbacks*, where functions are passed to the library, allowing it to inform the program of certain events, and promises [22], that rely on the ability to pass functions in both directions.

4.1 Syntax

To investigate higher-order functions, we extend the core language with a function expression, $\mathit{fun}\ \boldsymbol{x} \Rightarrow e$ and change function calls to a computed call target. The introduction of higher-order functions subsumes top-level function definitions. Instead, we allow for top-level *let* declarations, $\mathit{let}\ x = e$, and corresponding model declarations, $x :: \gamma$.

$$e ::= \ldots \mid e\ \boldsymbol{e} \mid \mathit{fun}\ \boldsymbol{x} \Rightarrow e \qquad d ::= \mathit{let}\ x = e \qquad m ::= x :: \gamma$$

4.2 Semantics

Fundamentally, we use the same approach as with lists and represent closures as functions instead of structured values. This allows us to marshal functions from the labeled world to the unlabeled world and back without the need to distinguish between the origin of the values in the respective semantics. Intuitively, this corresponds to using functions as the calling convention and mimics what is actually in a practical implementation[3].

Following the development of Section 3, we add functional closures to the values as follows.

$$\hat{v} ::= n^{\ell} \mid \hat{F}^{\ell} \quad v ::= n \mid F$$

where labeled closures, $\hat{F}$, take sequences of labeled values to labeled values and unlabeled closures, F, also thread a model state

$$\hat{F} : \hat{\boldsymbol{v}} \to \hat{v} \quad F : (\xi, \boldsymbol{v}) \to (\xi, v)$$

With respect to the asymmetry of the semantics, the intuition is the same as before: the model state resides in shared memory, but, since the labeled semantics never modifies the model state we do not need to thread the model state through the labeled semantics.

[3] In a practical implementation, the program and the library would use the calling convention of the computer — regardless of the implementation language of the two.

Stateful marshaling Conceptually, any function defined in the library that can be called from the monitored program, whether passed as a closure or called, must be given a label model, that defines how to label the closure as a value, how to unlabel the parameters and label the result (c.f., the function models in Section 2). The question is, how to unlabel a closure, when passing it from the monitored program to the library. Intuitively, the unlabel model should be the dual of the label model, i.e., unlabel the closure as a value, label the parameters and unlabel the result. The problem is, that both unlabeling and labeling is performed in relation to a model state, which cannot be assumed to be the same as when the closure was passed as a parameter (it could be an extension — the passed closure could be called from an inner function). For this reason, we cannot tie an unlabel model to the closure at the point of unlabeling; it must be provided at the point of call. To be able to connect closures to calls, closures are tagged with a provided abstract identifier, π, when unlabeled. This abstract identifier is used in the label models for library functions to connect called closures with *call models* that express how to label the parameters and unlabel the result in the model state of the caller.

$$\varphi ::= \alpha \mid \pi^{\alpha} \quad \gamma ::= \kappa \mid (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} \quad \zeta ::= \pi\ \boldsymbol{\gamma} \to \varphi$$

Unlabel models for labeled closures, π^{α}, provide both abstract identifiers, π, and label variables, α, while the label models of unlabeled closures, $(\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa}$, contain how to label the closure as a value, κ, how to unlabel the parameters, $\boldsymbol{\varphi}$, how to label the result, γ, and how to label calls to callbacks, $\boldsymbol{\zeta}$. These call models, ζ, tie abstract identifiers, π, to call models, i.e., how to label the parameters, $\boldsymbol{\gamma}$, and how to unlabel the result, φ. Linked by the abstract identifier, the unlabel model for labeled closures together with the call models can be seen as duals to the label models for unlabeled closures.

Unlabeling of labeled closures is similar to unlabeling of values and lists, and places an unlabel wrapper around the labeled closure. The unlabel wrapper is, additionally, given the abstract identifier, π, used to tie future calls to the corresponding call models.

$$v^{\ell} \downarrow \alpha = (v, \xi[\alpha \mapsto \ell]) \qquad \hat{F}^{\ell} \downarrow \pi^{\alpha} = (\text{unlabel}(\hat{F}^{\ell}, \pi), [\alpha \mapsto \ell])$$

