Dynamic Security Labels and Noninterference

Lantian Zheng Andrew C. Myers
Computer Science Department
Cornell University, Ithaca, NY 14853
{zlt,andru}@cs.cornell.edu

Abstract

This paper gives a language in which information flow is securely controlled by a dependent type system, yet the security classes of data can vary dynamically. Information flow policies provide the means to express strong security requirements for data confidentiality and integrity. Recent work on security-typed programming languages has shown that information flow can be analyzed statically, ensuring that programs will respect the restrictions placed on data. However, real computing systems have security policies that vary dynamically and that cannot be determined at the time of program analysis. For example, a file has associated access permissions that cannot be known with certainty until it is opened. Although one security-typed programming language has included support for dynamic security labels, there has been no demonstration that a general mechanism for dynamic labels can securely control information flow. In this paper, we present an expressive language-based mechanism for reasoning about dynamic security labels. The mechanism is formally presented in a core language based on the typed lambda calculus; any well-typed program in this language is provably secure because it satisfies noninterference.

1 Introduction

Information flow control protects information security by constraining how information is transmitted among objects and users of various security classes. These security classes are expressed as *labels* associated with the information or its containers. Denning [6] showed how to use static analysis to ensure that programs use information in accordance with its security class, and this approach has been instantiated in a number of languages in which the type system implements a similar static analysis (e.g., [25, 10, 30, 19, 3, 21]). These type systems are an attractive way to enforce security because they can be shown to enforce *noninterference* [9], a strong, end-to-end security property. For example, when applied to confidentiality, noninterference ensures that confidential information cannot be released by the program no matter how it is transformed.

However, security cannot be enforced purely statically. In general, programs interact with an external environment that cannot be predicted at compile time, so there must be a run-time mechanism that allows security-critical decisions to be taken based on dynamic observations of this environment. For example, it is important to be able to change security settings on files and database records, and these changes should affect how the information from these sources can be used. A purely static mechanism cannot enforce this.

To securely control information flow when access rights can be changed and determined dynamically, *dynamic* labels [15] are needed that can be manipulated and checked at run time. However, manipulating labels dynamically makes it more difficult to enforce a strong notion of information security for several reasons. First, changing the label of an object may convert sensitive data to public data, directly violating noninterference. Second, label changes (and changes to access rights in general) can be used to convey information covertly; some restriction has to be imposed to prevent covert channels [27, 22]. Some mandatory access control (MAC) mechanisms support dynamic labels but cannot prevent *implicit flows* arising from control flow paths not taken at run time [5, 12].

JFlow [14] and its successor, Jif [17] are the only implemented security-typed languages supporting dynamic labels. However, although the Jif type system is designed to control the new information channels that dynamic labels create, it has not been proved to enforce secure information flow. Further, the dynamic label mechanism in Jif has limitations that impair expressiveness and efficiency.

In this paper, we propose an expressive language-based mechanism for securely manipulating information with dynamic security labels. The mechanism is formalized in a core language (based on the typed lambda calculus) with first-class label values, dependent security types and run-time label tests. Further, we prove that any well-typed program of the core language is secure because it satisfies noninterference. This is the first noninterference proof for a security-typed language in which general security labels can be manipulated and tested dynamically, though a noninterference result has been obtained for a simpler language supporting the related notion of dynamic *principals* [24].

Some previous MAC systems have supported dynamic se-

curity classes as part of a downgrading mechanism [23, 16], in this work we treat the two mechanisms orthogonally. While downgrading is important, it is useful to treat it as a separate mechanism so that labels can be manipulated dynamically while preserving noninterference.

The remainder of this paper is organized as follows. Section 2 presents some background on lattice label models and security type systems. Section 3 introduces the core language λ_{DSec} and uses sample λ_{DSec} programs to show some important applications of dynamic labels. Section 4 describes the type system of λ_{DSec} and proves the noninterference result. Section 5 covers related work, and Section 6 concludes.

2 Background

Static information flow analysis can be formalized as a security type system, in which security levels of data are represented by security type annotations, and information flow control is performed through type checking.

2.1 Security classes

We assume that security requirements for confidentiality or integrity are defined by associating *security classes* with users and with the resources that programs access. These security classes form a lattice $\mathcal{L}$. We write $k \sqsubseteq k'$ to indicate that security class k' is at least as restrictive as another security class k. In this case it is safe to move information from security class k to k', because restrictions on the use of the data are preserved. To control data derived from sources with classes k and k', the least restrictive security class that is at least as restrictive as both k and k' is assigned. This is the least upper bound, or join, written $k \sqcup k'$.

2.2 Labels

Type systems for confidentiality or integrity are concerned with tracking information flows in programs. Types are extended with security *labels* that denote security classes. A label ℓ appearing in a program may be simply a constant security class k, or a more complex expression that denotes a security class. The notation $\ell_1 \sqsubseteq \ell_2$ means that ℓ_2 denotes a security class that is at least as restrictive as that denoted by ℓ_1.

Because a given security class may be denoted by different labels, the relation $\sqsubseteq$ generates a lattice of *equivalence classes* of labels with $\sqcup$ as the *join* (least upper bound) operator. Two labels ℓ_1 and ℓ_2 are equivalent, written $\ell_1 \approx \ell_2$, if $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$. The join of two labels, $\ell_1 \sqcup \ell_2$, denotes the security class that is the join of the security classes that ℓ_1 and ℓ_2 denote. For example, if `x` has label ℓ_x and `y` has label ℓ_y, then the sum `x+y` is given the label $\ell_x \sqcup \ell_y$.

2.3 Security type systems for information flow

Security type systems can be used to enforce security information flows statically. Information flows in programs may be explicit flows such as assignments, or *implicit flows* [6] arising from the control flow of the program. Consider an assignment statement `x=y`, which contains an information flow from `y` to `x`. Then the typing rule for the assignment statement requires that $\ell_y \sqsubseteq \ell_x$, which means the security level of `y` is lower than the security level of `x`, guaranteeing the information flow from `y` to `x` is secure.

One advantage of static analysis is more precise control of implicit flows. Consider a simple conditional:

```
if b then x = true else x = false
```

Although there is no direct assignment from `b` to `x`, this expression has an implicit flow from `b` into `x`. A standard technique for controlling implicit flows is to introduce a *program-counter label* [5], written *pc*, which indicates the security level of the information that can be learned by knowing the control flow path taken thus far. In this example, the branch taken depends on `b`, so the *pc* in the `then` and `else` clauses will be joined with ℓ_b, the label of `b`. The type system ensures that any effect of expression e has a label at least as restrictive as its *pc*. In other words, an expression e cannot generate any effects observable to users who should not know the current program counter. In this example, the assignments to `x` will be permitted only if $pc \sqsubseteq \ell_x$, which ensures $\ell_b \sqsubseteq \ell_x$.

3 The λ_{DSec} language

The core language λ_{DSec} is a security-typed lambda calculus that supports first-class dynamic labels. In λ_{DSec}, labels are terms that can be manipulated and checked at run time. Furthermore, label terms can be used as statically analyzed type annotations. Syntactic restrictions are imposed on label terms to increase the practicality of type checking, following the approach used by Xi and Pfenning in $\mathtt{ML}_0^{\Pi}(C)$ [29].

From the computational standpoint, λ_{DSec} is fairly expressive, because it supports both first-class functions and state (which together are sufficient to encode recursive functions).

3.1 Syntax

The syntax of λ_{DSec} is given in Figure 1. We use the name k to range over a lattice of label values $\mathcal{L}$ (more precisely, a join semi-lattice with bottom element $\bot$), x, y to range over variable names $\mathcal{V}$, and m to range over a space of memory addresses $\mathcal{M}$.

To make the lattice explicit, we write $\mathcal{L} \models k_1 \sqsubseteq k_2$ to mean that k_2 is at least as restrictive as k_1 in $\mathcal{L}$, and $\mathcal{L} \models k = k_1 \sqcup k_2$ to mean k is the join of k_1 and k_2 in $\mathcal{L}$. The least and greatest elements of $\mathcal{L}$ are $\bot$ and $\top$. Any non-trivial label lattice contains at least two points L and H

$$
\begin{array}{rrcl}
\text{Base Labels} & k & \in & \mathcal{L} \\
\text{Variables} & x, y, f & \in & \mathcal{V} \\
\text{Locations} & m & \in & \mathcal{M} \\
\text{Labels} & \ell, \mathit{pc} & ::= & k \mid x \mid \ell_1 \sqcup \ell_2 \\
\text{Constraints} & C & ::= & \ell_1 \sqsubseteq \ell_2 \ , \ C \mid \epsilon \\
\text{Base Types} & \beta & ::= & \texttt{int} \mid \texttt{label} \mid \texttt{unit} \mid (x{:}\tau_1)[C] * \tau_2 \mid \tau \ \texttt{ref} \mid (x{:}\tau_1) \xrightarrow{C\,;\,\mathit{pc}} \tau_2 \\
\text{Security Types} & \tau & ::= & \beta_\ell \\
\text{Values} & v & ::= & x \mid n \mid m^\tau \mid \lambda(x{:}\tau)[C\,;\mathit{pc}].\, e \mid () \mid k \mid (x{=}v_1[C],\ v_2{:}\tau) \\
\text{Expressions} & e & ::= & v \mid \ell_1 \sqcup \ell_2 \mid e_1\ e_2 \mid\ !e \mid e_1 := e_2 \mid \texttt{ref}^\tau e \mid \texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & & \mid & \texttt{let}\ (x, y){=}e_1\ \texttt{in}\ e_2
\end{array}
$$

Figure 1: Syntax of λ_{DSec}

where $H \not\sqsubseteq L$. Intuitively, the label L describes what information is observable by *low-security users* who are to be prevented from seeing confidential information. Thus, *low-security* data has a label bounded above by L; *high-security* data has a label (such as H) not bounded by L.

