Dynamic Security Labels and Static Information Flow Control

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

Abstract

This paper presents a language in which information flow is securely controlled by a type system, yet the security class 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 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 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 [8] 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., [32, 15, 37, 26, 4, 28]). These type systems are an attractive way to enforce security because they can be shown to enforce *noninterference* [13], a strong, end-to-end security property. For example, when applied to confidentiality, noninterference ensures that confidential information cannot be leaked 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 [22] are needed that can be manipulated and checked at run time. Dynamic information control mechanisms [33, 6, 11, 17, 29, 10] support dynamic labels and use run-time label tests to control information flows. However, these dynamic mechanisms incur large run-time overhead and generally cannot prevent *implicit flows* arising from the control flow paths not taken at run time [7, 19]. Thus, it is desirable to combine dynamic labels and static information flow control: making dynamic labels and run-time label tests explicit in programs and using static program analysis to reason about their security properties.

JFlow [21] and its successor, Jif [24] 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 λ_{DSec} (based on the typed lambda calculus) with first-class label values, dependent security types and run-time label tests. We prove the correctness of this mechanism by showing that any well-typed program of the core language satisfies noninterference, which intuitively means that confidential inputs cannot interfere with outputs observable to attackers. In this paper, attackers are assumed to be *passive* in the sense that they can compromise data confidentiality only by observing program outputs. With this passive attack model, if a program satisfies noninterference, then attackers can learn nothing about confidential inputs of the program. This simple form of noninterference is standard for security-typed languages, although dynamic labels introduce a subtle complexity: whether an input is confidential may not be statically determinable.

Some previous MAC systems have supported dynamic security classes as part of a downgrading mechanism [30]. While downgrading is important, it is useful to treat it as a separate mechanism so that dynamic manipulation of labels does not necessarily destroy noninterference.

This paper is a revised and expanded version of a paper presented at the second international Workshop on Formal Aspects in Security and Trust [39]. Compared to that conference version, this paper includes a complete proof that the λ_{DSec} type system enforces noninterference. Another improvement is that we demonstrate the dynamic label mechanisms of λ_{DSec} can be applied in practice by proposing a corresponding extension to Jif.

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}. Section 5 proves that the λ_{DSec} type system enforces noninterference. Section 6 interprets and extends the dynamic label mechanism of Jif based on the ideas of λ_{DSec}. Section 7 covers related work, and Section 8 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 variable x has label $\ell_{\mathtt{x}}$ and variable y has label $\ell_{\mathtt{y}}$, then the sum x+y is given the label $\ell_{\mathtt{x}} \sqcup \ell_{\mathtt{y}}$.

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 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 $\ell_{\mathtt{y}} \sqsubseteq \ell_{\mathtt{x}}$, 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 the ability to control implicit flows in all possible execution paths. Consider a simple conditional:

$$\mathtt{if\ s <= } 0 \mathtt{\ then\ x := 0\ else\ y := 0}$$

Although there is no direct assignment from s to x or y, this expression may cause implicit flows from s into x and y, since the values of x and y depend on s after evaluating the expression. A standard technique for controlling implicit flows is to introduce a *program-counter label* [7], 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 the value of s, so the *pc* in the then and else clauses will be joined with $\ell_{\mathtt{s}}$, the label of s. 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 assignment to x will be permitted only if *pc* $\sqsubseteq \ell_{\mathtt{x}}$, which ensures $\ell_{\mathtt{s}} \sqsubseteq \ell_{\mathtt{x}}$. Similarly, $\ell_{\mathtt{s}} \sqsubseteq \ell_{\mathtt{y}}$ is also ensured by the static analysis.

Dynamic mechanisms such as the Data Mark Machine [11] are able to control implicit flows by tracking the program counter label *pc* at run time and check the constraint *pc* $\sqsubseteq \ell_{\mathtt{x}}$ or *pc* $\sqsubseteq \ell_{\mathtt{y}}$ depending on which branch is taken. However, the dynamic mechanisms do not check the label constraints required by the control flow path not taken at run time. For example, suppose the value of s is positive, and *pc* $\sqsubseteq \ell_{\mathtt{y}}$ holds while *pc* $\sqsubseteq \ell_{\mathtt{x}}$ does not hold. Then attackers can infer that s is positive from the absence of run-time label test failures.