The unlabel wrapper becomes an unlabeled closure, that takes a model state, ξ, and a sequence of unlabeled values, $\boldsymbol{v}$, and finds the call model $\boldsymbol{\gamma} \to \varphi$ corresponding to the abstract identifier, π. Thereafter, $\boldsymbol{\gamma}$ is used to label the values, which are passed to the labeled closure, $\hat{F}$, to get a labeled value, $\hat{v}$. The labeled value is unlabeled using φ, which produces an unlabeled value and an update to the model state, ξ'. The result of the call to the wrapper is an updated model state and the unlabeled value. Notice how the label of the closure ℓ is used to raise the returned value before the unlabeling.

$$\begin{aligned}&\text{unlabel}(\hat{F}^{\ell}, \pi) = \lambda(\xi, \boldsymbol{v})\ .\ (\xi \sqcup \xi', v),\\ &\quad \text{where } \xi[\pi] = \boldsymbol{\gamma} \to \varphi \text{ and } \hat{F}(\boldsymbol{v} \uparrow_{\xi} \boldsymbol{\gamma}) = \hat{v} \text{ and } \hat{v}^{\ell} \downarrow \varphi = (v, \xi')\end{aligned}$$

$$\text{fun}\frac{v = \text{lclos}(\hat{\delta}, \boldsymbol{x}, e)}{\hat{\delta} \models \mathit{fun}\ \boldsymbol{x} \Rightarrow e \to v^L} \qquad \text{app}\frac{\hat{\delta} \models e \to \hat{F}^\ell \quad \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \quad \hat{F}(\hat{\boldsymbol{v}}) = \hat{v}}{\hat{\delta} \models e\ \boldsymbol{e} \to \hat{v}^\ell}$$

$$\text{lib}\frac{\delta_0[f] = F \quad \xi_0[f] = (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa \quad F \uparrow_{\xi_0} (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa = \hat{F}^\ell}{\hat{\delta} \models f_{lib} \to \hat{F}^\ell}$$

Fig. 5. Labeled semantics for higher-order functions

Labeling of unlabeled closures places a label wrapper around the closure. The label wrapper is additionally given the model state, ξ, how to unlabel the parameters, $\boldsymbol{\varphi}$, how to label return value, γ, and the call models, $\boldsymbol{\zeta}$.

$$v \uparrow_\xi \kappa = v^{[\![\kappa]\!]_\xi} \qquad F \uparrow_\xi (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa = \text{label}(F, \xi, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{[\![\kappa]\!]_\xi}$$

The label wrapper becomes a labeled closure, that takes a sequence of labeled values, $\hat{\boldsymbol{v}}$, unlabels the value producing a sequence of values, $\boldsymbol{v}$, and an update to the model state, ξ'. The updated model state is extended with the call models of the function (replacing the previously defined), producing a new model state ξ_2 by threading

$$[\![\pi\ \boldsymbol{\kappa} \to \varphi]\!]_\xi = \xi[\pi \mapsto (\boldsymbol{\kappa} \to \varphi)]$$

through the sequence $\boldsymbol{\zeta}$. The produced model state is used in the execution of the unlabeled closure, F, together with the unlabeled values producing an unlabeled value, v, and the final model state, ξ_3. The result is the labeled value $\hat{v}$, created by labeling v with respect to γ and the final model state.

$$\begin{aligned}&\text{label}(F, \xi, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta}) = \lambda \hat{\boldsymbol{v}}\ .\ \hat{v},\\ &\quad \text{where } \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi') \text{ and } [\![\boldsymbol{\zeta}]\!]_{\xi \sqcup \xi'} = \xi_2\\ &\quad \text{and } F(\xi_2, \boldsymbol{v}) = (\xi_3, v) \text{ and } v \uparrow_{\xi_3} \gamma = \hat{v}\end{aligned}$$

Labeled semantics The labeled semantics is mostly unaffected by the extension, apart from the rule for higher-order functions (`fun`), the rule for function call (`app`) and the rule for library call (`lib`). The modified rules are found in Figure 5 and make use of closure creation, lclos, defined as follows.

$$\text{lclos}(\hat{\boldsymbol{\delta}}, \boldsymbol{x}, e) = \lambda \hat{\boldsymbol{v}}\ .\ \hat{v}, \text{where } \hat{\delta}[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e \to \hat{v}$$

In the semantics $\hat{\delta}_0$, and δ_0 are created by evaluating the top levels of the labeled and the unlabeled world, respectively. This creates all top level closures used in function and library calls. Similarly, ξ_0 is created from the model definitions of the library, and is used as the initial model state.