In λ_{DSec}, a label can be either a label value k, a variable x, or the join of two other labels $\ell_1 \sqcup \ell_2$. For example, L, x, and $L \sqcup x$ are all valid labels, and $L \sqcup x$ can be interpreted as a security policy that is as restrictive as both L and x. The security type $\tau = \beta_\ell$ is the base type β annotated with label ℓ. The base types include integers, unit, labels, functions, references and products.

The function type $(x : \tau_1) \xrightarrow{C\,;\,\mathit{pc}} \tau_2$ is a dependent type since τ_1, τ_2, C and *pc* may mention x. The component C is a set of *label constraints* each with the form $\ell_1 \sqsubseteq \ell_2$; they must be satisfied when the function is invoked. The *pc* component is a lower bound on the memory effects of the function, and an upper bound on the *pc* label of the caller. Consequently, a function is not able to leak information about where it is called. Without the annotations C and *pc*, this kind of type is sometimes written as $\Pi x{:}\tau_1.\tau_2$ [13].

The product type $(x{:}\tau_1)[C] * \tau_2$ is also a dependent type in the sense that occurrences of x can appear in τ_1, τ_2 and C. The component C is a set of label constraints that any value of the product type must satisfy. If τ_2 does not contain x and C is empty, the type may be written as the more familiar $\tau_1 * \tau_2$. Without the annotation C, this kind of type is sometimes written $\Sigma x{:}\tau_1.\tau_2$ [13].

In λ_{DSec}, values include integers n, typed memory locations m^τ, functions $\lambda(x{:}\tau)[C\,;\mathit{pc}].\, e$, the unit value $()$, constant labels k, and pairs $(x = v_1[C],\ v_2 : \tau)$. A function $\lambda(x : \tau)[C\,;\mathit{pc}].\, e$ has one argument x with type τ, and the components C and *pc* have the same meanings as those in function types. The empty constraint set C or the top *pc* can be omitted. A pair $(x{=}v_1[C],\ v_2{:}\tau)$ contains two values v_1 and v_2. The second element v_2 has type τ and may mention the first element v_1 by the name x. The component C is a set of label constraints that the first element of the pair must satisfy. For example, suppose C contains the constraint $x \sqsubseteq L$, then $v_1 \sqsubseteq L$ must be true since inside the pair the value of x is v_1.

Expressions include values v, variables x, the join of two labels $\ell_1 \sqcup \ell_2$, applications $e_1\ e_2$, dereferences $!e$, assignments $e_1 := e_2$, references $\texttt{ref}^\tau e$, label-test expressions $\texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, and product destructors $\texttt{let}\ (x, y){=}v\ \texttt{in}\ e_2$.

The label-test expression $\texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ is used to examine labels. At run time, if the value of ℓ_2 is a constant label at least as restrictive as the value of ℓ_1, then e_1 is evaluated; otherwise, e_2 is evaluated. Consequently, the constraint $\ell_1 \sqsubseteq \ell_2$ can be assumed when type-checking e_1.

The product destructor $\texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2$ unpacks the result of e_1, which is a pair, assigns the first element to x and the second to y, and then evaluates e_2.

3.2 Operational Semantics

The small-step operational semantics of λ_{DSec} is given in Figure 2. Let M represent a memory that is a finite map from typed locations to closed values, and let $\langle e,\ M\rangle$ be a machine configuration. Then a small evaluation step is a transition from $\langle e,\ M\rangle$ to another configuration $\langle e',\ M'\rangle$, written $\langle e,\ M\rangle \longmapsto \langle e',\ M'\rangle$.

It is necessary to restrict the form of $\langle e,\ M\rangle$ to avoid using undefined memory locations. Let $\mathit{loc}(e)$ represent the set of memory locations appearing in e. A memory M is well-formed if every address m appears at most once in $\mathit{dom}(M)$, and for any m^τ in $\mathit{dom}(M)$, $\mathit{loc}(M(m^\tau)) \subseteq \mathit{dom}(M)$. The configuration $\langle e,\ M\rangle$ is well-formed if M is well-formed, $\mathit{loc}(e) \subseteq \mathit{dom}(M)$, and e contains no free variables. By induction on the derivation of $\langle e,\ M\rangle \longmapsto \langle e',\ M'\rangle$, we can prove that if $\langle e,\ M\rangle$ is well-formed, then $\langle e',\ M'\rangle$ is also well-formed.

The notation $e[v/x]$ indicates capture-avoiding substitution of value v for variable x in expression e. Unlike in the typed lambda calculus, $e[v/x]$ may generate a syntactically ill-formed expression if x appears in type annotations inside e, and v is not a label. However, this is not a problem because the type system of λ_{DSec} guarantees that a well-typed expression can only be evaluated to another well-typed and thus well-formed expression.

$$[E1] \quad \frac{\mathcal{L} \models k = k_1 \sqcup k_2}{\langle k_1 \sqcup k_2,\ M\rangle \longmapsto \langle k,\ M\rangle}$$

$$[E2] \quad \langle !m^\tau,\ M\rangle \longmapsto \langle M(m^\tau),\ M\rangle$$

$$[E3] \quad \frac{m \notin \mathit{address\text{-}space}(M)}{\langle \mathtt{ref}^\tau v,\ M\rangle \longmapsto \langle m^\tau,\ M[m^\tau \mapsto v]\rangle}$$

$$[E4] \quad \langle m^\tau := v,\ M\rangle \longmapsto \langle (),\ M[m^\tau \mapsto v]\rangle$$

$$[E5] \quad \langle (\lambda(x{:}\tau)[C\,;\mathit{pc}].\,e)\ v,\ M\rangle \longmapsto \langle e[v/x],\ M\rangle$$

$$[E6] \quad \frac{\mathcal{L} \models k_1 \sqsubseteq k_2}{\langle \mathtt{if}\ k_1 \sqsubseteq k_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\ M\rangle \longmapsto \langle e_1,\ M\rangle}$$

$$[E7] \quad \frac{\mathcal{L} \models k_1 \not\sqsubseteq k_2}{\langle \mathtt{if}\ k_1 \sqsubseteq k_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\ M\rangle \longmapsto \langle e_2,\ M\rangle}$$

$$[E8] \quad \langle \mathtt{let}\ (x,y){=}(x{=}v_1[C],\ v_2{:}\tau)\ \mathtt{in}\ e,\ M\rangle \longmapsto \langle e[v_2/y][v_1/x],\ M\rangle$$

$$[E9] \quad \frac{\langle e,\ M\rangle \longmapsto \langle e',\ M'\rangle}{\langle E[e],\ M\rangle \longmapsto \langle E[e'],\ M'\rangle}$$

$$\begin{array}{rcl} E[\cdot] & ::= & [\cdot]\ e \mid v\ [\cdot] \mid [\cdot] := e \mid v := [\cdot] \mid\ !\,[\cdot] \mid \mathtt{ref}^\tau\ [\cdot] \mid [\cdot] \sqcup \ell_2 \mid k_1 \sqcup [\cdot] \\ & \mid & \mathtt{if}\ [\cdot] \sqsubseteq \ell_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{if}\ k_1 \sqsubseteq [\cdot]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{let}\ (x,y){=}[\cdot]\ \mathtt{in}\ e \end{array}$$

Figure 2: Small-step operational semantics of λ_{DSec}

The notation $M(m^\tau)$ denotes the value of location m^τ in M, and the notation $M[m^\tau \mapsto v]$ denotes the memory obtained by assigning v to m^τ in M.

The evaluation rules are standard. In rule (E3), the notation *address-space*(M) represents the set of location names in M, that is, $\{m \mid \exists \tau\ s.t.\ m^\tau \in \mathit{dom}(M)\}$. In rule (E8), v_2 may mention x, so substituting v_2 for y in e is performed before substituting v_1 for x. The variable name in the product value matches x so that no variable substitution is needed when assigning v_1 and v_2 to x and y. In rule (E9), E represents an evaluation context, a term with a single hole in redex position, and the syntax of E specifies the evaluation order.

3.3 Examples

As discussed in Section 1, dynamic labels are vital for precisely controlling information flows between security-typed programs and the external environment. A practical program often needs to access files or communicate through networks. These activities can be viewed as communication through an *I/O channel* with a corresponding label consistent with the security policy of the entity (such as a file or network socket) represented by the channel. Because the security policy of an external entity may be discovered and even changed at run time, the precise label of an I/O channel is dynamic and operations on a channel cannot be checked at compile time.