2.4 Noninterference

In general, the goal of static information flow control is to enforce noninterference, which intuitively means that confidential inputs cannot interfere with outputs observable to attackers. Formally, the security level of attackers is represented by a label L. Then any input with a label H such that $H \not\sqsubseteq L$ is confidential, and any output with a label less than or equal to L is observable to attackers.

Suppose expression e is a program. Then the inputs of e are the values of free variables of e, and the outputs are simply the result of evaluating e. More formally, the inputs of e are represented by an *input map* A, mapping free variables of e to values, and the notation $e[A]$ denotes the expression obtained by substituting every free variable x of e with $A(x)$. Program e satisfies the noninterference property if changing the confidential inputs of e does not affect the outputs observable to attackers, that is, the following statement holds:

For two arbitrary labels L and H and any two input maps A_1 and A_2 of e satisfying

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

Figure 1: Syntax of λ_{DSec}

- $L \not\sqsubseteq H$,
- the label of e is less than or equal to L, and
- $A_1 \approx_H A_2$, which means that for any free variable x of e, if the label of x is not higher than or equal to H, then $A_1(x) = A_2(x)$,

if $e[A_1]$ and $e[A_2]$ are evaluated to v_1 and v_2, then $v_1 = v_2$.

The noninterference property discussed here is *termination insensitive* [28] because $e[A_1]$ and $e[A_2]$ are required to generate the same result only if both evaluations terminate. In this work, we do not attempt to deal with termination and timing channels. Control of these channels is largely an orthogonal problem. In average, termination channels can leak at most one bit per run, so they have often been considered acceptable (e.g., [8, 32]). Some recent work [1, 27, 38] partially addresses the control of timing channels.

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 $\mathtt{ML}_0^\Pi(C)$ [36].

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 $\perp$), 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 bottom element of $\mathcal{L}$ is $\perp$. Any non-trivial label lattice contains at least two points L and H 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, references, functions and products.

The function type $(x:\tau_1) \xrightarrow{C\,;\,pc} \tau_2$ 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 $\Pi x\!:\!\tau_1.\tau_2$ [20].

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 $\Sigma x\!:\!\tau_1.\tau_2$ [20].

In λ_{DSec}, values include variables x, integers n, constant labels k, the unit value $()$, typed memory locations m^τ, functions $\lambda(x\!:\!\tau)[C\,;pc].\,e$ and pairs $(x\!=\!v_1[C],\ v_2\!:\!\tau)$. A function $\lambda(x\!:\!\tau)[C\,;pc].\,e$ has one argument x with type τ, and the components C and *pc* have the same meanings as those in function types. For simplicity, C can be omitted if it is empty, and the *pc* component can be omitted if e has no side effects. 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 $x \sqsubseteq L$ (which implies v_1 is a label value), 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 $\texttt{let}\ (x,y)\!=\!e_1\ \texttt{in}\ e_2$.

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, substitutes the first element for x and the second for y, and then evaluates e_2.

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. For example, suppose $\lambda f(x:\tau)[C\,;pc].\,e$ is a recursive function (f may appear in e) with type τ_f. Then we can encode the recursive function using the following λ_{DSec} code:

$$\lambda(x\!:\!\tau)[C\,;pc].\,((\lambda(y\!:\!\texttt{unit})[C\,;pc].\,!m^{\tau_f} x)\ (m^{\tau_f} := \lambda(x\!:\!\tau)[C\,;pc].\,e[!m^{\tau_f}/f]))$$

where $e[!m/f]$ is the expression obtained by substituting $!m^{\tau_f}$ for f in e.