Function call (`app`) evaluates the function expression to a closure and the parameters to a sequence of labeled values, $\hat{\boldsymbol{v}}$. The closure is called by supplying the labeled values and the result is returned, but with the label raised to the label of the closure. The library call has been replaced with a rule that lifts an

$$\text{fun}\frac{v = \text{uclos}(\boldsymbol{\delta}, \boldsymbol{x}, e)}{\boldsymbol{\delta} \models \langle \xi, \mathit{fun}\ \boldsymbol{x} \Rightarrow e \rangle \rightsquigarrow \langle \xi, v \rangle}$$

$$\text{app}\frac{\boldsymbol{\delta} \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, F \rangle \quad \boldsymbol{\delta} \models \langle \xi_2, \boldsymbol{e} \rangle \rightsquigarrow \langle \xi_3, \boldsymbol{v} \rangle \quad F(\xi_3, \boldsymbol{v}) = \langle \xi_4, v \rangle}{\boldsymbol{\delta} \models \langle \xi_1, e\ \boldsymbol{e} \rangle \rightsquigarrow \langle \xi_4, v \rangle}$$

Fig. 6. Unlabeled semantics for higher-order functions

.

unlabeled closure to the labeled world (`lib`). This is done by looking up the unlabeled closure in the initial environment of the library δ_0, and the corresponding function model in the initial model state ξ_0. The labeled (wrapped) closure is then returned as the result. Thus, in line with the intuition of using functions as the calling convention, functions in the program and in the library are translated to functions that are called in the same manner in the function call rule.

Unlabeled semantics In the unlabeled semantics, a rule for higher-order functions (`fun`) has been added and the rule for function application (`app`) has been changed. The modified rules are found in Figure 6 and are analogous with the changes made to the labeled semantics, including the use of closure creation defined as follows.

$$\text{uclos}(\boldsymbol{\delta}, \boldsymbol{x}, e) = \lambda(\xi_1, \boldsymbol{v})\ .\ (\xi_2, v), \text{where } \delta[\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_1, e \rangle \rightarrow \langle \xi_2, v \rangle$$

4.3 Correctness

We prove correctness under the assumption that the library model correctly models the library.

Definition 3 (Correctness of the library models) *A library model correctly models a library if every closure, f, in the library, $\delta_0[f] = F$, respects the associated function model, $\xi_0[f] = (\boldsymbol{\varphi} \rightarrow \gamma, \boldsymbol{\zeta})^{\kappa}$, if present.*

$$\begin{aligned}\forall f\ .\ \xi_0[f] = (\boldsymbol{\varphi} \rightarrow \gamma, \boldsymbol{\zeta})^{\kappa} \wedge \delta_0[f] = F \\ \wedge\ \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_1) \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi_1) \wedge [\![\boldsymbol{\zeta}]\!]_{\xi_1} = \xi_2 \wedge \\ F(\xi_2, \boldsymbol{v}) = (\xi_3, v) \wedge F(\xi_2, \boldsymbol{v}') = (\xi_3', v') \wedge \Rightarrow v \uparrow_{\xi_3} \gamma \simeq v' \uparrow_{\xi_3'} \gamma\end{aligned}$$

As is standard we prove noninterference as the preservation of a low-equivalence relation under execution, extended from Section 2.3 with higher-order functions as follows.

$$\frac{}{v_1^H \simeq v_2^H} \qquad \frac{\forall \hat{v}, \hat{v}'\ .\ \hat{v} \simeq \hat{v}' \Rightarrow \hat{F}(\hat{v}) \simeq \hat{F}'(\hat{v}')}{\hat{F}^L \simeq \hat{F}'^L}$$

Under Definition 3 holds, we can prove noninterference for labeled execution.

Theorem 3 (Noninterference for labeled execution)

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \rightarrow \hat{v} \wedge \hat{\delta}' \models e \rightarrow \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

4.4 Examples

To illustrate models for higher-order functions we consider three examples. In the examples, the library top-level contains a let with a higher-order function, which is paired with a function model. Before the program is run the top-level let bindings in the library and the unmonitored program (in that order) is evaluated to values. As illustrated in the second example, this means that execution no longer needs to start in a predefined function. Instead, computation can be started from any of the let bindings that do not produce closures.