3.3.1 Run-time access control

Implementing run-time access control is one of the most important applications of dynamic label mechanisms. Suppose there exists a file that stores one integer, and the access control policy of the file is unknown at compile time. In λ_{DSec}, the file can be encoded as a reference of type $(x : \mathtt{label}_\bot) * (\mathtt{int}_x\ \mathtt{ref})_\bot$, where x is a dynamic label consistent with the access control policy of the file, and the reference component of type $(\mathtt{int}_x\ \mathtt{ref})_\bot$ stores the contents of the file. Thus storing an integer of type $\mathtt{int}_H$ in the file is equivalent to assigning the integer to the memory reference component, which requires that x is at least as high as H. Since the value of x is not known at compile time, the condition $H \sqsubseteq x$ can only be checked at run time, using a label-test expression. The following function stores a high-security integer z in the file w:

$$\lambda w{:}((x{:}\mathtt{label}_\bot) * (\mathtt{int}_x\ \mathtt{ref})_\bot)_\bot\ \mathtt{ref}_\bot.\,\lambda(z{:}\mathtt{int}_H)[H].\\ \mathtt{let}\ (x,y){=}!w\ \mathtt{in}\ \mathtt{if}\ H \sqsubseteq x\ \mathtt{then}\ y := z\ \mathtt{else}\ ()$$

Note that the *pc* label of the function is H because the function body contains a memory effect of label x when $H \sqsubseteq x$.

It is also important to be able to change file permissions at run time. The following code changes the access control policy of the file w to label z. However, the original contents of w need to be wiped out to prevent them from being implic-

itly declassified, which provides stronger security assurance than an ordinary file system.

$$\lambda w{:}((x{:}\texttt{label}_\bot) * \texttt{int}_x\ \texttt{ref}_\bot)_\bot\ \texttt{ref}_\bot.\ \lambda(z{:}\texttt{label}_\bot)[\bot].\\ (\lambda(y{:}\texttt{int}_z\ \texttt{ref}_\bot)[\bot].\ w := (x{=}z,\ y{:}\texttt{int}_x\ \texttt{ref}_\bot))\texttt{ref}^{\texttt{int}_z}0$$

3.3.2 Multilevel communication channels

Information flows inside a program are controlled by static type checking. When information is exported outside a program through an I/O channel, the receiver might want to know the exact label of the information, which calls for *multilevel communication channels* [7] unambiguously pairing the information sent or received with its corresponding security label. Supporting multilevel channels is one of the basic requirements for a MAC system [7].

In λ_{DSec}, a multilevel channel can be encoded by a memory reference of type $((x : \texttt{label}_x) * \texttt{int}_x)_\bot\ \texttt{ref}$, which stores a pair composed of an integer value and its label. The confidentiality of the integer component is protected by the label component, since extracting the integer component from such a pair requires testing the label component:

$$\lambda z{:}((x{:}\texttt{label}_x) * \texttt{int}_x)_\bot.\ \texttt{let}\ (x,y){=}z\ \texttt{in}\\ \texttt{if}\ x \sqsubseteq L\ \texttt{then}\ m^{\texttt{int}_L} := y\ \texttt{else}\ ()$$

In the above example, the constraint $x \sqsubseteq L$ must be satisfied in order to store the integer component in $m^{\texttt{int}_L}$. Since the readability of the integer component depends on the value of x, letting x recursively label itself ensures that all the authorized readers of the integer component can test x and retrieve the integer value.

Sending an integer through a multilevel channel is implemented by pairing the integer and its label and storing the pair in the reference representing the channel:

$$\lambda z{:}(((x{:}\texttt{label}_x)) * \texttt{int}_x)_\bot\ \texttt{ref})_\bot.\ \lambda w{:}\texttt{label}_w.\\ \lambda(y{:}\texttt{int}_w)[\bot].\ z := (x{=}w,\ y{:}\texttt{int}_x)$$

Like other I/O channels, a multilevel channel may have a label that is an upper bound of the security levels of the information that can be sent through the channel. Product label constraints can be used to specify the label of a multilevel channel. For example, a bounded multilevel channel can be represented by a memory reference with type $((x : \texttt{label}_x)[x \sqsubseteq \ell] * \texttt{int}_x)_\bot\ \texttt{ref}$, where ℓ is the label of the channel, and the constraint $x \sqsubseteq \ell$ guarantees any information stored in the reference has a security label at most as high as ℓ. Sending information through a bounded multilevel channel often needs a run-time check as in the following code:

$$\lambda z{:}(((x{:}\texttt{label}_x))[x \sqsubseteq \ell] * \texttt{int}_x)_\bot\ \texttt{ref})_\bot.\ \lambda w{:}\texttt{label}_w.\\ \lambda(y{:}\texttt{int}_w)[\bot].\ \texttt{if}\ w \sqsubseteq \ell\ \texttt{then}\ z := (x{=}w,\ y{:}\texttt{int}_x)\ \texttt{else}\ ()$$

The ability to recursively use a variable to construct the label of its own type provides a useful kind of polymorphism, which this example demonstrates. Without recursive labels, the type of a multilevel channel cannot be constructed so simply, because selecting a label for the label component x becomes problematic. Any constant label that is chosen may be inappropriate; for example, if the label has the label $\bot$ then it may be impossible to compute a suitable label to supply as x. Another possibility is to provide yet another label that is to function as the label of x, but this merely pushes the problem back by one level. Giving x the type $\texttt{label}_x$ is a neat way to tie off this sequence.

4 Type system and noninterference

This section describes the type system of λ_{DSec} and proves that the type system guarantees that any well-typed program has the noninterference property.

4.1 Label constraints

Because of dynamic labels, it is not always possible to decide whether the relationship $\ell_1 \sqsubseteq \ell_2$ holds at compile time; therefore, the label-test expression (`if`) must be used to query the relationship. However, this dynamic query may create new information flows; the language λ_{DSec} and its type system are designed to statically control these new information flows.

Although labels are first-class values in λ_{DSec}, label terms have a restricted syntactic form so that any label term can be used as a type annotation. Therefore, constraints on label terms are also type-level information that can be used by the type checker.

Furthermore, in λ_{DSec} label terms are purely functional: they have no side effects and evaluate to the same value in the same context. As a result, any label constraint of the form $\ell_1 \sqsubseteq \ell_2$ that is known to hold in a typing context can be used for type checking in that context. For example, consider the following code:

$$\lambda x{:}\texttt{label}_\bot.\ \lambda y{:}(\texttt{int}_x\ \texttt{ref})_\bot.\ \lambda(z{:}\texttt{int}_H)[H].\\ \texttt{if}\ H \sqsubseteq x\ \texttt{then}\ y := z\ \texttt{else}\ ()$$

According to the semantics of the label-test expression, the assignment $y := z$ will be executed only if $H \sqsubseteq x$ holds. Thus, the constraint $H \sqsubseteq x$ can be used to decide whether $z := y$ is secure. In this example, any information stored in z is only accessible to users with security level at least as high as x. So it is secure to store z in y because x is at least as high as H.

In general, for each expression e, the type checker keeps track of the set of constraints C that are known to be satisfied when e is executed, and uses C in type-checking e.

Another common approach for relating type information to term-level constructs is to use singleton types, types containing only one value [2]. We have chosen to use dependent types because it is the approach used by Jif, and the approach based on singleton types neither provides more expressiveness nor simplifies the type system or the noninterference

$$[C1]\quad \frac{\mathcal{L} \models k_1 \sqsubseteq k_2}{C \vdash k_1 \sqsubseteq k_2} \qquad [C2]\quad \frac{\ell_1 \sqsubseteq \ell_2 \in C}{C \vdash \ell_1 \sqsubseteq \ell_2}$$

$$[C3]\quad C \vdash \ell \sqsubseteq \top \qquad [C4]\quad C \vdash \bot \sqsubseteq \ell$$

$$[C5]\quad C \vdash \ell \sqsubseteq \ell \sqcup \ell'$$

$$[C6]\quad \frac{C \vdash \ell_1 \sqsubseteq \ell_2 \qquad C \vdash \ell_2 \sqsubseteq \ell_3}{C \vdash \ell_1 \sqsubseteq \ell_3}$$

$$[C7]\quad \frac{C \vdash \ell_1 \sqsubseteq \ell_3 \qquad C \vdash \ell_2 \sqsubseteq \ell_3}{C \vdash \ell_1 \sqcup \ell_2 \sqsubseteq \ell_3}$$

Figure 3: Relabeling rules

$$[S1]\quad \frac{C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_2 \leq \tau_1}{C \vdash \tau_1 \ \texttt{ref} \leq \tau_2 \ \texttt{ref}}$$

$$[S2]\quad \frac{\begin{array}{c} C \vdash \tau_2 \leq \tau_1 \quad C \vdash \tau_1' \leq \tau_2' \\ C \vdash \mathit{pc}_2 \sqsubseteq \mathit{pc}_1 \quad C, C_2 \vdash C_1 \end{array}}{C \vdash (x{:}\tau_1) \xrightarrow{C_1\,;\,\mathit{pc}_1} \tau_1' \leq (x{:}\tau_2) \xrightarrow{C_2\,;\,\mathit{pc}_2} \tau_2'}$$

$$[S3]\quad \frac{C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_1' \leq \tau_2' \quad C, C_1 \vdash C_2}{C \vdash (x{:}\tau_1)[C_1] * \tau_1' \leq (x{:}\tau_2)[C_2] * \tau_2'}$$

$$[S4]\quad \frac{C \vdash \beta_1 \leq \beta_2 \quad C \vdash \ell_1 \sqsubseteq \ell_2}{C \vdash (\beta_1)_{\ell_1} \leq (\beta_2)_{\ell_2}}$$

Figure 4: Subtyping rules

proof in any substantial way. In general, we feel that the choice between dependent types and singletons is a matter of taste.

