Information Flow Tracking for Side-effectful Libraries

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

[1] Chalmers University of Technology
[2] Mälardalen University

Abstract. Dynamic information flow control is a promising technique for ensuring confidentiality and integrity of applications that manipulate sensitive information. While much progress has been made on increasingly powerful programming languages ranging from low-level machine languages to high-level languages for distributed systems, surprisingly little attention has been devoted to libraries and APIs. The state of the art is largely an all-or-nothing choice: either a *shallow* or *deep* library modeling approach. Seeking to break out of this restrictive choice, we formalize a general mechanism that tracks information flow for a language that includes higher-order functions, structured data types and references. A key feature of our approach is the *model heap*, a part of the memory, where security information is kept to enable the interaction between the labeled program and the unlabeled library. We provide a proof-of-concept implementation and report on experiments with a file system library. The system has been proved correct using Coq.

1 Introduction

While useful, access control is not enough: it is crucial what applications do with the data after access has been granted [25]. Information flow control tracks the propagation of data in programs, thus enforcing confidentiality and integrity policies. Due to the widespread use of highly dynamic languages, such as JavaScript, there has been a growing interest in *dynamic information flow control*. There are two basic kinds of flows to consider: *explicit* and *implicit* [5], related to the notions of *data flow* and *control flow*. Dynamic information flow is tracked at runtime by extending the data with *security labels*, which are propagated and checked against a *security policy* during execution. The detection of potential security violations cause program execution to halt.

While much progress has been made on increasingly powerful programming languages ranging from low-level machine languages to high-level languages for distributed systems, surprisingly little attention has been devoted to *libraries* and *APIs*[3]. The main challenge is when the library is not written in the language itself, and thus not compatible with the labeled semantics of the program. There are mainly two situations where this occurs: 1) when the library is part of the

[3] For elegance of expression, when we write library in this paper we refer to both libraries and APIs.

standard execution environment, and 2) when the library is brought into the language using some form of *foreign function interface* (FFI). In such cases, values passing between the program and the library must be translated. The process of translating values from one programming language to another is known as *marshaling*.

Marshaling of labeled values additionally entails that security labels must be removed from the values being passed from the program to the library, and reattached on the values returned from the library to the program. We refer to those steps as *unlabeling* and *relabeling* of the values, and the description of how it should be done as a *library model*. The main difference between standard marshaling and marshaling of labeled values is the latter removes information from the values passed to the library. To be able to correctly relabel values going from the library to the program, the labels removed during the unlabeling process must be used, since the returned value contains no security information. This means that the library models are inherently stateful — the removed labels are stored in a *model state* used when relabeling.

Library models can be split into two categories: *deep* and *shallow* models [14]. Deep models track information flow inside the library, requiring precise modeling of the execution of the library, while shallow models are limited to the security labels on the boundary of the library. Often, deep models necessitate reimplementation of parts of the library functionality within the model, making them difficult to create and maintain. Shallow models, on the other hand, are significantly more lightweight, but possibly too imprecise. In this work, we are interested in the boundary between deep and shallow models.

Current state of the art in dynamic information-flow tracking does not fit this classification entirely, in part due to ad-hoc handling of libraries. To the extent addition of new libraries is supported, the models used tend towards shallow models. This is true for, e.g., FlowFox [13], and experimental extensions of JSFlow [15]. On the other hand, JSFlow and FlowFox both use deep models to provide fine grained information-flow tracking for built in libraries. JSFlow, e.g., implements the full ECMA-262 version 5 standard using what is best considered a deep approach.

In recent work, Hedin et al. [17] initiate a framework for tracking information flow in libraries. The setting is a labeled *program* and an unlabeled *library* that share the same core semantics (*split semantics*) in order to limit the marshaling to security labels only. Their work targets a focused functional language with higher-order functions (which allows for both callbacks and promises to exist), and structured data in terms of lists. It does not, however, handle side effects, which means that many libraries cannot be modeled in a satisfactory way. As an example, it is unavoidable for a standard file system library to maintain state to keep track of open files, stream positions and buffers. The success of a function `read(path, success, fail)` is dependent both on the file path and the state of the library which must be reflected by security models for the library.

The combination of state and higher-order functions significantly complicates the library models and the model state over the ones used by Hedin et al. If

the state is first-class (i.e., it can be sent around as values, as in languages with mutable references, records or objects) the situation is further complicated. *This is the setting we are interested in handling, as it captures the essence of many of the problems found when modeling real libraries.*

To this end we introduce a model heap, allowing library values to be tied to a mutable model state, which allows for secure modeling of the interaction between first-class state and higher-order functions.

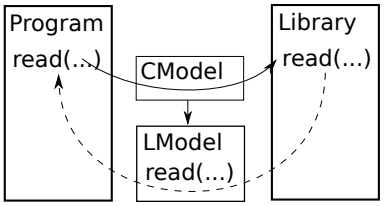

Fig. 1: Model heap illustration

Consider the file system example, depicted in Figure 1. When the program calls the library function `read`, the library function is first lifted into the program using the corresponding function model defined by the library model, LModel. The lifting (illustrated by the dotted arrow in the figure) is done by means of wrapping and results in an unlabeled function that can be called by the program. When the wrapper is called with labeled arguments, a new call model state, CModel, is created and used to hold the labels of the arguments, since the underlying library function requires unlabeled values. As can be seen in the figure, the call model state is connected to the library model state and together they define the model state that the function model of `read` interacts with. Any other values, including higher-order functions and first-class state, defined in the library share the same library model state, which guarantees that they have the same view of the library state, even in the presence of mutability.

There are two main benefits of our work over ad-hoc modeling of libraries. First, it lowers the modeling effort significantly, and, second, given that the models properly describe the library, it guarantees noninterference. Both benefits stem from expressing the models in a simplified model language that controls the marshaling process, thus sidestepping the need to reimplement it repeatedly.

Considering the dimension of shallow and deep models, our work can be seen as exploring the boundary. Shallow models are expressed solely in terms of the boundary labels, while our work gains access to intermediate labels when models for lazy marshaling, higher-order functions and first-class state are triggered. In addition, it is relatively easy to extend our system to allow models to use the runtime values allowing for *dependent models* [17]. Compared to fully deep models, our work is limited to the information passing between the program and the library at the point of passing. Thus, intermediate values and labels that do not participate in cross-boundary activity is without reach. While deep models in theory have access to more information and therefore have the potential to be more precise, it is unclear if the added precision is significant in practice, in particular in the light of the added implementation cost.

Contributions The main contributions of this paper are:

– We have created a language containing three cornerstones of library modeling: higher-order functions, first-class state, and structured values (the syntax

and semantics are presented in Section 2 and Section 3, respectively, while Section 6 discusses correctness).
- We have implemented a prototype and used it to explore the interaction between the different features of the language (examples that illustrate our mechanism are reported in Section 4).
- We have conducted a case study on a file system library, inspired by the file system library in node.js [10], showing that our language is able to handle stateful libraries (the case study is reported in Section 5).
- We have formalized the language and its correctness proof in Coq [19].