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 $loc(e)$ represent the set of memory locations appearing in e. A memory M is well-formed if every address m appears at most once in $dom(M)$, and for any m^τ in $dom(M)$, $loc(M(m^\tau)) \subseteq dom(M)$, where $M(m^\tau)$ denotes the value of location m^τ in M. The configuration $\langle e,\ M\rangle$ is well-formed if M is well-formed, $loc(e) \subseteq dom(M)$, 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

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

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

$$[E3]\quad \frac{m = \mathit{newloc}(M)}{\langle \mathtt{ref}^\tau v,\, M\rangle \longmapsto \langle m^\tau,\, M[m^\tau \mapsto v]\rangle}$$

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

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

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

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

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

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

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

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

of λ_{DSec} guarantees that a well-typed expression can only be evaluated to another well-typed and thus well-formed expression.

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 $\mathit{address\text{-}space}(M)$ represents the set of location names in M, that is, $\{m \mid \exists\tau\ s.t.\ m^\tau \in \mathit{dom}(M)\}$; the allocator $\mathit{newloc}(M)$ deterministically generates a fresh memory location m such that $m \notin \mathit{address\text{-}space}(M)$, and $\mathit{newloc}(M') = m$ if $\mathit{address\text{-}space}(M') = \mathit{address\text{-}space}(M)$. In rule (E8), v_2 may mention x, so substituting v_2 for y in e is performed before substituting v_1 for x. For simplicity, the variable name in the product value matches x so that no variable renaming (alpha conversion) is needed when substituting v_1 and v_2 for x and y in e. In rule (E9), E represents an evaluation context, a term with a single hole (denoted by $[\cdot]$) 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 $(x : \texttt{label}_\bot) * (\texttt{int}_x\ \texttt{ref})_\bot$, where x is a dynamic label consistent with the access control policy of the file, and the reference component of type $(\texttt{int}_x\ \texttt{ref})_\bot$ stores the contents of the file and can be viewed as modeling the physical address of the file on a storage device. Thus storing an integer of type $\texttt{int}_H$ 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:

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

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 implicitly declassified, which provides stronger security assurance than an ordinary file system. This is done by replacing the old memory reference component in the value of w with a new memory reference storing the initial value 0.

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

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* [9] 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 [9].

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:

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

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:

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

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

This section describes the type system of λ_{DSec}, which is designed to check the label constraints that enforce secure information flow.

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 $y := z$ 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.

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 $\tau_1 \le \tau_2$. 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

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

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

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

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

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

Figure 3: Relabeling rules

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

$$[S2]\quad \frac{\begin{array}{c} C \vdash \tau_2 \leq \tau_1 \quad C \vdash \tau_1' \leq \tau_2' \\ C \vdash pc_2 \sqsubseteq pc_1 \quad C, C_2 \vdash C_1 \end{array}}{C \vdash (x\!:\!\tau_1) \xrightarrow{C_1\,;\,pc_1} \tau_1' \leq (x\!:\!\tau_2) \xrightarrow{C_2\,;\,pc_2} \tau_2'}$$

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

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

Figure 4: Subtyping rules

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 $\tau = (x : \tau_1) \xrightarrow{C_1\,;\,pc_1} \tau_1'$ is a subtype of $\tau' = (x : \tau_2) \xrightarrow{C_2\,;\,pc_2} \tau_2'$. 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 $pc_2 \sqsubseteq pc_1$ 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.

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

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

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

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

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

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

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

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

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

This assumption also applies to the typing rules.

Rule (S4) is used to determine the subtyping on security types. The premise $C \vdash \beta_1 \le \beta_2$ 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 any well-typed program satisfies the noninterference property discussed in Section 2. 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 $0 \le i \le n$, 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

$0 \le i \le n$, 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 "$\Gamma\,; C\,; pc$" is well-formed if Γ is well-formed, and pc and all the constraints in C are well-formed with respect to Γ.