4.2 Subtyping

The subtyping relationship between security types plays an important role in enforcing information flow security. Given two security types $\tau_1 = \beta_{1_{\ell_1}}$ and $\tau_2 = \beta_{2_{\ell_2}}$, suppose τ_1 is a subtype of τ_2, written as $\tau_1 \leq \tau_2$. Then any data of type τ_1 can be treated as data of type τ_2. Thus, data with label ℓ_1 may be treated as data with label ℓ_2, which requires $\ell_1 \sqsubseteq \ell_2$.

The type system keeps track of the set of label constraints that can be used to prove relabeling relationships between labels. Let $C \vdash \ell_1 \sqsubseteq \ell_2$ denote that $\ell_1 \sqsubseteq \ell_2$ can be inferred from the set of constraints C. The inference rules are shown in Figure 3; they are standard and consistent with the lattice properties of labels. Rule (C2) shows that all the constraints in C are assumed to be true. The constraint set C may contain constraints that are inconsistent with the lattice $\mathcal{L}$, such as $H \sqsubseteq L$. Inconsistent constraint sets are harmless because they always indicate dead code, such as expression e_1 in "$\texttt{if}\ H \sqsubseteq L\ \texttt{then}\ e_1\ \texttt{else}\ e_2$".

Since the subtyping relationship depends on the relabeling relationship, the subtyping context also needs to include the C component. The inference rules for proving $C \vdash \tau_1 \leq \tau_2$ are the rules shown in Figure 4 plus the standard reflexivity and transitivity rules.

Rules (S1)–(S3) are about subtyping on base types. These rules demonstrate the expected covariance or contravariance. In λ_{DSec}, function types contain two additional components *pc* and C, both of which are contravariant. Suppose the function type $\tau = (x : \tau_1) \xrightarrow{C_1\,;\,\mathit{pc}_1} \tau_1'$ is a subtype of $\tau' = (x{:}\tau_2) \xrightarrow{C_2\,;\,\mathit{pc}_2} \tau_2'$. Then wherever functions with type τ' can be called, functions with type τ can also be called. This implies two necessary premises. First, wherever C_2 is satisfied, C_1 is also satisfied. Since C is satisfied, this premise is written $C, C_2 \vdash C_1$, meaning that for any constraint $\ell_1 \sqsubseteq \ell_2$ in C_1, we can derive $C, C_2 \vdash \ell_1 \sqsubseteq \ell_2$. Second, the premise $\mathit{pc}_2 \sqsubseteq \mathit{pc}_1$ is needed because the *pc* of a function type is an upper bound on the *pc* where the function is applied.

In rules (S2) and (S3), variable x is bound in the function and product types. For simplicity, we assume that x does not appear in C, since α-conversion can always be used to rename x to another fresh variable. This assumption also applies to the typing rules.

Rule (S4) is used to determine the subtyping on security types. The premise $C \vdash \beta_1 \leq \beta_2$ is natural. The other premise $C \vdash \ell_1 \sqsubseteq \ell_2$ guarantees that coercing data from τ_1 to τ_2 does not violate information flow policies.

4.3 Typing

The type system of λ_{DSec} prevents illegal information flows and guarantees that well-typed programs have a noninterference property. The typing rules are shown in Figure 5. The notation $\mathit{label}(\beta_\ell) = \ell$ is used to obtain the label of a type, and the notations $\ell \sqsubseteq \tau$ and $\tau \sqsubseteq \ell$ are abbreviations for $\ell \sqsubseteq \mathit{label}(\tau)$ and $\mathit{label}(\tau) \sqsubseteq \ell$, respectively.

The typing context includes a *type assignment* Γ, a set of constraints C and the program-counter label *pc*. Γ is a finite *ordered* list of $x : \tau$ pairs in the order that they came into scope. For a given x, there is at most one pair $x{:}\tau$ in Γ.

A variable appearing in a type must be a label variable. Therefore, a type τ is well-formed with respect to type assignment Γ, written $\Gamma \vdash \tau$, if Γ maps all the variables in τ to label types. The definition of well-formed labels ($\Gamma \vdash \ell$) is the same. Consider $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. For any $0 \leq i \leq n$, the type τ_i may only mention label variables that are already in scope: x_1 through x_i. Therefore, Γ is well-formed if for any $0 \leq i \leq n$, τ_i is well-formed with respect to $x_1 : \tau_1, \ldots, x_i : \tau_i$. For example, "$x : \texttt{label}_L, y : \texttt{int}_x$" is well-formed, but "$y : \texttt{int}_x, x : \texttt{label}_L$" is not. A constraint $\ell_1 \sqsubseteq \ell_2$ is well-formed with respect to Γ if both ℓ_1 and ℓ_2 are well-formed with respect to Γ. A typing context "$\Gamma\,;C\,;\mathit{pc}$" is well-formed if Γ is well-formed, and *pc* and all the constraints in C are well-formed with respect to Γ.

$$[\textit{INT}] \quad \Gamma\,;C\,;\,\textit{pc} \vdash n : \texttt{int}_\bot \qquad [\textit{UNIT}] \quad \Gamma\,;C\,;\,\textit{pc} \vdash () : \texttt{unit}_\bot$$

$$[\textit{LABEL}] \quad \Gamma\,;C\,;\,\textit{pc} \vdash k : \texttt{label}_\bot \qquad [\textit{LOC}] \quad \frac{\textit{FV}(\tau) = \emptyset}{\Gamma\,;C\,;\,\textit{pc} \vdash m^\tau : (\tau\ \texttt{ref})_\bot}$$

$$[\textit{JOIN}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash \ell_1 : \texttt{label}_{\ell'_1} \quad \Gamma\,;C\,;\,\textit{pc} \vdash \ell_2 : \texttt{label}_{\ell'_2}}{\Gamma\,;C\,;\,\textit{pc} \vdash \ell_1 \sqcup \ell_2 : \texttt{label}_{\ell'_1 \sqcup \ell'_2}} \qquad [\textit{VAR}] \quad \frac{x{:}\tau \in \Gamma}{\Gamma\,;C\,;\,\textit{pc} \vdash x : \tau}$$

$$[\textit{REF}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau \qquad C \vdash \textit{pc} \sqsubseteq \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash \texttt{ref}^\tau e : (\tau\ \texttt{ref})_\bot} \qquad [\textit{DEREF}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e : (\tau\ \texttt{ref})_\ell}{\Gamma\,;C\,;\,\textit{pc} \vdash !e : \tau \sqcup \ell}$$

$$[\textit{ABS}] \quad \frac{\Gamma, x{:}\tau'\,;C'\,;\textit{pc}' \vdash e : \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash \lambda(x{:}\tau')[C'\,;\textit{pc}'].\,e : ((x{:}\tau') \xrightarrow{C'\,;\,\textit{pc}'} \tau)_\bot} \qquad [\textit{ASSIGN}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : (\tau\ \texttt{ref})_\ell \quad \Gamma\,;C\,;\,\textit{pc} \vdash e_2 : \tau \quad C \vdash \textit{pc} \sqcup \ell \sqsubseteq \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash e_1 := e_2 : \texttt{unit}_\bot}$$

$$[\textit{L-APP}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : ((x{:}\texttt{label}_{\ell'}) \xrightarrow{C'\,;\,\textit{pc}'} \tau)_\ell \\ \Gamma\,;C\,;\,\textit{pc} \vdash \ell_2 : \texttt{label}_{\ell'[\ell_2/x]} \\ C \vdash \textit{pc} \sqcup \ell \sqsubseteq \textit{pc}'[\ell_2/x] \qquad C \vdash C'[\ell_2/x] \\ x \in \textit{FV}(\tau) \cup \textit{FV}(\ell') \cup \textit{FV}(C') \cup \textit{FV}(\textit{pc}')\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash e_1\ \ell_2 : \tau[\ell_2/x] \sqcup \ell} \qquad [\textit{APP}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : ((x{:}\tau') \xrightarrow{C'\,;\,\textit{pc}'} \tau)_\ell \\ \Gamma\,;C\,;\,\textit{pc} \vdash e_2 : \tau' \\ C \vdash \textit{pc} \sqcup \ell \sqsubseteq \textit{pc}' \qquad C \vdash C' \\ x \notin \textit{FV}(\tau) \cup \textit{FV}(\tau') \cup \textit{FV}(C') \cup \textit{FV}(\textit{pc}')\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash e_1\ e_2 : \tau \sqcup \ell}$$

$$[\textit{PROD}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash v_1 : \tau_1[v_1/x] \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 \\ \Gamma\,;C\,;\,\textit{pc} \vdash v_2[v_1/x] : \tau_2[v_1/x] \qquad C \vdash C'[v_1/x]\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash (x{=}v_1[C'],\ v_2{:}\tau_2) : ((x{:}\tau_1)[C'] * \tau_2)_\bot} \qquad [\textit{UNPACK}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : ((x{:}\tau_1)[C'] * \tau_2)_\ell \\ \Gamma, x{:}\tau_1 \sqcup \ell, y{:}\tau_2 \sqcup \ell\,;\,C, C'\,;\,\textit{pc} \vdash e_2 : \tau\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash \texttt{let}\ (x, y){=}e_1\ \texttt{in}\ e_2 : \tau}$$

$$[\textit{IF}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash \ell_i : \texttt{label}_{\ell'_i}\ \ i \in \{1,2\} \\ \Gamma\,;\,C, \ell_1 \sqsubseteq \ell_2\,;\,\textit{pc} \sqcup \ell'_1 \sqcup \ell'_2 \vdash e_1 : \tau \\ \Gamma\,;C\,;\textit{pc} \sqcup \ell'_1 \sqcup \ell'_2 \vdash e_2 : \tau\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash \texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \sqcup \ell'_1 \sqcup \ell'_2} \qquad [\textit{SUB}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau \qquad C \vdash \tau \leq \tau'}{\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau'}$$

Figure 5: Typing rules for the λ_{DSec} language

The typing assertion $\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau$ means that with the type assignment Γ, current program-counter label as *pc*, and the set of constraints C satisfied, expression e has type τ. The assertion $\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau$ is well-formed if $\Gamma\,;C\,;\,\textit{pc}$ is well-formed, and $\Gamma \vdash \tau$.