The scope of the prototype is to experimentally verify applicability of models, not to assess performance in a full-scale implementation. The prototype serves as a complement to the formal proof to create a system that is both correct and useful. The full version of the paper, along with the formalization in Coq and the proof-of-concept prototype can be found at [27].

2 Syntax

The language we present is a small functional language with split semantics and lazy marshaling. The syntax of the language is defined as follows, where n denotes numbers and x denotes identifiers.

$$\begin{aligned} e \ ::= \ & n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{fun}\ x = e \mid e_1\ e_2 \mid \\ & x_{\mathit{lib}} \mid e_1 \oplus e_2 \mid \ominus e \mid \mathit{head}\ e \mid \mathit{tail}\ e \mid e_1 : e_2 \mid [\,] \mid (e_1, e_2) \mid (\,) \mid \\ & \mathit{ref}\ e \mid !e \mid \{\ \underline{x} : \underline{e}\ \} \mid e.x \mid e_1 := e_2 \mid e_1\ ;\ e_2 \mid \mathit{upg}\ e\ \ell \end{aligned}$$

The syntax of the language is entirely standard apart from the x_{lib} construction that lifts a *library value* to a *program value*, and $\mathit{upg}\ e\ \ell$ that gives the result of the expression a given label, $\ell ::= L \mid H$. For simplicity, we identify sets with the meta variables ranging over them. Let $\underline{X}$ range over lists of X for any set X, where $[\,]$ denotes the empty list and $\cdot$ denotes the cons operator. An application in the language is a triple $(\underline{d}_p, \underline{d}_l, \underline{m})$, where the first component is the labeled *program*, the second component is the unlabeled *library* and the third component is the *library model*. Throughout the rest of this paper, we use *program* when referring to the labeled part, and *library* when referring to the unlabeled part.

The top-level definitions, d, allow for named definitions of functions and values $d ::= \mathit{fun}\ f(x) = e \mid \mathit{let}\ x = e$. The top-level model definitions, m, allow for named definitions of models and labels $m ::= \mathit{mod}\ x :: \gamma \mid \mathit{lbl}\ x :: \kappa$, where γ denotes *relabel models* and κ denotes *label terms*. The label terms, $\kappa ::= \ell \mid \alpha \mid \kappa_1 \sqcup \kappa_2$ are terms that evaluate to labels in a given model state and consist of *labels*, ℓ, *label variables*, α, and the least upper bound of two label terms. The relabel models, γ, used to relabel library values, are defined as follows

$$\gamma ::= \ \kappa \mid (\gamma_1, \gamma_2)^\kappa \mid [\gamma]^\kappa \mid (\varphi \to \gamma, \underline{\zeta})^\kappa \mid \mathrm{ref}(\varphi, \gamma)^\kappa$$

where φ denotes *unlabel models*, used to unlabel program values, and ζ denotes *effect constraints* defined below. All values are given a label by a label term, and the relabeling of structured values follows the structure of the value. To relabel a

function, we must know how to unlabel the argument, how to relabel the result, and how the function interacts with the model state. To relabel a reference we must know how to unlabel the values written and how to relabel the values read. The unlabel models, φ, are defined as follows.

$$\varphi ::= \alpha \mid \#\alpha^{\alpha} \mid (\varphi_1, \varphi_2)^{\alpha} \mid [\varphi]^{\alpha}$$

Unlabeling of values is performed by storing the label of the value in the corresponding label variable in the model state. As for relabeling, unlabeling of structured values follows the structure of the value. Unlabeling of functions and references introduces an *abstract name*, $\#\alpha$, used by library functions to tie any interaction to their model state in the effect constraints, ζ.

$$\zeta ::= \ !\#\alpha \to \varphi \mid \kappa \vdash \#\alpha \leftarrow \gamma \mid \kappa \vdash \#\alpha\ \gamma \to \varphi \mid \kappa \vdash \alpha \leftarrow \kappa$$

In the order of definition: a library function that reads a labeled reference defines how to unlabel the read value, a library function that writes to a labeled reference defines the security context in which the write occurs and how to relabel the value to be written, a library function that calls a labeled function defines the security context in which the call occurs, how to relabel the parameter and how to unlabel the result, and finally, a library function that modifies the library state defines the security context of the update and how the security model changes.

3 Semantics

We define the semantics step-wise in three parts. The first part defines the labeled values, and the execution environment. The second part defines the evaluation relation and how the function representations of the values are created and used in the semantics. Finally, the third part defines how values are marshaled between the program and the library. For space reasons, parts of the semantic definitions have been left out. We refer the reader to the full version of this paper [27] for the missing definitions.

3.1 Values

In order to differentiate between the labeled semantics and the unlabeled semantics, we use $\hat{X}$ to denote an entity in the labeled semantics corresponding to the entity X in the unlabeled semantics. We only give the labeled values. The unlabeled values are defined analogously. The values in the language, $\hat{v}$, are integers n, tuples, higher-order functions $\hat{F}$, lists $(\hat{H}, \hat{T})$, references $(\hat{R}, \hat{W})$, and records $\hat{O}$, where higher-order functions, lists, references and records are represented as (pairs of) functions in order to simplify the marshaling.

$$\hat{v}\ ::= n^{\ell} \mid (\hat{v}_1, \hat{v}_2)^{\ell} \mid ()^{\ell} \mid \hat{F}^{\ell} \mid (\hat{H}, \hat{T})^{\ell} \mid [\ ]^{\ell} \mid (\hat{R}, \hat{W})^{\ell} \mid \hat{O}^{\ell}$$

The labels, ℓ, form a two-point upper semi-lattice $L \sqsubseteq H$, where L denotes *low* (public) and H denotes *high* (private). 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 $\hat{v} = v^{\ell_1}$.

The execution environment is a triple $(\varsigma, \Gamma, \Sigma)$ of the security context, ς, the stack, and the heap. The security context ς ranges over labels ℓ. The stack

Γ is a triple of stacks $(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}})$, containing pointers to the labeled frames, the unlabeled frames and the model frames, respectively. The heap Σ is a triple of heaps, $(\hat{\sigma}, \sigma, \ddot{\sigma})$, consisting of the labeled heap, the unlabeled heap and the model heap. The labeled and unlabeled heaps can contain values (for implementing references), and frames, whereas the model heap only contains frames. The labeled and unlabeled frames, $\hat{\omega}$ and ω, are maps from identifiers to values, and the model frames, $\ddot{\omega}$ are maps from identifiers to *model items*. Each frame represents a scope, and together with the corresponding stacks they form scope chains. The model items, $\ddot{\iota} ::= \ell \mid \gamma \mid \zeta$, consists of labels, relabel models and effect constraints.