The typing assertion $\Gamma\,; C\,;\, pc \vdash e : \tau$ 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 $\Gamma\,; C\,;\, pc \vdash e : \tau$ is well-formed if $\Gamma\,; C\,; pc$ 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 $FV(\tau) = \emptyset$, where $FV(\tau)$ 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 $C \vdash pc \sqsubseteq \tau$. 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 $C \vdash pc \sqsubseteq \tau$ is required to prevent illegal implicit flows. In addition, the premise $C \vdash pc \sqcup \ell \sqsubseteq \tau$ 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 pc', so the function can only be called at places where C' is satisfied and the *pc* label is not more restrictive than 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, as stated in the following two theorems. Theorem 4.1 is an instantiation of Theorem 5.1, which is proved in Section 5.

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 M is a well-typed memory. If $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$, then M' is well-typed, and $pc \vdash e' : \tau$.

Theorem 4.2 (Progress). If $pc \vdash e : \tau$, and $FV(e) = \emptyset$, 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 $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$ and $FV(e') = \emptyset$.

Proof. By induction on the derivation of $pc \vdash e : \tau$. The base cases are cases (INT), (UNIT), (LABEL), (LOC), (ABS), (PROD), in which e is a value.

- Case (JOIN). In this case, e is $\ell_1 \sqcup \ell_2$. If ℓ_1 is not a value, then $\langle \ell_1,\, M\rangle \longmapsto \langle \ell_1',\, M\rangle$ by induction, and $\langle e,\, M\rangle \longmapsto \langle \ell_1' \sqcup \ell_2,\, M\rangle$ by rule (E9). If ℓ_1 is a value, and ℓ_2 is not a value, then $\langle e,\, M\rangle \longmapsto \langle \ell_1 \sqcup \ell_2',\, M\rangle$ by the same argument. Otherwise, ℓ_1 and ℓ_2 are both values, then $\langle e,\, M\rangle \longmapsto \langle k,\, M\rangle$ by rule (E1), where $k = \ell_1 \sqcup \ell_2$.
- Case (VAR). Since $FV(e) = \emptyset$, this case cannot occur.
- Case (REF). e is $\texttt{ref}^\tau\ e_1$. If e_1 is not a value, then $\langle e_1,\, M\rangle \longmapsto \langle e_1',\, M'\rangle$ by induction, and $\langle \texttt{ref}^\tau\ e_1,\, M\rangle \longmapsto \langle \texttt{ref}^\tau\ e_1',\, M'\rangle$. If e_1 is a value v, then $\langle \texttt{ref}^\tau\ e_1,\, M\rangle \longmapsto \langle m^\tau,\, M[m^\tau \mapsto v]\rangle$ by rule (E3).
- Case (DEREF). By induction and rule (E2).
- Case (ASSIGN). By induction and rule (E4).
- Cases (L-APP) and (APP). e is $e_1 e_2$. If e_1 or e_2 is not a value, then $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$ by induction and (E9). Otherwise, e_1 is $\lambda(x : \tau)[C\,;pc].\, e_1'$, and e_2 is v. By rule (E5), $\langle e,\, M\rangle \longmapsto \langle e_1'[v/x],\, M\rangle$. Since $FV(e_1') = FV(e_1) \cup \{x\} = \{x\}$, we have $FV(e_1'[v/x]) = \emptyset$.
- Case (UNPACK). By induction and rule (E8).
- Case (IF). By induction and rules (E6) and (E7).
- Case (SUB). By induction.

□

5 Noninterference

This section proves that any well-typed program in λ_{DSec} satisfies the noninterference property as discussed in Section 2. Let $\longmapsto^*$ denote the transitive closure of the $\longmapsto$ relationship. The following definitions and theorem formalize the claim that the type system of λ_{DSec} enforces noninterference. For simplicity, we only consider that results are integers because they can be compared outside the context of λ_{DSec}.

Definition 5.1 (Well-typed input). An input map A is well-typed with respect to Γ, written $\Gamma \vdash A$, if for any x in $dom(\Gamma)$, we have $\vdash A(x) : \Gamma(x)[A]$, where $\Gamma(x)[A]$ represents the type obtained by substituting every free variable y in $\Gamma(x)$ with $A(y)$.