Rules (INT), (UNIT), (LABEL) and (LOC) are used to check values. Value v has type $\beta_\bot$ if v has base type β. Rule (LOC) requires typed location m^τ contain no label variables so that m^τ remains a constant during evaluation. This is enforced by the premise $\textit{FV}(\tau) = \emptyset$, where $\textit{FV}(\tau)$ denotes the set of free variables appearing in τ.

Rule (VAR) is standard: variable x has type $\Gamma(x)$. Rule (JOIN) checks the join of two labels and assigns a result label that is the join of the labels of the operands.

Rule (REF) checks memory allocation operations. If the *pc* label is high, the generated memory location must not be observable to low-security users, which is guaranteed by the premise $C \vdash \textit{pc} \sqsubseteq \tau$. Rule (DEREF) checks dereference expressions. Since some information about a reference can be learned by knowing its contents, the result of dereferencing a reference with type $(\tau\ \texttt{ref})_\ell$ has type $\tau \sqcup \ell$, where $\tau \sqcup \ell = \beta_{\ell' \sqcup \ell}$ if τ is $\beta_{\ell'}$.

Rule (ASSIGN) checks memory update. As in rule (REF), if the updated memory location has type $(\tau\ \texttt{ref})_\ell$, then $C \vdash \textit{pc} \sqsubseteq \tau$ is required to prevent illegal implicit flows. In addition, the premise $C \vdash \textit{pc} \sqcup \ell \sqsubseteq \tau$ implies another condition $C \vdash \ell \sqsubseteq \tau$ that is required to protect the confidentiality of the reference that is assigned to. Consider the following code that allows low-security users to learn whether $x \sqsubseteq L$ by observing which of m_1 and m_2 is updated to 0:

$$\lambda(x{:}\texttt{label}_H)[L].\,((\texttt{if}\, x \sqsubseteq L\, \texttt{then}\, m_1^{\texttt{int}_L}\, \texttt{else}\, m_2^{\texttt{int}_L}) := 0)$$

The code is not well-typed because the condition $C \vdash \ell \sqsubseteq \tau$ does not hold for the assignment expression.

Rule (ABS) checks function values. The body is checked with the constraint set C' and the program-counter label $\textit{pc}'$, so the function can only be called at places where C' is satisfied and the *pc* label is not more restrictive than $\textit{pc}'$.

Rule (L-APP) is used to check applications of dependent functions. Expression e_1 has a dependent function type $((x{:}$

$\texttt{label}_{\ell'}) \xrightarrow{C';\, pc'} \tau)_\ell$, where x does appear in ℓ', C', pc' or τ. As a result, rule (L-APP) needs to use $\ell'[\ell_2/x]$, $C'[\ell_2/x]$, $pc'[\ell_2/x]$ and $\tau[\ell_2/x]$, which are well-formed since ℓ_2 is a label. That also explains why e_1, with its dependent function type, cannot be applied to an arbitrary expression e_2: substituting e_2 for x in ℓ', C', pc' and τ may generate ill-formed labels or types, and it is generally unacceptable for the type checker to evaluate e_2 to value v_2 and substitute v_2 for x, which would make type-checking undecidable. The expressiveness of λ_{DSec} is not substantially affected by the restriction that a dependent function can only be applied to label terms, because the function can be applied to a variable that receives the result of an arbitrary expression. For example, in the following code, the application $e_1 x$ indirectly applies e_1 to e_2:

$$(\lambda x{:}\texttt{label}_\ell.\, \texttt{if}\ x \sqsubseteq L\ \texttt{then}\ e_1 x\ \texttt{else}\ ())e_2$$

This works as long as the function enclosing $e_1 x$ is not dependent.

In rule (L-APP), the label of $e_1 \ell_2$ is at least as restrictive as ℓ, preventing the result of e_1 from being leaked. The premise $C \vdash C'[\ell_2/x]$ guarantees that $C'[\ell_2/x]$ are satisfied when the function is invoked. The premise $C \vdash pc \sqcup \ell \sqsubseteq pc'[\ell_2/x]$ ensures that the invocation cannot leak the program counter or the function itself through the memory effects of the function.

Rule (APP) applies when x does not appear in C', pc' or τ. In this case, the type of e_1 is just a normal function type, so e_1 can be applied to arbitrary terms.

Rule (PROD) is used to check product values. To check v_2, the occurrences of x in v_2 and τ_2 are both replaced by v_1, since x is not in the domain of Γ. If v_1 is not a label, then x cannot appear in τ_2. Thus, $\tau_2[v_1/x]$ is always well-formed no matter whether v_1 is a label or not. Similarly, the occurrences of x in τ_1 and C' are also replaced by v_1 when v_1 and C' are checked.

Rule (UNPACK) checks product destructors straightforwardly. After unpacking the product value, those product label constraints in C' are in scope and used for checking e_2.

Rule (IF) checks label-test expressions. The constraint $\ell_1 \sqsubseteq \ell_2$ is added into the typing context when checking the first branch e_1. When checking the branches, the program-counter label subsumes the labels of ℓ_1 and ℓ_2 to protect them from implicit flows. The resulting type contains ℓ_1' and ℓ_2' because the result is influenced by the values of ℓ_1 and ℓ_2.

Rule (SUB) is the standard subsumption rule. If τ is a subtype of τ' with the constraints in C satisfied, then any expression of type τ also has type τ'.

This type system satisfies the subject reduction property and the progress property. The proof is standard, so we simply state the theorems here.

Definition 4.1 (Well-typed memory). A memory M is well-typed if for any memory location m^τ in M, $\vdash M(m^\tau) : \tau$.

Theorem 4.1 (Subject reduction). Suppose $pc \vdash e : \tau$, and there exists a well-typed memory M such that $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$, then M' is well-typed, and $pc \vdash e' : \tau$.

Theorem 4.2 (Progress). If $pc \vdash e : \tau$, and M is a well-typed memory such that $\langle e,\, M\rangle$ is a well-formed configuration, then either e is a value or there exists e' and M' such that $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$.

4.4 Noninterference proof

This section outlines a proof that any well-typed program in λ_{DSec} satisfies the noninterference property. (The full proof is given in the appendix.) Consider an expression e in λ_{DSec}. Suppose e has one free variable x, and $x : \tau \vdash e : \texttt{int}_L$ where $H \sqsubseteq \tau$. Thus, the value of x is a high-security input to e, and the result of e is a low-security output. Then noninterference requires that for all values v of type τ, evaluating $e[v/x]$ in the same memory must generate the same result, if the evaluation terminates. For simplicity, we only consider that results are integers because they can be compared outside the context of λ_{DSec}.

The noninterference property discussed here is *termination insensitive* [21] because $e[v/x]$ is required to generate the same result only if the evaluation terminates. The type system of λ_{DSec} does not attempt to control termination and timing channels. Control of these channels is largely an orthogonal problem. Termination channels can leak at most one bit per run, so they have often been considered acceptable (e.g., [6, 25]). Some recent work [1, 20, 31] partially addresses the control of timing channels.

Let $\longmapsto^*$ denote the transitive closure of the $\longmapsto$ relationship. The following theorem formalizes the claim that the type system of λ_{DSec} enforces noninterference:

Theorem 4.3 (Noninterference). Suppose $x : \tau \vdash e : \texttt{int}_L$, and $H \sqsubseteq \tau$. Given two arbitrary values v_1 and v_2 of type τ, and an initial memory M, if $\langle e[v_i/x],\, M\rangle \longmapsto^* \langle v_i',\, M_i'\rangle$ for $i \in \{1, 2\}$, then $v_1' = v_2'$.

To prove this noninterference theorem, we adapt the elegant proof technique developed by Pottier and Simonet for an ML-like security-typed language [19] (which did not have dynamic labels). To show that noninterference holds, it is necessary to reason about the executions of two related terms: $e[v_1/x]$ and $e[v_2/x]$. We extend λ_{DSec} with a bracket construct $(e_1 \mid e_2)$ that represents alternative expressions that might arise during the evaluation of two programs that differs initially only in v_1 and v_2. Then $e[v_1/x]$ and $e[v_2/x]$ can be incorporated into a single term $e[(v_1 \mid v_2)/x]$ in the extended language λ^2_{DSec}, providing a syntactic way to reason about two executions.