3.2 Evaluation relations

The evaluation relation for program execution is of the form $\varsigma, \Gamma \models (\Sigma_1, e) \rightarrow (\Sigma_2, \hat{v})$, read "expression e evaluates in the environment consisting of the security context, ς, the stack, Γ, and the heap, Σ_1, resulting in the updated heap Σ_2 and value $\hat{v}$". Similarly, library execution is of the form $\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, v)$, where the unlabeled semantics is parameterized over the security context to model that the context is global and always available to the marshaling functions[4].

Figure 2 contains a selection of the semantic rules of the program semantics related to the marshaling of values.

The rules of the core language are standard. Whenever an integer is created (`int`), it is always originally labeled L. Variables are retrieved from the labeled heap using lookupL in `var`. If-statements (`if-true` and `if-false`) evaluate the conditional expression and based on the result select which branch to take. The branch taken is evaluated in a security context of $\varsigma \sqcup \ell$ and the returned value is raised to ℓ, where ℓ is the label of the result of the conditional expression.

Function closures are represented as functions, $\hat{F} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \rightarrow (\Sigma_2, \hat{v})$, created by lclos (`fun`) in the following way.

$$\begin{aligned} &\text{lclos}(\underline{\hat{\rho}'}, x, e) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}_1, \sigma_1, \ddot{\sigma}_1), \hat{v}_1) \,.\, (\Sigma, \hat{v}_2) \\ &\quad \text{where } \hat{\sigma}_2 = \hat{\sigma}_1[\hat{\rho} \mapsto \{x \mapsto \hat{v}_1^{\varsigma}\}], \hat{\rho} \text{ fresh} \\ &\quad\ \text{and } \varsigma, (\hat{\rho} \cdot \underline{\hat{\rho}'}, \underline{\rho}, \underline{\ddot{\rho}}) \models ((\hat{\sigma}_2, \sigma_1, \ddot{\sigma}_1), e) \rightarrow (\Sigma, \hat{v}_2) \end{aligned}$$

The function closure will, when interacted with, create a new pointer to a labeled frame containing the mapping of the parameter name x and the actual value $\hat{v}_1$, which is raised to the current security context. The function expression e is then evaluated, using the newly created pointer along with the updated heap. When applying a function closure (`app`), the body of the function is executed in the program semantics, under the elevated context consisting of the current security context raised to the label of the function closure. Creation and application of library closures, $F : (\varsigma, \Gamma, \Sigma_1, v) \rightarrow (\Sigma_2, v)$, is analogous.

Safe implementation of marshaling of references requires the ability to trap and modify reads and writes in order to marshal the values passed by the interaction. For this reason, references are represented as pairs of functions, one

[4] In an operational semantics global non-constant values must be passed around during execution, similar to in a pure functional language.

$$\texttt{int}\ \frac{}{\varsigma, \Gamma \models (\Sigma, n) \to (\Sigma, n^L)} \qquad \texttt{var}\ \frac{\text{lookupL}(\Gamma, \Sigma, x) = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x) \to (\Sigma, \hat{v})}$$

$$\texttt{if-true}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^\ell) \qquad v_1 \neq 0 \qquad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \to (\Sigma_3, \hat{v}_2^\ell)}$$

$$\texttt{if-false}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^\ell) \qquad v_1 = 0 \qquad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_3) \to (\Sigma_3, \hat{v}_3)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \to (\Sigma_3, \hat{v}_3^\ell)}$$

$$\texttt{fun}\ \frac{}{\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}) \models (\Sigma, \mathit{fun}\ x = e) \to (\Sigma, \text{lclos}(\underline{\hat{\rho}}, x, e)^L)}$$

$$\texttt{app}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{F}^\ell) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_1) \qquad \hat{F}(\varsigma \sqcup \ell, \Gamma, \Sigma_3, \hat{v}_1) = (\Sigma_4, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1\ e_2) \to (\Sigma_4, \hat{v}_2^\ell)}$$

$$\texttt{ref}\ \frac{\varsigma, \Gamma \models (\Sigma, e) \to ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}) \qquad \hat{\rho}\ \text{fresh}}{\varsigma, \Gamma \models (\Sigma, \mathit{ref}\ e) \to ((\hat{\sigma}[\hat{\rho} \mapsto \hat{v}], \sigma, \ddot{\sigma}), (\text{lread}(\hat{\rho}), \text{lwrite}(\hat{\rho}))^L)}$$

$$\texttt{deref}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \qquad \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, !e) \to (\Sigma_3, \hat{v}^\ell)} \qquad \texttt{assign}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}) \qquad \hat{W}(\varsigma \sqcup \ell, \Gamma, \Sigma_3 \hat{v}) = \Sigma_4}{\varsigma, \Gamma \models (\Sigma_1, e_1 := e_2) \to (\Sigma_4, \hat{v})}$$

$$\texttt{lib}\ \frac{\text{lookupU}(\Gamma, \Sigma, x) = v \qquad \text{lookupM}(\Gamma, \Sigma, x) = \gamma \qquad v \uparrow_{\Gamma, \Sigma} \gamma = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x_{lib}) \to (\Sigma, \hat{v})}$$

Fig. 2: Selected labeled semantics

function for reading the reference, $\hat{R} : (\varsigma, \Gamma, \Sigma_1) \to (\Sigma_2, \hat{v})$, and one function for updating the reference, $\hat{W} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \to \Sigma_2$. This allows us to marshal references by wrapping the read and the write functions in functions that perform the marshaling of the values at the time of interaction, similar to lazy marshaling of lists [17]. Most languages do not support the creation of functions that are triggered on interaction with values such as references or objects, which means they cannot support marshaling of first-class mutable state. A notable exception to this is JavaScript that allows methods to be tied to different aspects of object interaction via the use of Proxy objects [22].

Creation of references given a fresh pointer into the labeled heap is defined by lread and lwrite as follows.

$$\begin{aligned}
&\text{lread}(\hat{\rho}) = \lambda(\varsigma, \Gamma, (\hat{\sigma}, \sigma, \ddot{\sigma}))\ .\ ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}),\ \text{where}\ \hat{v} = \hat{\sigma}[\hat{\rho}] \\
&\text{lwrite}(\hat{\rho}) = \lambda(\varsigma, \Gamma, (\hat{\sigma}_1, \sigma_1, \ddot{\sigma}_1), \hat{v})\ .\ (\hat{\sigma}_2, \sigma_1, \ddot{\sigma}_1) \\
&\quad \text{where}\ v^\ell = \hat{\sigma}_1[\hat{\rho}], \varsigma \sqsubseteq \ell, \hat{\sigma}_2 = \hat{\sigma}_1[\hat{\rho} \mapsto \hat{v}^\varsigma]
\end{aligned}$$

References (`ref`) are created by selecting a fresh heap location made to point to the value of the reference. The heap location is then used to create a pair of access

functions. The created reference follows the same intuition as for all created values. All values are labeled L upon creation, which is why the pair of access functions are labeled L in `ref`. Note that the value that the reference is referring to may be labeled differently, due to the distinction between reference as a value and the value the reference is referring to. Dereferencing (`deref`) uses the read function of the reference to get the value to be read, while assignment (`assign`) uses the write function. Creation and use of library references, $R : (\varsigma, \Gamma, \Sigma_1) \to (\Sigma_2, v)$ and $W : (\varsigma, \Gamma, \Sigma_1, v) \to \Sigma_2$ is analogous.