The first example takes a callback and immediately calls it with a constant, and the associated function model expresses that the function takes a closure, which will be unlabeled as α_1 and associated with the abstract name x (nothing prevents us from using the same name as the parameter). Further, the closure is called with a public parameter, and the result will be unlabeled as α_2, which is also the label of the result of the function.

```
f :: (x^α1 -> α2, x L -> α2)^L
let f = fun x => x 42
```

When calling the closure, the call model will be looked up and used to label the parameters — in this case giving 42 labeled with L. The result of the call will be unlabeled as α_2, before being labeled by α_2 and returned by the function.

The second example illustrates why callbacks cannot be associated with an unlabel model on the point of unlabeling.

```
let cb = fun x => x + 1
let main = let g = f_lib cb in g 10

-- library part
f :: (x^α1 -> (α2 -> α3, x α2 -> α3)^L)^L
let f = fun x => fun y => x y
```

When the callback `cb` is passed to `f` it is not called, rather a closure is returned which takes another parameter that is unlabeled into α_2, which in turn is used as the parameter to the callback. Thus, in order to correctly label the value of the parameter to the callback, α_2 must be in the model state. This is true for the second call `g 10` but not for the first $\texttt{f}_{lib}$ `cb` in the monitored program.

Finally, consider an example with a conditional callback.

```
f :: (x^α1 -> (α2 -> α2 ⊔ α3, x α2 -> α3)^L)^L
let f = fun x => fun y => if y then x 42 else 42
```

The example illustrates the situation, where the callback may or may not be called depending on other values inspired by the frequent use of *coercions* in JavaScript libraries. This means that in some executions the variable α_2 may not be set. To handle this kind of situations it suffices that $[\![\alpha]\!]_\xi = L$, when $\xi[\alpha]$ is undefined. In addition, this interpretation allows for a limited form of dependent models.

5 Related work

There has been a substantial body of work in the area of dynamic information flow control in the past decade, to a large extent motivated by the desire to provide security and privacy for JavaScript web applications. There are two big lines of work. First, execution monitors [15,1,18,17,3] attach additional metadata (for instance, a security level) and propagate that metadata during the execution of a program. Second, multi-execution based approaches [6,20,28] essentially execute a program multiple times, and make sure that the execution that performs outputs at a certain security level has only seen information less than or equal to that security level. The multiple-facets approach [2] is an optimized implementation of multi-execution, but it is less transparent. Bielova and Rezk [4] give a detailed survey and comparison of all kinds of dynamic information flow mechanisms, and we refer the reader to that paper for a detailed discussion. Both lines of work on dynamic information flow control (execution monitoring and multi-execution) have been applied to JavaScript in the browser [13,16], and both have dealt with the problem of interfacing with libraries in a relatively ad-hoc way — essentially by manual programming of models of the library functions, or by treating API calls as I/O operations [14]. Rajani et al. [29] propose detailed and rigorous formal models of the DOM and event-handling parts of the browser, and find several potential information leaks. The work in this paper is a first step to a more principled approach of interfacing with such libraries that avoids the labor-intensive manual construction of such models (at the cost of potentially losing some precision).

The problem of interfacing with libraries where no dynamic checking of information flow control is possible, is related to the problem of checking contracts at the boundary between statically type-checked code and dynamically type-checked code. The problem of checking such contracts has been studied extensively in higher-order programming languages. Findler and Felleisen pioneered this line of work and proposed higher-order contracts [11]. The main challenge addressed is that of function values passed over the boundary. Compliance of such function values with their specified contract is generally undecidable. But it can be handled by wrapping the function with a wrapper that will check the contract of the function value at the point where the function is called. This is similar to how we handle function values in this paper, and an interesting question for future work is whether we can avoid the use of abstract identifiers for closures by injecting the appropriate labeling/unlabeling functionality using proxies only guided by how this is done in higher-order contract checking [8]. One concern that has received extensive attention is the proper assignment of *blame* once a contract violation is detected [12,7]. Assigning blame for information flow violations has been investigated by King et al. [21] in the setting of static information flow checking. Our work could be seen as an application of the idea of dynamic higher-order contract checking to information flow contracts, something that to the best of our knowledge has not yet been considered before. We do not consider the issue of assigning blame: if the library does not comply with the specified contract, this is not detected at run-time.