Using λ^2_{DSec}, the noninterference theorem can be proved in three steps:

1. Prove that the evaluation of λ^2_{DSec} adequately represents the execution of two λ_{DSec} terms. Given a λ^2_{DSec}

term e, let $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ represent the two λ_{DSec} terms encoded by e. Further, if M maps x to a λ^2_{DSec} term e, then $\lfloor M \rfloor_i$ maps x to $\lfloor e \rfloor_i$ for $i \in \{1, 2\}$. Then we can formalize the adequacy of λ^2_{DSec} as the following two lemmas (their proof is straightforward):

Lemma 4.1 (Soundness). If $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$, then $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i\rangle \longmapsto \langle \lfloor e' \rfloor_i,\, \lfloor M' \rfloor_i\rangle$ for $i \in \{1, 2\}$.

Lemma 4.2 (Completeness). If $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i\rangle \longmapsto^* \langle v_i,\, M'_i\rangle$ for $i \in \{1, 2\}$, then there exists a configuration $\langle v,\, M'\rangle$ such that $\langle e,\, M\rangle \longmapsto^* \langle v,\, M'\rangle$.

2. Prove that λ^2_{DSec} satisfies subject reduction: the result of an expression has the same type as the expression. The type system of λ^2_{DSec} explicitly enforces noninterference by requiring that any bracket expression $(e_1|e_2)$ has a high-security type. The differences between two executions are completely captured by bracket expressions, so the requirement that brackets must have high-security types ensures that the differences between the two executions are unobservable to low-security users. Intuitively, it is because of the explicit enforcement of noninterference that the noninterference theorem of λ_{DSec} can be reduced to the soundness (subject reduction) of the type system of λ^2_{DSec}.

3. Prove the noninterference theorem of λ_{DSec}: Because $\langle e[v_i/x],\, M\rangle \longmapsto^* \langle v'_i,\, M'_i\rangle$ and $e[v_i/x] = \lfloor e[(v_1 \mid v_2)/x] \rfloor_i$ for $i \in \{1, 2\}$, by the completeness lemma there exists $\langle v',\, M'\rangle$ such that $\langle e[(v_1 \mid v_2)/x],\, M\rangle \longmapsto^* \langle v',\, M'\rangle$. Moreover, $\lfloor v' \rfloor_i = v'_i$ for $i \in \{1, 2\}$ by the soundness lemma. To prove the noninterference theorem, we only need to prove $v'_1 = v'_2$, that is, $\lfloor v' \rfloor_1 = \lfloor v' \rfloor_2$. By the subject reduction theorem of λ^2_{DSec}, $\vdash v' : \mathtt{int}_L$. By the type system of λ^2_{DSec}, v' cannot be a bracket construct because it has a low-security type. Consequently, v' must be an integer n. Then we have $\lfloor v' \rfloor_1 = n = \lfloor v' \rfloor_2$.

The appendix details the syntax and semantic extensions of λ^2_{DSec} and proves the key subject reduction theorem of λ^2_{DSec}. The major extension to Pottier's proof technique is that the bracket construct must also be applied to labels. Because types may contain bracketed labels, the projection operation also applies to typing environments.

5 Related Work

Dynamic information flow control mechanisms [26, 27] track security labels dynamically and use run-time security checks to constrain information propagation. These mechanisms are transparent to programs, but they cannot prevent illegal implicit flows arising from the control flow paths not taken at run time.

Various general security models [11, 23, 8] have been proposed to incorporate dynamic labeling. Unlike noninterference, these models define what it means for a system to be secure according to a certain relabeling policy, which may allow downgrading labels.

Using static program analysis to check information flow was first proposed by Denning and Denning [6]; later work phrased the analysis as type checking (e.g., [18]). Noninterference was later developed as a more semantic characterization of security [9], followed by many extensions. Volpano, Smith and Irvine [25] first showed that type systems can be used to enforce noninterference, and proved a version of noninterference theorem for a simple imperative language, starting a line of research pursuing the noninterference result for more expressive security-typed languages. Heintze and Riecke [10] proved the noninterference theorem for the SLam calculus, a purely functional language. Zdancewic and Myers [30] investigated a secure calculus with first-class continuations and references. Pottier and Simonet [19] considered an ML-like functional language and introduced the proof technique that is extended in this paper. A more complete survey of language-based information-flow techniques can be found in [21, 32].

The Jif language [14, 17] extends Java with a type system for analyzing information flow, and aims to be a practical language for developing secure applications. However, there is not yet a noninterference proof for the type system of Jif, because of its complexity. This work is inspired by the dynamic label mechanism of Jif, although the dynamic label mechanism in λ_{DSec} is more expressive. Jif provides two constructs for run-time label tests: the `switch-label` statement and the `actsFor` statement, both of which can be encoded using the label-test expression in λ_{DSec}. The typing rules for `switch-label` and `actsFor` are as restrictive as the typing rule of the label-test expression. Thus, the noninterference result for λ_{DSec} provides strong evidence that these dynamic label constructs in Jif are secure.

Banerjee and Naumann [4] proved a noninterference result for a Java-like language with simple access control primitives. Unlike in λ_{DSec}, run-time access control in this language is separate from the static label mechanism because it is inspired by Java stack inspection. In their language, the label of a method result may depend in limited ways on the (implicit) security state of its caller; however, it does not seem to be possible in the language to control the flow of information from an I/O channel or file based on permissions discovered at run time.

Concurrent to our work, Tse and Zdancewic proved a noninterference result for a security-typed lambda calculus ($\lambda_{\mathtt{RP}}$) with run-time principals [24], which can be used to construct dynamic labels. However, $\lambda_{\mathtt{RP}}$ does not support references or existential types, which makes it unable to represent dynamic security policies that may be changed at run time, such as file permissions. As discussed in Section 1, modeling real

systems requires this ability. By comparison, in λ_{DSec} the label stored in a reference may be updated at run time, and with dependent existential types, we can ensure that a piece of data and its label are updated consistently. Therefore, updating a label dynamically does not declassify confidential data. In addition, support for references makes λ_{DSec} more powerful than λ_{RP} computationally.

Other work [29, 28] has used dependent type systems to specify complex program invariants and to statically catch program errors considered run-time errors by traditional type systems. This work also makes a trade-off between expressive power and practical type checking.

6 Conclusions

This paper formalizes computation and static checking of dynamic labels in the type system of a core language λ_{DSec} and proves a noninterference result: well-typed programs have the noninterference property. The language λ_{DSec} is the first language supporting general dynamic labels whose type system provably enforces noninterference.

Acknowledgements

The authors would like to thank Greg Morrisett, Steve Zdancewic and Amal Ahmed for their insightful suggestions. Steve Chong, Nate Nystrom, and Michael Clarkson also helped improve the presentation of this work.

References

[1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[2] David Aspinall. Subtyping with singleton types. In *Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.

[3] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[4] Anindya Banerjee and David A. Naumann. Using access control for secure information flow in a java-like language. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 155–169, June 2003.

[5] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[6] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[7] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[8] Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *IEEE Symposium on Security and Privacy*, pages 142–154, Oakland, CA, 1996. IEEE Computer Society Press.

[9] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[10] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[11] John McLean. The algebra of security. In *IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, California, 1988.

[12] Catherine Meadows. Policies for dynamic upgrading. In *Database Security, IV: Status and Prospects*, pages 241–250. North Holland, 1991.

[13] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1996.

[14] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[15] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[16] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[17] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2003.

[18] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[19] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[20] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer-Verlag.

[21] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[22] Ravi S. Sandhu and Sushil Jajodia. Honest databases that can keep secrets. In *Proceedings of the 14th National Computer Security Conference*, Washington, DC, 1991.

[23] Ian Sutherland, Stanley Perlo, and Rammohan Varadarajan. Deducibility security with dynamic level assignments. In *Proc. 2nd IEEE Computer Security Foundations Workshop*, Franconia, NH, June 1989.

[24] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[25] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[26] Clark Weissman. Security controls in the ADEPT-50 timesharing system. In *AFIPS Conference Proceedings*, volume 35, pages 119–133, 1969.

[27] John P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *IEEE Symposium on Security and Privacy*, pages 23–30, Oakland, California, 1987.

[28] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th Symposium on Logic in Computer Science*, Santa Barbara, June 2000.

[29] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 214–227, San Antonio, TX, January 1999.

[30] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

[31] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.

[32] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. Technical Report 2004–1924, Cornell University Computing and Information Science, 2004.

A Subject Reduction Proof

As described in Section 4.4, the noninterference result for λ_{DSec} is proved by extending the language to a new language λ^2_{DSec} that includes the special bracket construct. Then the subject reduction property for λ^2_{DSec} implies the noninterference property for λ_{DSec}. The appendix details the syntax and semantic extensions of λ^2_{DSec} and proves the key subject reduction theorem.

A.1 Syntax extensions

The syntax extensions of λ^2_{DSec} include the bracket constructs and a new value `void` that can have any type. A λ^2_{DSec} memory encodes two λ_{DSec} memories, which may have distinct domains. The bindings of the form $m^\tau \mapsto (v \mid \texttt{void})$ and $m^\tau \mapsto (\texttt{void} \mid v)$ represent situations where m^τ is bound within only one of the two λ_{DSec} memories.

$$\begin{array}{rcl} \ell & ::= & \ldots \mid (\ell \mid \ell) \\ v & ::= & \ldots \mid (v \mid v) \mid \texttt{void} \\ e & ::= & \ldots \mid (e \mid e) \end{array}$$

The bracket constructs cannot be nested, so the subterms of a bracket construct must be λ_{DSec} terms or `void`. Given a λ^2_{DSec} expression e, let $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ represent the two λ_{DSec} terms that e encodes. The projection functions satisfy $\lfloor (e_1 \mid e_2) \rfloor_i = e_i$ and are homomorphisms on other expression forms. In addition, $(e_1 \mid e_2)[v/x]$, the capture-free substitution of v for x in $(e_1 \mid e_2)$, must use the corresponding projection of v in each branch: $(e_1 \mid e_2)[v/x] = (e_1[\lfloor v \rfloor_1/x] \mid e_2[\lfloor v \rfloor_2/x])$.