It is worthwhile to point out the *no-sensitive upgrade* (NSU) check in lwrite, which demands that the context, which the label of the reference is a part of, is lower or equal to the label of the referenced value, $\varsigma \sqsubseteq \ell$. Allowing labels of values to change freely leads to an unsound system, due to the possibility of implicit flows into the labels themselves [1,28].

Disregarding the encoding of functions and references into functions, up to this point, the labeled and unlabeled semantics are equivalent to their standard formulations. The essence of this paper is in the marshaling of values between the program and the library, performed by the unlabeling and relabeling functions, defined in the following section.

3.3 Marshaling

All interaction between the program and the library is initiated by lifting named library values into the program. This is done (`lib`) by looking up the value, and the corresponding relabel model used to relabel the value. Interaction with the relabeled value may cause further marshaling. Unlabeling of a value is done w.r.t. an unlabel model, φ, which defines how to store the removed label(s) in the model state. Relabeling of a value is done w.r.t. a relabel model, γ, which defines how to compute the label in terms of the model state. Formally, unlabeling is a function of the form $\hat{v} \downarrow_{\varsigma,\Gamma,\Sigma_1} \varphi = (\Sigma_2, v)$ taking a labeled value $\hat{v}$, an environment, $\varsigma, \Gamma, \Sigma_1$ and an unlabel model φ and returning an updated heap, Σ_2, and an unlabeled value v. Similarly, relabeling is a function of the form $v \uparrow_{\Gamma,\Sigma} \gamma = \hat{v}$, taking an unlabeled value, v, an environment, Γ, Σ, and a relabel model, γ, and returning a labeled value $\hat{v}$. The only modified part of the heap for both unlabeling and relabeling is the model heap.

There are six types of values: integers, tuples, lists, records, higher-order functions and references. In the rest of this section we describe how to evaluate label terms (used when relabeling) and how to marshal higher-order functions and references. We refer the reader to the full version of this paper [27] for the treatment of the other constructs.

Label terms Evaluation of label terms is done w.r.t. a model state, where lookupM is used to traverse the model scope chain to find the first label corresponding to a given label variable.

$$[\![\alpha]\!]_{\Gamma,\Sigma} = \begin{cases} \ell, & \text{if lookupM}(\Gamma, \Sigma, \alpha) = \ell \\ L, & \text{otherwise} \end{cases}$$
$$[\![\ell]\!]_{\Gamma,\Sigma} = \ell$$
$$[\![\kappa_1 \sqcup \kappa_2]\!]_{\Gamma,\Sigma} = [\![\kappa_1]\!]_{\Gamma,\Sigma} \sqcup [\![\kappa_2]\!]_{\Gamma,\Sigma}$$

Higher-Order Functions Marshaling of higher-order functions involves both marshaling the functions as values as well as ensuring the parameter and return value are properly marshaled.

Unlabeling Unlabeling a program closure removes and stores the label and returns a library closure created by wrapping the program closure. The library closure is tied to the abstract name, π, used by the wrapper to relabel the parameters before the call and unlabel the result after the call.

$$\hat{F}^{\ell} \downarrow_{\varsigma,\Gamma,\Sigma} \#\pi^{\alpha} = (\text{updateM}(\varsigma, \Gamma, \Sigma, \alpha, \ell), \text{u-lclos}(\hat{F}, \ell, \#\pi))$$

The translation of a program closure, $\hat{F}$, into an library closure is performed by u-lclos, that takes the program closure, the label of the program closure and the abstract name. When the library closure returned by u-lclos is applied the following occurs. First, the function call model bound to the abstract name is fetched using lookupM. The function call model contains a label term representing the security context of the application, how to relabel the parameter and how to unlabel the return value. Second, the relabel model, γ, is used to relabel the parameter, v_1. Third, the program closure is called in the security context of the call raised to the label of the closure and the evaluation of the context label term, κ. The result of the call is a labeled value, $\hat{v}_2$. Finally, $\hat{v}_2$ is unlabeled which gives the result, v_2, of the application of the unlabeled closure. Notice that all relabeling and unlabeling is done with respect to the model state of the caller.

$$\begin{array}{c}\text{u-lclos}(\hat{F}, \ell_1, \#\pi) = \lambda(\varsigma, \Gamma, \Sigma_1, v_1)\ .\ (\Sigma_3, v_2) \\ \text{where } \kappa \vdash \gamma \rightarrow \varphi = \text{lookupM}(\Gamma, \Sigma_1, \pi) \\ \ell_2 = [\![\kappa]\!]_{\Gamma,\Sigma_1} \\ \hat{v}_1 = v_1 \uparrow_{\Gamma,\Sigma_1} \gamma \\ (\Sigma_2, \hat{v}_2) = \hat{F}(\varsigma \sqcup \ell_1 \sqcup \ell_2, \Gamma, \Sigma_1, \hat{v}_1) \\ (\Sigma_3, v_2) = \hat{v}_2 \downarrow_{\varsigma \sqcup \ell_1 \sqcup \ell_2, \Gamma, \Sigma_2} \varphi\end{array}$$

Relabeling Relabeling a library closure is done by labeling the program closure created by wrapping the library closure. The wrapper unlabels the arguments before the call and relabels the result of the call.

$$F \uparrow_{\Sigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}})} (\varphi \rightarrow \gamma, \underline{\zeta})^{\kappa} = \text{l-uclos}(F, \underline{\ddot{\rho}}, (\varphi \rightarrow \gamma, \underline{\zeta}))^{[\![\kappa]\!]_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}),\Sigma}}$$

The process is controlled by the function relabel model, $(\varphi \rightarrow \gamma, \underline{\zeta})^{\kappa}$, where the evaluation of κ gives the label of the wrapper closure.