Gradual typing [32,33] is an approach to support the evolution of dynamically typed code to statically typed code, and it shares with our work the challenge of interfacing soundly between the dynamically checked part of the program and the statically checked part that no longer propagates all run-time type information. It has also been applied in the setting of security type systems [9,10], but it fundamentally differs in objective from our work. With gradual typing, the idea is to start from a program that is checked dynamically, and to gradually grow the parts that are statically checked. Our objective is to support interfacing with parts of the program for which dynamic checking is infeasible, either because the part is written in another language like C, or because dynamic checking would be too expensive to start with.

6 Conclusion

In this paper we have explored a method, *stateful marshaling*, that enables an information flow monitored program to call unmonitored libraries. The approach relies on storing the labels in a *model state* in accordance with an *unlabel model* before calling the library, and labeling the returned result by interpreting a *label model* in that model state.

Additionally, we have investigated *lazy marshaling* of structured data in terms of lists. The idea is similar to the concept of proxies and works by semantically representing lists as pairs of functions, that can be wrapped without recursively marshaling the entire list. When interacted with, the wrappers unlabel one step and return unlabeled primitive values or new lazy wrappers.

Finally, using functions to represent closures, we have shown how higher-order functions can be allowed to be passed in both directions. The approach relies on the concept of *abstract identifiers* that tie labeled closures, passed from the monitored program to the library, to call models, which describe how to label the parameters and unlabel the result with respect to the model state of the caller.

Future work We have preliminary results that show that lazy marshaling in combination with abstract identifiers is able to successfully handle references and the challenging combination of references and higher-order functions. Further, as discussed above, we aim to explore richer model languages, including but not limited to dependent models and model polymorphism. Finally, experiments with integrating our approach into JSFlow are subject to our current and future work.

Acknowledgments This work was partly funded by the European Community under the ProSecuToR project and the Swedish research agency VR.

References

1. T. H. Austin and C. Flanagan. Permissive Dynamic Information Flow Analysis. In *PLAS*, 2010.

2. T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL*, 2012.
3. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *POST*, 2014.
4. N. Bielova and Tamara Rezk. A taxonomy of information flow monitors. In *POST*, 2016.
5. A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *ICISS*, 2010.
6. D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *S&P*, 2010.
7. C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *POPL*, 2011.
8. C. Dimoulas, M. S. New, R. B. Findler, and M. Felleisen. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *ICFP*, 2016.
9. T. Disney and C. Flanagan. Gradual information flow typing. In *STOP*, 2011.
10. L. Fennell and P. Thiemann. Gradual security typing with references. In *CSF*, 2013.
11. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.
12. M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL*, 2010.
13. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *CCS*, 2012.
14. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 2014.
15. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
16. D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 2015.
17. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.
18. D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *CSF*, 2012.
19. D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld. A Principled Approach to Tracking Information Flow in the Presence of Libraries - full version. `http://www.cse.chalmers.se/research/group/security/libraries/`.
20. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *S&P*, 2011.
21. D. King, T. Jaeger, S. Jha, and S. A. Seshia. Effective blame for information-flow violations. In *FSE*, 2008.
22. B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *PLDI*, 1988.
23. Mozilla Developer Network. Proxy. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`. accessed: Oct 2016.
24. Mozilla Developer Network. Web APIs. `https://developer.mozilla.org/en-US/docs/Web/API`. accessed: Oct 2016.
25. Node.js v6.9.1 Documentation. `https://nodejs.org/dist/latest-v6.x/docs/api/`. accessed: Oct 2016.
26. Node Package Manager. `https://www.npmjs.com/`. accessed: Oct 2016.
27. Oracle. Java Native Interface. `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`. accessed: Oct 2016.

28. W. Rafnsson and A. Sabelfeld. Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent. In *CSF*, 2013.
29. V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015.
30. N. Rajlich. node-ffi. `https://www.npmjs.com/package/node-ffi`. accessed: Oct 2016.
31. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
32. J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *SFP*, 2006.
33. J. G. Siek and W. Taha. Gradual Typing for Objects. In *ECOOP*, 2007.
34. Haskell wiki. Foreign Function Interface. `https://wiki.haskell.org/Foreign_Function_Interface`. accessed: Oct 2016.