In λ^2_{DSec}, labels can be bracket constructs, and types may contain bracketed labels. Thus, the projection operation can be applied to labels, types, type assignments, and label constraints. Similarly, the projection functions are homomorphisms on these typing constructs. For example, $\lfloor \texttt{int}_{(L \mid H)} \rfloor_1 = \texttt{int}_L$, and $\lfloor x : \tau, y : \tau' \rfloor_1 = x : \lfloor \tau \rfloor_1, y : \lfloor \tau' \rfloor_1$.

The following relabeling rule is added to reason about relabeling relationship between bracketed labels:

$$\frac{\lfloor C \rfloor_1 \vdash \lfloor \ell_1 \rfloor_1 \sqsubseteq \lfloor \ell_2 \rfloor_1 \qquad \lfloor C \rfloor_2 \vdash \lfloor \ell_1 \rfloor_2 \sqsubseteq \lfloor \ell_2 \rfloor_2}{C \vdash \ell_1 \sqsubseteq \ell_2}$$

Since a λ^2_{DSec} term effectively encodes two λ_{DSec} terms, the evaluation of a λ^2_{DSec} term can be projected into two λ_{DSec} evaluations. An evaluation step of a bracket expression $(e_1 \mid e_2)$ is an evaluation step of either e_1 or e_2. and e_1 or e_2 can only access the corresponding projection of the memory. Thus, the configuration of λ^2_{DSec} has an index $i \in \{\bullet, 1, 2\}$ that indicates whether the term to be evaluated is a subterm of a bracket expression, and if so which branch of a bracket the term belongs to. For example, the configuration $\langle e, M \rangle_1$ means that e belongs to the first branch of a bracket, and e can only access the first projection of M. We write "$\langle e, M \rangle$" for "$\langle e, M \rangle_\bullet$", which means e does not belong to any bracket.

A.2 Operational semantics

The operational semantics of λ^2_{DSec} is shown in Figure 6. It is based on the semantics of λ_{DSec} and contains some new evaluation rules (E10–E14) for manipulating bracket constructs. Rules (E2)–(E4) are modified to access the memory projection corresponding to index i. The rest of the rules in Figure 2 are adapted to λ^2_{DSec} by indexing each configuration with i. The following two lemmas state that the operational semantics of λ^2_{DSec} is adequate to encode the execution of two λ_{DSec} terms. Their proof is straightforward.

Lemma A.1 (Soundness). If $\langle e, M \rangle \longmapsto \langle e', M' \rangle$, then $\langle \lfloor e \rfloor_i, \lfloor M \rfloor_i \rangle \longmapsto \langle \lfloor e' \rfloor_i, \lfloor M' \rfloor_i \rangle$ for $i \in \{1, 2\}$.

Lemma A.2 (Completeness). If $\langle \lfloor e \rfloor_i, \lfloor M \rfloor_i \rangle \longmapsto^* \langle v_i, M'_i \rangle$ for $i \in \{1, 2\}$, then there exists a configuration $\langle v, M' \rangle$ such that $\langle e, M \rangle \longmapsto^* \langle v, M' \rangle$.

The type system of λ^2_{DSec} includes all the typing rules in Figure 5 and has two additional rules, one for typing `void`, the other for typing bracket constructs.

[*E2*] $\langle !m^\tau,\, M\rangle_i \longmapsto \langle \mathtt{read}_i\, M(m^\tau),\, M\rangle_i$

[*E3*] $$\frac{m \notin \textit{address-space}(M)}{\langle \mathtt{ref}^\tau v,\, M\rangle_i \longmapsto \langle m^\tau,\, M[m^\tau \mapsto \mathtt{new}_i\, v]\rangle_i}$$

[*E4*] $\langle m^\tau := v,\, M\rangle_i \longmapsto \langle (),\, M[m^\tau \mapsto \mathtt{update}_i\, M(m^\tau)\, v]\rangle_i$

[*E10*] $$\frac{\langle e_i,\, M\rangle_i \longmapsto \langle e_i',\, M'\rangle_i \quad e_j = e_j' \qquad \{i,j\} = \{1,2\}}{\langle (e_1 \mid e_2),\, M\rangle \longmapsto \langle (e_1' \mid e_2'),\, M'\rangle}$$

[*E11*] $\langle (v_1 \mid v_2)v,\, M\rangle \longmapsto \langle (v_1\lfloor v\rfloor_1 \mid v_2\lfloor v\rfloor_2),\, M\rangle$

[*E12*] $\langle (v_1 \mid v_2) := v,\, M\rangle \longmapsto \langle (v_1 := \lfloor v\rfloor_1 \mid v_2 := \lfloor v\rfloor_2),\, M\rangle$

[*E13*] $\langle !(v_1 \mid v_2),\, M\rangle \longmapsto \langle (!v_1 \mid !v_2),\, M\rangle$

[*E14*] $\langle \mathtt{if}\ v_1 \sqsubseteq v_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\, M\rangle \longmapsto \langle (\mathtt{if}\ \lfloor v_1\rfloor_1 \sqsubseteq \lfloor v_2\rfloor_1\ \mathtt{then}\ \lfloor e_1\rfloor_1\ \mathtt{else}\ \lfloor e_2\rfloor_1 \mid \mathtt{if}\ \lfloor v_1\rfloor_2 \sqsubseteq \lfloor v_2\rfloor_2\ \mathtt{then}\ \lfloor e_1\rfloor_2\ \mathtt{else}\ \lfloor e_2\rfloor_2), M\rangle$
if $v_1 = (v \mid v')$ or $v_2 = (v \mid v')$

[*E15*] $\langle v_1 \sqcup v_2,\, M\rangle \longmapsto \langle (\lfloor v_1\rfloor_1 \sqcup \lfloor v_2\rfloor_1 \mid \lfloor v_1\rfloor_2 \sqcup \lfloor v_2\rfloor_2),\, M\rangle$ if $v_1 = (v \mid v')$ or $v_2 = (v \mid v')$

[*E16*] $\langle \mathtt{let}\ (x,y) = ((x = v_1[C],\, v_2 : \tau) \mid (x = v_1'[C'],\, v_2' : \tau'))\ \mathtt{in}\ e,\, M\rangle \longmapsto \langle e[(v_2 \mid v_2')/y][(v_1 \mid v_1')/x],\, M\rangle$

[Auxiliary functions]

$\mathtt{new}_\bullet\, v = v$	$\mathtt{update}_\bullet\, vv' = v'$	$\mathtt{read}_\bullet\, v = v$
$\mathtt{new}_1\, v = (v \mid \mathtt{void})$	$\mathtt{update}_1\, vv' = (v' \mid \lfloor v\rfloor_2)$	$\mathtt{read}_1\, v = \lfloor v\rfloor_1$
$\mathtt{new}_2\, v = (\mathtt{void} \mid v)$	$\mathtt{update}_2\, vv' = (\lfloor v\rfloor_1 \mid v')$	$\mathtt{read}_2\, v = \lfloor v\rfloor_2$

Figure 6: Small-step operational semantics of λ^2_{DSec}

[*VOID*] $\Gamma\,;C\,;\, pc \vdash \mathtt{void} : \tau$

[*BRACKET*] $$\frac{\begin{array}{c}\lfloor\Gamma\rfloor_1\,;\lfloor C\rfloor_1\,;\lfloor pc'\rfloor_1 \vdash e_1 : \lfloor\tau\rfloor_1 \\ \lfloor\Gamma\rfloor_2\,;\lfloor C\rfloor_2\,;\lfloor pc'\rfloor_2 \vdash e_2 : \lfloor\tau\rfloor_2 \\ H \sqcup pc \sqsubseteq pc' \quad H \sqsubseteq \tau\end{array}}{\Gamma\,;C\,;\, pc \vdash (e_1 \mid e_2) : \tau}$$

A.3 Subject reduction

The proof of subject reduction starts with some lemmas about projection and substitution.

Lemma A.3 (Label Projection). If $C \vdash \ell_1 \sqsubseteq \ell_2$, then $\lfloor C\rfloor_i \vdash \lfloor\ell_1\rfloor_i \sqsubseteq \lfloor\ell_2\rfloor_i$ for $i \in \{1,2\}$.

Proof. By induction on the derivation of $C \vdash \ell_1 \sqsubseteq \ell_2$. □

Lemma A.4 (Constraint Reduction). If $\Gamma\,;C, \ell_1 \sqsubseteq \ell_2\,;\, pc \vdash e : \tau$ and $C \vdash \ell_1 \sqsubseteq \ell_2$, then $\Gamma\,;C\,;\, pc \vdash e : \tau$.

Proof. By induction on the derivation of $\Gamma\,;C, \ell_1 \sqsubseteq \ell_2\,;\, pc \vdash e : \tau$. □

Lemma A.5 (Projection). If $\Gamma\,;C\,;\, pc \vdash e : \tau$, then $\lfloor\Gamma\rfloor_i\,;\lfloor C\rfloor_i\,;\lfloor pc\rfloor_i \vdash \lfloor e\rfloor_i : \lfloor\tau\rfloor_i$, for $i \in \{1,2\}$.