The translation of the library closure, F, into a program closure is performed by l-uclos, which takes the library closure, the current model frame stack, the unlabel model for the parameters, φ, the relabel model for the return value, γ, and the effect constraints, $\underline{\zeta}$. When called the program closure produces a fresh frame pointer, pointing to a new model frame in the model

$$\begin{array}{c}\text{l-uclos}(F, \underline{\ddot{\rho}}_2, (\varphi \rightarrow \gamma, \underline{\zeta})) = \\ \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), (\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}_1).\ (\Sigma_4, \hat{v}_2) \\ \text{where } \Sigma_1 = (\hat{\sigma}, \sigma, \ddot{\sigma}[\ddot{\rho} \mapsto \varnothing]),\ \ddot{\rho} \text{ fresh} \\ (\Sigma_2, v_1) = \hat{v}_1 \downarrow_{\varsigma,(\underline{\hat{\rho}},\underline{\rho},\ddot{\rho}\cdot\underline{\ddot{\rho}}_2),\Sigma_1} \varphi \\ \Sigma_3 = \{|\underline{\zeta}|\}_{\varsigma,(\underline{\hat{\rho}},\underline{\rho},\ddot{\rho}\cdot\underline{\ddot{\rho}}_2),\Sigma_2} \\ (\Sigma_4, v_2) = F(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}_1), \Sigma_3, v_1) \\ \hat{v}_2 = v_2 \uparrow_{(\underline{\hat{\rho}},\underline{\rho},\ddot{\rho}\cdot\underline{\ddot{\rho}}_2),\Sigma_4} \gamma\end{array}$$

heap. The parameter to the library function is unlabeled based on the unlabel model, φ, and the effect constraints, $\underline{\zeta}$, are evaluated to update the model state accordingly. After that, the library function is called with the unlabeled parameter in the security context, ς, of the call. The result of the function call is relabeled with the relabel model, γ, and returned to the program. Note that all labeling and unlabeling is done w.r.t. the model frame stack of the unlabeled closure. Also note that the order is important; if the unlabeling of the parameter occurs after evaluating the effect constraints, the label of the parameter cannot be used when updating the model state with the side effects.

Effect constraints Effect constraints define how a library function interacts with unlabeled program functions and references and how the library function changes the model state. Model state changes are effectuated on call to the library function whereas effect constraints that define interaction with unlabeled program functions and references are stored in the model state. When a library function or reference is interacted with, the abstract name will tie the interaction to the corresponding effect constraint in the model state of the interaction. The meaning of the effect constraints is defined as follows

$$\begin{aligned}
\{|!\#\alpha \to \varphi|\}_{\varsigma,\Gamma,\Sigma} &= \text{defineM}(\Gamma, \Sigma, \alpha, \varphi) \\
\{|\kappa \vdash \#\alpha \leftarrow \gamma|\}_{\varsigma,\Gamma,\Sigma} &= \text{defineM}(\Gamma, \Sigma, \alpha, \kappa \vdash \gamma) \\
\{|\kappa \vdash \#\alpha\ \gamma \to \varphi|\}_{\varsigma,\Gamma,\Sigma} &= \text{defineM}(\Gamma, \Sigma, \alpha, \kappa \vdash \gamma \to \varphi) \\
\{|\kappa_1 \vdash \alpha \leftarrow \kappa_2|\}_{\varsigma,\Gamma,\Sigma} &= \text{updateM}(\varsigma \sqcup [\![\kappa_1]\!]_{\Gamma,\Sigma}, \Gamma, \Sigma, \alpha, [\![\kappa_2]\!]_{\Sigma,\Gamma}) \\
&\qquad \text{when } \varsigma \sqcup [\![\kappa_1]\!]_{\Gamma,\Sigma} \sqsubseteq \text{lookupM}(\Gamma, \Sigma, \alpha)
\end{aligned}$$

where defineM binds the name α to its corresponding model value in the top model frame, if α is not defined in that model frame, updateM updates the label pointed to by α in the scope chain, or inserts it if it is not present, and lookupM returns the model value that is the first to match the name α in the scope chain.

References Marshaling of references shares some similarities with marshaling of higher-order functions. Calling a function passes the argument and the return value in opposite directions, similar to reading and writing to a reference.

Unlabeling Unlabeling a program reference removes and stores the label, and the read and write functions are wrapped to create library counterparts.

$$\begin{aligned}
&(\hat{R}, \hat{W})^{\ell} \downarrow_{\varsigma,\Gamma,\Sigma} \#\pi^{\alpha} = \\
&\quad (\text{updateM}(\varsigma, \Gamma, \Sigma, \alpha, \ell), (\text{u-lread}(\hat{R}, \ell, \#\pi), \text{u-lwrite}(\hat{W}, \ell, \#\pi)))
\end{aligned}$$

The read and the write functions are translated independently w.r.t. the abstract name $\#\pi$.

The program read function, $\hat{R}$ is translated by u-lread, which takes the read function, the label of the reference and the abstract name. When the resulting library read function is interacted with, the program read function is used to get the labeled value of the reference. This value must be unlabeled before being returned, which is

$$\begin{aligned}
&\text{u-lread}(\hat{R}, \ell, \#\pi) = \lambda(\varsigma, \Gamma, \Sigma_1)\ .\ (\Sigma_3, v) \\
&\quad \text{where } \varphi = \text{lookupM}(\Gamma, \Sigma_1, \pi) \\
&\qquad\quad (\Sigma_2, \hat{v}) = \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_1) \\
&\qquad\quad (\Sigma_3, v) = \hat{v} \downarrow_{\varsigma \sqcup \ell, \Gamma, \Sigma_2} \varphi
\end{aligned}$$

done by looking up a program reference read model, φ, in the model state of the interaction. It is the model of the caller, i.e., a library function model that provides the read model for the references it reads.

The program write function, $\hat{W}$ is translated by u-lwrite, which takes the write function, the label of the reference and the abstract name. When the resulting library write function is used, the associated program reference write model, $\kappa \vdash \gamma$, is fetched in the current model state. This model defines both how to relabel the written unlabeled value, and the context in which the write occurs. Then the unlabeled value, v is relabeled before being written using the labeled write function in a context consisting of the current security context of the call raised to the reference label and the evaluation of the context label term, κ.

$$\begin{aligned}\text{u-lwrite}(\hat{W}, \ell, \#\pi) = \lambda(\varsigma, \Gamma, \Sigma_1, v)\ .\ \Sigma_2 \\ \text{where } \kappa \vdash \gamma = \text{lookupM}(\Gamma, \Sigma_1, \pi) \\ \hat{v} = v \uparrow_{\Gamma, \Sigma_1} \gamma \\ \Sigma_2 = \hat{W}(\varsigma \sqcup \ell \sqcup [\![\kappa]\!]_{\Gamma, \Sigma_1}, \Gamma, \Sigma, \hat{v})\end{aligned}$$

Relabeling Relabeling a library reference is done by translating the read and write functions into program counterparts and relabeling the result.

$$(R, W) \uparrow_{\Sigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}})} ref(\varphi, \gamma)^{\kappa} = (\text{l-uread}(R, \underline{\ddot{\rho}}, \gamma), \text{l-uwrite}(W, \underline{\ddot{\rho}}, \gamma, \varphi))^{[\![\kappa]\!]_{(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma}}$$

The read and the write functions are translated independently w.r.t. the relabel model, $ref(\varphi, \gamma)^{\kappa}$.