Proof. By induction on the derivation of $\Gamma\,;C\,;\, pc \vdash e : \tau$, and using the label projection lemma. □

Lemma A.6 (Store Access). Let i be in $\{\bullet, 1, 2\}$. Suppose $pc \vdash v : \tau$ and $pc \vdash v' : \tau$. In addition, $i \in \{1,2\}$ implies $H \sqsubseteq \tau$. Then $pc \vdash \mathtt{read}_i\, v : \lfloor\tau\rfloor_i$, $pc \vdash \mathtt{new}_i\, v : \tau$ and $pc \vdash \mathtt{update}_i\, vv' : \tau$.

Proof. By the definition of the functions `read`, `new` and `update` in Figure 6, by the projection lemma, and rules (VOID) and (BRACKET). □

Lemma A.7 (Substitution). If $x : \tau', \Gamma\,;C\,;\, pc \vdash e : \tau$, and $\vdash v : \tau'[v/x]$, then $\Gamma[v/x]\,;C[v/x]\,;\, pc[v/x] \vdash e[v/x] : \tau[v/x]$.

Proof. By induction on the derivation of $x : \tau', \Gamma\,;C\,;\, pc \vdash e : \tau$. □

Theorem A.1 (Subject Reduction). Suppose $pc \vdash e : \tau$, memory M is well-typed, $\langle e,\, M\rangle_i \longmapsto \langle e',\, M'\rangle_i$, and $i \in$

$\{1,2\}$ implies $H \sqsubseteq pc$. Then $pc \vdash e' : \tau$, and M' is also well-typed.

Proof. By induction on the derivation of $\langle e,\, M\rangle_i \longmapsto \langle e',\, M'\rangle_i$. Without loss of generality, we assume that the last step of the derivation of $pc \vdash e : \tau$ does not use the rule (SUB). Here we just show eight cases: (E3), (E5), (E6), (E8), (E10), (E11), (E14) and (E16). The rest of evaluation rules are treated similarly.

- Case (E3). e is $\texttt{ref}^{\tau'}\, v$, and τ is $(\tau'\ \texttt{ref})_\perp$. Then e' is $m^{\tau'}$. By (LOC), $pc \vdash e' : (\tau'\ \texttt{ref})_\perp$. By Lemma A.6, $pc \vdash \texttt{new}_i v : \tau'$. Thus, $M[m^{\tau'} \mapsto \texttt{new}_i v]$ is well-typed.

- Case (E5). e is $(\lambda(x:\tau')[C'\,;pc'].\, e')v$. Then $pc \vdash \lambda(x:\tau')[C'\,;pc'].\, e' : ((x:\tau'') \xrightarrow{C''\,;\,pc''} \tau_1)_\ell$, and $pc \vdash v : \tau''$, and $\vdash C''[v/x]$. By rules (APP) and (L-APP), $\tau = \tau_1[v/x] \sqcup \ell$, and $pc \sqsubseteq pc''[v/x]$. By rules (ABS) and (SUB), $x:\tau'\,;C'\,;pc' \vdash e' : \tau_1$, and $\vdash \tau'' \leq \tau'$, $\vdash pc'' \sqsubseteq pc'$, and $C'' \vdash C'$. Therefore, $\vdash C'[v/x]$, and $pc \sqsubseteq pc'[v/x]$. By the substitution lemma, $C'[v/x]\,;pc'[v/x] \vdash e'[v/x] : \tau_1[v/x]$. By Lemma A.4, $pc'[v/x] \vdash e'[v/x] : \tau_1[v/x]$. Since $pc \sqsubseteq pc'[v/x]$ and $\tau_1[v/x] \sqsubseteq \tau$, we have $pc \vdash e'[v/x] : \tau$.

- Case (E6). By rule (IF), $k_1 \sqsubseteq k_2\,; pc \vdash e_1 : \tau$. By Lemma A.4 and $\mathcal{L} \models k_1 \sqsubseteq k_2$, we have $pc \vdash e_1 : \tau$.

- Case (E8). e is $\texttt{let}\ (x,y) = (x = v_1[C],\ v_2 : \tau_2)\ \texttt{in}\ e'$. By rule (UNPACK), $pc \vdash (x = v_1[C],\ v_2 : \tau_2) : ((x:\tau_1)[C] * \tau_2)_\ell$, and $x:\tau_1 \sqcup \ell, y:\tau_2 \sqcup \ell\,; pc \vdash e' : \tau$. By rule (PROD), $pc \vdash v_1 : \tau_1[v_1/x]$, and $pc \vdash v_2[v_1/x] : \tau_2[v_1/x]$, and $\vdash C[v_1/x]$. Using the substitution lemma twice, we get $C[v_1/x]\,; pc \vdash e'[v_1/x][v_2[v_1/x]/y] : \tau[v_1/x][v_2[v_1/x]/y]$. It is easy to show that $e'[v_1/x][v_2[v_1/x]/y] = e'[v_2/y][v_1/x]$. According to rule (UNPACK), $x, y \notin FV(\tau)$. Thus, $\tau[v_1/x][v_2[v_1/x]/y] = \tau$. In addition, we have $\vdash C[v_1/x]$. Therefore, $pc \vdash e[v_1/x][v_2/y] : \tau$.

- Case (E10). e is $(e_1 \mid e_2)$. Without loss of generality, assume $\langle e_1,\, M\rangle_1 \longmapsto \langle e_1',\, M'\rangle_1$ and $e_2 = e_2'$. By rule (BRACKET), $H \sqsubseteq pc$, and $\lfloor pc\rfloor_1 \vdash e_1 : \lfloor\tau\rfloor_1$. $H \sqsubseteq pc$ implies $H \sqsubseteq \lfloor pc\rfloor_1$. By induction, $\lfloor pc\rfloor_1 \vdash e_1' : \lfloor\tau\rfloor_1$, and M' is well-typed. Using rule (BRACKET), we can get $pc \vdash (e_1' \mid e_2') : \tau$.

- Case (E11). e is $(v_1 \mid v_2)v$. By (APP) and (L-APP), $pc \vdash (v_1 \mid v_2) : ((x:\tau') \xrightarrow{C'\,;\,pc'} \tau'')_\ell$, and $pc \vdash v : \tau'$. Then $\tau = \tau''[v/x] \sqcup \ell$. In addition, $pc \sqcup \ell \sqsubseteq pc'$. By (BRACKET), $H \sqsubseteq \ell$, which implies $H \sqsubseteq pc'$. By Lemma A.5, $\lfloor pc\rfloor_i \vdash v_i : ((x : \lfloor\tau'\rfloor_i) \xrightarrow{\lfloor C'\rfloor_i\,;\lfloor pc'\rfloor_i} \lfloor\tau\rfloor_i)_{\lfloor\ell\rfloor_i}$, and $\lfloor pc\rfloor_i \vdash \lfloor v\rfloor_i : \lfloor\tau'\rfloor_i$, which imply $\lfloor pc\rfloor_i \vdash v_i\lfloor v\rfloor_i : \lfloor\tau\rfloor_i$. According to (APP) and (L-APP), a well-typed application expression e_1e_2 can be type-checked with the pc component of the type of e_1 in the typing context. Therefore, $\lfloor pc'\rfloor_i \vdash v_i\lfloor v\rfloor_i : \lfloor\tau\rfloor_i$. Since $H \sqsubseteq pc'$, we can apply (BRACKET) to get $pc \vdash (v_1\lfloor v\rfloor_1 \mid v_2\lfloor v\rfloor_2) : \tau$.

- Case (E14). e is $\texttt{if}\ v_1 \sqsubseteq v_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, and there exists $j \in \{1,2\}$ such that $v_j = (v \mid v')$. Suppose $pc \vdash v_i : \texttt{label}_{\ell_i}$ for $i \in \{1,2\}$. Since v_j is a bracket construct, $H \sqsubseteq \ell_j$. By (IF), both e_1 and e_2 are type-checked with $pc \sqcup \ell_1 \sqcup \ell_2$ in the typing context. Thus, we can get $pc \sqcup \ell_1 \sqcup \ell_2 \vdash e : \tau$. By Lemma A.5, $\lfloor pc \sqcup \ell_1 \sqcup \ell_2\rfloor_i \vdash \lfloor e\rfloor_i : \lfloor\tau\rfloor_i$. $H \sqsubseteq \ell_j$ implies $H \sqsubseteq \lfloor pc \sqcup \ell_1 \sqcup \ell_2\rfloor_i$. Applying (BRACKET), we get $pc \vdash (\lfloor e\rfloor_1 \mid \lfloor e\rfloor_2) : \tau$.

- Case (E16). e is $\texttt{let}\ (x,y) = ((x = v_1[C],\ v_2 : \tau) \mid (x = v_1'[C'],\ v_2' : \tau'))\ \texttt{in}\ e'$. Suppose expression $((x = v_1[C],\ v_2 : \tau) \mid (x = v_1'[C'],\ v_2' : \tau'))$ has type $(x:\tau_1)[C_0] * \tau_2)_\perp$. It is easy to show that $(v_1 \mid v_1')$ and $(v_2 \mid v_2')$ have type τ_1 and τ_2 respectively. Then this case is reduced to case (E8), which is standard.

□