The library read function, R, is translated by l-uread, which takes the read function, the current model frame stack, and the relabel model, γ. When the resulting program read function is interacted with, the unlabeled read function is used to fetch the unlabeled value of the reference. The result is relabeled using the relabel model in the model state of the reference and the result is returned.

$$\begin{aligned}\text{l-uread}(R, \underline{\ddot{\rho}}_2, \gamma) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1)\ .\ (\Sigma_2, \hat{v}) \\ \text{where } (\Sigma_2, v) = R(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1) \\ \hat{v} = v \uparrow_{(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_2), \Sigma_2} \gamma\end{aligned}$$

The library write function W is translated by l-uwrite, which takes the write function, the current model frame stack, the relabel model, γ, and the unlabel model, φ. The reason l-uwrite takes the relabel model in addition to the unlabel model is that it is used to calculate the label against which the NSU check is made. The label of the stored value is represented by the label term of the relabel model, extracted by the lblterm function, defined in the obvious way by pattern matching. If the write is allowed, the labeled value to be written to the library reference is raised to the context ς, before being unlabeled using the unlabel model, φ. Finally, the unlabeled value is written to the library reference, using the unlabeled write function.

$$\begin{aligned}\text{l-uwrite}(W, \underline{\ddot{\rho}}_2, \gamma, \varphi) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1, \hat{v})\ .\ \Sigma_3 \\ \text{where } \ell = [\![\text{lblterm}(\gamma)]\!]_{(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_2), \Sigma_1},\ \varsigma \sqsubseteq \ell \\ (\Sigma_2, v) = \hat{v}^{\varsigma} \downarrow_{\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_2), \Sigma_1} \varphi \\ \Sigma_3 = W(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_2, v)\end{aligned}$$

Interaction with the model heap To see how higher-order functions and references interact with the model heap, consider the code snippet below to the right. The program calls the library function `f`, which takes a parameter, and creates a reference `r` initially set to the value of the parameter.

`f` returns a pair, where the first element is a function that, given any argument, will dereference the reference and the second element is the actual reference. This pair is stored as `(g, r)`. Thereafter, `r` is assigned the value 15^H, before `g` is called with the parameter 20^L.

```
let (g, r) = lib f 10
in r := upg 15 H;
   g 20
%%
lbl l :: L
mod r :: ref (l, l)
mod f :: x -> (y -> l, r)
fun f x = let r = ref x
          in (\y. !r, r)
```

The following occurs w.r.t. relabeling and unlabeling in the program, where the initial setting can be seen in Figure 3.

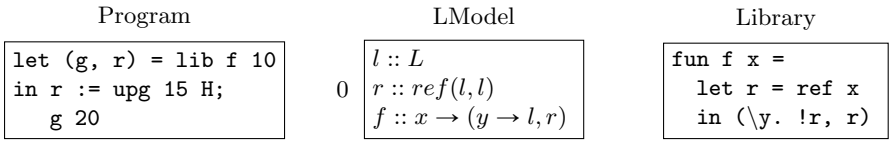

Fig. 3: Initial structure

When `f` is lifted to the program, l-uclos is used to relabel the library closure, which will copy the model frame stack to the wrapped `f` and store the function model $x \to (y \to l, r)$. In the example, the resulting program closure is applied to 10^L, which causes a new model frame to be allocated on the model heap, into which the argument is unlabeled, causing L to be stored in the new model frame as the label for `x`, and the pointer to the new model frame is stored in the model frame stack. After this, the actual unlabeled function is called, which results in the returned pair being relabeled. The relabeling of the pair results in l-uclos being used to relabel `\y. !r` with the model $y \to l$, and l-uread and l-uwrite being used to relabel `r` with the reference model r. The key here is that the relabeling occurs in the same model state, which means that the produced program function and reference will be bound to the same model frame stack. This causes writes to the reference to modify the model frame shared with the function, ensuring that they have the same view of the model of the reference. The entire process is highlighted in Figure 4.

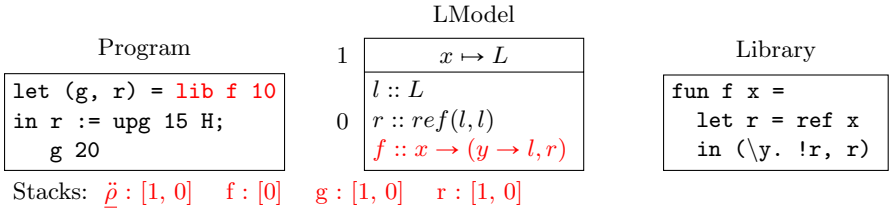

Fig. 4: Calling relabeled closure

When the program writes to the reference (`r := upg 15 H`), the closure from l-uwrite is triggered, causing l in the shared model frame to be updated to H, which can be seen in Figure 5. Note that the pointer to the model frame created from the call to the wrapped `f` is removed from the model frame stack. This ensures any subsequent calls to the wrapped `f`, as well as any created wrappers will not be able to use that model frame, as it belongs only to the first call to

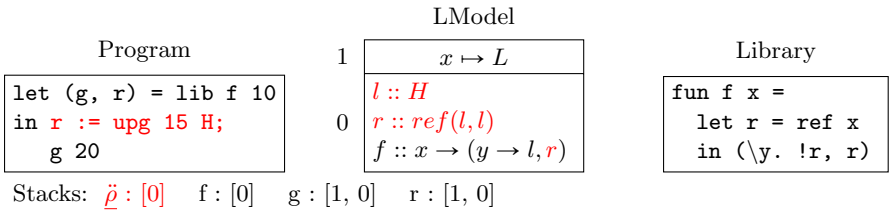

Fig. 5: Writing to `r`

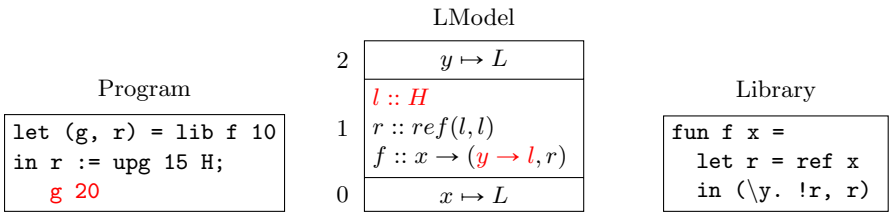

Fig. 6: Calling `g`

the wrapped `f` and the created wrappers *within* the call. When the function `g` is called, it will trigger its l-uclos wrapper and, as can be seen in Figure 6, the model $y \to l$ is used in the l-uclos wrapper for `g`, with l being used to relabel the result. Since l was modified by the writing to the reference (Figure 5), the shared view of the library model state, will make the function `g` return a secret value.

4 Examples

In the following section we provide some examples to highlight how the language would interact with common programming techniques. The language used in this section is an extended version of the language of the paper. The major differences are the addition of records, functions with multiple arguments, a limited form of pattern matching, and optional unlabeling. The extensions are all present as experimental features in the implementation. In all examples, the code above `%%` is the program and the code below is the library.

Writebacks Returning two or more results from a function can be done in two ways: 1) tupling the result, or 2) by using writebacks. When using writebacks for, e.g., reading a file, the read function is provided a pointer to a place in the memory where the contents of the file should be stored instead of returning a pointer to the data.

In our language, writebacks can be modeled by passing program references to the library as shown to the right. In the example, the program variable `buf` is a program reference. The reference is passed to the library function `action` that writes the result to the buffer. When interacting

```
let buf = ref 0
fun main () = (lib action) buf;
              !buf
%%
let data = 42
mod action :: #b -> L {| #b <- H |}
fun action b = b := data; ()
```

with a program reference, the reference is given an abstract name (`b` for buffer in this case) that the function interacting with the buffer uses to relabel the interaction.

In case the function used the writeback under secret control, represented by the model `mod action :: #b -> L {| H |- #b <- H |}`, the example would fail due to NSU. The reason being the value the reference `buf` is pointing to is public, and is not allowed to change label under secret control. Modifying the declaration of `buf` to be `let buf = ref (upg 0 H)` solves this, as the reference will point to a secret value.

Library state Libraries often keep state, e.g., error codes, computation results or options set by the program. Typical examples are the predefined object properties `$1,..,$9` from JavaScript RegExp [23].

```
fun main () =
  (lib action) (upg 42 H);
  print !(lib errno)
%%
lbl l :: L
mod errno :: ref (l, l)
let errno = ref 0

mod action :: a -> L {| l <- a |}
fun action x = if x == 1
               then errno := 1
               else ();
               ()
```

The example to the right shows how state can be used to store error information. In the example, the function `action` may fail depending on the value of the parameter. The reason it failed is stored into the library reference `errno`, which is modeled by a security label used to relabel program reads and writes of the reference. Since the update of `errno` is conditional, it means that the value of `errno` is dependent of the argument of the `action` function. To model this, the argument label is stored in the model variable `a`, which is used to update the security label of `errno`. Note that the update of the security label is independent on whether the operation fails or not. This is needed to ensure that the label of `errno` is independent of secrets. The label of `errno` indicates that the error code is public. Consider the case where an action sets the error code under secret control, represented by the following model `mod action :: a -> L {| H |- l <- a |}`. If such an action was used our system would halt execution, since the update of the error code would trigger NSU.

The one-place buffer In the previous example, the library state is exposed to the program, which can freely read and write to `errno`. Frequently it is good practice to hide the internal state of the library and only allow the program to access it indirectly via the functions of the library. We exemplify this by implementing a simple one-place buffer, seen to the right. While simple, the example captures the essence of, e.g., buffered file access.

```
fun main () = (lib set) (upg 42 H);
              (lib getAsync) print;
              (lib get) ()
%%
lbl l :: L
let buf = ref 0

mod set :: a -> a {| l <- a |}
fun set x = buf := x

mod get :: _ -> l
fun get () = !buf

mod getAsync :: #cb -> L {| #cb l -> _ |}
fun getAsync cb = cb !buf; ()
```

Since there is no model for `buf`, it is not accessible from the program. Instead, the state of the library is modeled using the label `l`. This label is used by the operations that give the program access to the buffer contents. When setting the

value of the buffer via `set`, the label of the value is used to update the label of the library state. When reading, either via the synchronous function `get` or via the asynchronous function `getAsync`, `l` is used to relabel the dereferenced value from `buf`. In the synchronous case by relabeling the dereferenced value directly, and in the asynchronous case by relabeling the parameter to the callback. Note the use of the wildcard `_` to indicate values that are not important for the model.

Stored callbacks Stored callbacks are callbacks that are saved in the internal state of the library and used, e.g., to signal the occurrence of some event. A typical example of stored callbacks is the event handlers present in many languages.

Consider the program to the right that registers an event handler by storing the event handler (`print` in this case) in the `event` reference of the library. The relabel model of the `event` will unlabel the function and give it the abstract name `event`.

```
fun main () = lib event := print;
              (lib fire) (upg 42 H)
%%
lbl l :: L
mod event :: ref (l, #event^l)
let event = ref 0

mod fire :: a -> L {| #event a -> _ |}
fun fire x = !event x; ()
```

The event is triggered by calling the `fire` function, which takes the event data and passes it to the stored event handler. In the example, the `fire` function may be called from the program. In a practical setting, events may be triggered by interacting with the library (e.g., by adding values to a data structure) or from the library itself to indicate that certain events, such as mouse movement or clicks, have occurred.

In the example, it is not possible to fetch the event handler from the library and call it. In order to allow for this, we have to change the relabel model for the library reference to relabel read interactions as functions, changing the `event` model to be `mod event :: ref (a -> b {| #event a -> b |}^l, #event^l)`. To understand the new relabel model we must recognize that unlabeled program functions that are passed back need to be relabeled as any other library function. In this case, the library function that should be relabeled calls the unlabeled program function, and needs a corresponding call model. The result is a function that unlabels its argument into the label variable `a`, which is used to relabel the argument before calling the program function. The result of calling the program function is unlabeled into the label variable `b`, which in turn is used to relabel the result of the relabeled function.

5 Case study

For case study, we model an API inspired by the fs API of node.js [10]. In the interest of exposition we model the file system state as a single label as shown to the right. The extension of the model to nested records is simple but space demanding.

```
lbl l :: L
mod state :: ref (l, l)
let state = ref 1
```

Examples of functions in the API are the `rmdir` function and its synchronous sibling `rmdirSync`. Both will, given a path, remove the folder pointed to by the path. In addition, `rmdir` also takes a callback that is called with an error if the removal of the folder pointed to by the path fails.

```
mod rmdirSync :: a -> l + a {| l <- a |}
mod rmdir :: (a, #cb) -> L {| l <- a, #cb (l + a) -> b |}
```

We use the name `a` to represent the path and the abstract name `cb` to represent the callback. From a modeling standpoint, we need to ensure that the level of the path is propagated to the state, since removing the folder influences the file system state. We can see this in the effect constraint `l <- a`, where the label of the path is propagated to the label of the state. The success of the operation is depending on the library state and the security label of the path, `l + a`. Where `rmdirSync` returns the result, `rmdir` communicates the result to the callback as an argument, `#cb (l + a)`. The immediate return value of the latter is `undefined`, regardless of the outcome of the operation and hence labeled `L`.

A more complex function in the API is `createWriteStream` that returns a record. Calling `createWriteStream` with a path and an optional argument that defines options (e.g. the encoding) returns a WriteStream.

```
mod createWriteStream :: (a, b)
  -> { path         : a
     , bytesWritten : a + b + l
     , open         : #op -> L {| #op (a + b + l) -> o |}
     , close        : #cl -> L {| #cl (a + b + l) -> c |}
     }
```

The WriteStream has four parts; the fields `path` and `bytesWritten`, as well as the events `open` and `close`. For the model of the returned record, the property `path` is modeled by the argument `a`, which is the label of the path. The property `bytesWritten`, which corresponds to the amount of bytes written so far, is modeled as the least upper bound of `a`, `b` and `l`, i.e., the path, the options and the current library state. The events are modeled as functions that accept (and store) callbacks — the event handler — as modeled by the properties `open` and `close`. When the stream is opened or closed, the path, the options and the current library state all influence the parameter to those callbacks.

To contrast the case study with the examples, note that Section 4 makes the assumption that the source code of the library is available (albeit not supporting the labeled semantics) whereas this section makes the assumption it is not. Both cases are common, and can be modeled in our approach. In case the source code is indeed available an interesting line of future work is to look at the possibilities of automatically deducing models, e.g., using something similar to summary functions [26].

6 Correctness

The correctness of the language is complicated by the fact that it is parameterized over a library model that defines how to marshal values between the program and the library. Since we make no assumption on the implementation language of the library or the availability of the source code we cannot reason about the correctness of the model w.r.t. the library. Instead we assume the correctness of library models in terms of three hypotheses used in the noninterference proof. The low-equivalence definition, the model hypotheses and more information on the proof can be found in the full version of the paper [27].

We prove noninterference assuming that the library model correctly models the library as the preservation of a low-equivalence relation under execution. Apart from covering a larger language, the proof improves over [17] in two important aspects: 1) it significantly weakens the model hypothesis, and 2) the proof has been formalized in Coq [19].

Theorem 1 (Noninterference of labeled execution)

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma_1') \ \wedge \ \varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, \hat{v}) \ \wedge \\ \varsigma, \Gamma' \models (\Sigma_1', e) \to (\Sigma_2', \hat{v}') \Rightarrow (\Gamma, \Sigma_2) \simeq (\Gamma', \Sigma_2') \ \wedge \ \hat{v} \simeq \hat{v}'$$

Proof. By mutual induction on labeled and unlabeled evaluation (via u-lclos and l-uclos). The theorem makes use of *confinement*, i.e., that evaluation under high security does not modify the public part of the environment.

7 Related work

Bielova and Rezk present a comprehensive taxonomy of information flow monitors [4]. Some monitors [16,15,14,3] and secure multi-execution [13,12,24,6,20] mechanisms have been integrated in a browser. Bichhawat et al. instrumented the WebKit JavaScript interpreter [3]. While taking advantage of the current optimizations in the interpreter, it loses the differentiation between the program and library execution. FlowFox [13], which implements *secure multi-execution (SME)* [6], modifies the SpiderMonkey engine in two ways: 1) augmenting the internal objects representing the JavaScript context with a current execution level, as well as a boolean indicating if SME is active, and 2) augmenting the internal representation of JavaScript values with a security level. Unfortunately, API calls are only treated as I/O actions. JSFlow [16] is an information-flow aware JavaScript interpreter, augmented with security labels on the JavaScript values. In order to allow for libraries in JSFlow, deep hand-written models must be used, with reimplementation of the libraries as a result [15]. To allow for scaling, JSFlow attempts to automatically wrap libraries, albeit in an ad-hoc manner. While the correctness of simple examples are easy to see, the correctness and scalability when passing, e.g., functions to and from the library remain unclear. Bauer et al. [2] developed a light-weight coarse-grained run-time monitor for Chromium, using taint tracking, to help reasoning about information flow in a fully fledged browser. In this work, formal models of, e.g., cookies, history and the *document object model (DOM)* are defined, as well as event handlers, to model the browser internals and help prove noninterference. Heule et al. [18] provided a theoretical foundation for a language-based approach for coarse-grained dynamic information flow control, that can be applied to any programming language where external effects can be controlled. A first step for handling libraries in environments where dynamic information flow control is not possible was taken by Hedin et al. [17], falling short by not supporting references, and thereby not allowing for first-class mutable state in combination with higher-order functions.

Findler and Feleisen's higher-order contracts [9] address the problem of checking contracts at the boundary between statically type-checked and dynamically

type-checked code. The problem relates to the problem of interfacing with libraries where it is impossible to check dynamic information flow control. In particular, when considering function values crossing the boundary, the compliance of such function values with their respective contracts is undecidable. Findler and Feleisen proposed to wrap the function and check the contract at the point where the function is called. This is comparable to how we handle structured data, including references and function values. A question for future work is if we can remove our abstract identifiers for function values and references, and instead inject the unlabeling/relabeling functionality using proxies, similar to how it is done in higher-order contract checking [8]. If a contract is violated, the proper assignment of blame must be given [7,11]. In static information flow checking, the assignment of blame has been investigated by King et al. for information flow violations [21]. Although our work can be seen as an application of dynamic higher-order contract checking for information flow contracts, we do not consider assigning blame. Indeed, runtime detection of a library which does not obey the specified contract (i.e. the given model) is not possible in this work.

8 Conclusion

Based on a central idea of a model heap, we have developed a foundation for information flow tracking in the presence of libraries with side effects in a language with higher-order functions, first-class state and lazy-marshaling — three cornerstones of practical libraries. We have implemented a prototype to verify the examples and performed a larger case study that shows that the language is able to model key parts of a real file system library. In addition, we have formalized the language and its correctness proof in Coq.

Future work includes support for model abstraction and application, and dependent models. Thanks to the three cornerstones, we believe modeling JavaScript objects does not require development of new theory, indicating that it is possible to use this technique in tools like JSFlow.

Acknowledgments This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

1. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.
2. L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in chromium. In *NDSS*. The Internet Society, 2015.
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 T. Rezk. A taxonomy of information flow monitors. In *POST*, 2016.
5. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
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. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.
10. File System — Node.js v9.2.0 Documentation. `https://nodejs.org/api/fs.html`. accessed: Nov 2017.
11. M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL*, 2010.
12. W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *CCS*, 2012.
13. W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 2014.
14. D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 2015.
15. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.
16. D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *CSF*, 2012.
17. D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld. A principled approach to tracking information flow in the presence of libraries. In *POST*, 2017.
18. S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *POST*, 2015.
19. INRIA. The Coq Proof Assistant. `https://coq.inria.fr/`. accessed: Nov 2017.
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. Mozilla Developer Network. Proxy. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`. accessed: Mar 2018.
23. Mozilla Developer Network. RegExp. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp`. accessed: Mar 2018.
24. W. Rafnsson and A. Sabelfeld. Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent. In *CSF*, 2013.
25. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
26. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.
27. A. Sjösten, D. Hedin, and A. Sabelfeld. Information Flow Tracking for Side-effectful Libraries - Full version. `http://www.cse.chalmers.se/research/group/security/side-effectful-libraries/`.
28. S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.