Definition 5.2 (Input low-equivalence). Two input maps A_1 and A_2 are equivalent with respect to Γ and label H, written as $\Gamma \vdash A_1 \approx_H A_2$, if $\Gamma \vdash A_1, A_2$, and for any x in $dom(\Gamma)$, $H \not\sqsubseteq \Gamma(x)$ implies $A_1(x) = A_2(x)$.

Noninterference Theorem. Suppose $L \not\sqsubseteq H$, and $\Gamma \vdash e : \mathtt{int}_L$. Given two input maps A_1 and A_2 such that $\Gamma \vdash A_1 \approx_H A_2$, if $\langle e[A_i],\, M\rangle \longmapsto^* \langle v_i,\, M'_i\rangle$ 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 [26]. Suppose expression e has only one free variable x. 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. We can show that two λ_{DSec} terms only differ at the confidential part if the two terms can be encoded by a well-typed λ^2_{DSec} term. Therefore, proving the noninterference theorem of λ_{DSec} can be reduced to proving the 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.

The rest of this section details the syntax and semantic extensions of λ^2_{DSec} and proves the key subject reduction theorem of λ^2_{DSec} and the noninterference theorem of λ_{DSec}.

5.1 Syntax extensions

The λ^2_{DSec} language extends λ_{DSec} with the bracket constructs and a new value void that can have any type:

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

where the ellipses represent the terms also belonging to λ_{DSec}. The bracket constructs cannot be nested, so the subterms of a bracket construct must be λ_{DSec} terms or void. A λ^2_{DSec} memory encodes two λ_{DSec} memories, which may have distinct domains. The bindings of the form $m^\tau \mapsto (v \mid \mathtt{void})$ and $m^\tau \mapsto (\mathtt{void} \mid v)$ represent situations where m^τ is bound within only one of the two λ_{DSec} memories.

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 $\lfloor (e_1 | e_2) \rfloor_i = e_i$ 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, $\lfloor \mathtt{int}_{(L \mid H)} \rfloor_1 = \mathtt{int}_L$, 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:

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

5.2 Operational semantics

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 $(e_1 \,|\, e_2)$ 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.

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} satisfies (A1) and (A2), and is adequate to encode the execution of two λ_{DSec} terms.

Lemma 5.1 (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 $i \in \{1, 2\}$.

Proof. By inspection of the evaluation rules. □

Lemma 5.2 (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 $\langle e,\, M\rangle \longmapsto^* \langle v,\, M'\rangle$.

Proof. First, $\langle e,\, M\rangle$ cannot admit an infinite evaluation sequence. Rules (E11)–(E16) can only be applied for finite times because each of these rules moves some pair constructor strictly closer to the term's root. These rules are the only rules that leave both projections of a configuration unchanged. Therefore, by Lemma 5.1, an infinite evaluation sequence of $\langle e,\, M\rangle$ implies that for some $i \in \{1, 2\}$, $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i\rangle$ admits an infinite evaluation sequence, which contradicts $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i\rangle \longmapsto^* \langle v_i,\, M'_i\rangle$, since the operational semantics of λ_{DSec} is deterministic.

By induction on the structure of e, we can prove that if $\langle e,\, M\rangle$ is stuck, then $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i\rangle$ for some $i \in \{1, 2\}$ is also stuck. Therefore, $\langle e,\, M\rangle$ cannot be stuck, and then it must terminate normally.

□

5.3 Typing and subject reduction

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.

$$[\mathit{VOID}] \qquad \Gamma\,; C\,;\, pc \vdash \texttt{void} : \tau$$

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

Before proving the λ^2_{DSec} type system satisfies the subject reduction property, we first prove some lemmas about projection and substitution.

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

[E3] $\quad \dfrac{m = \mathit{newloc}(M)}{\langle \texttt{ref}^\tau v,\, M\rangle_i \longmapsto \langle m^\tau,\, M[m^\tau \mapsto \texttt{new}_i\, v]\rangle_i}$

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

[E10] $\quad \dfrac{\langle e_i,\, M\rangle_i \longmapsto \langle e'_i,\, M'\rangle_i \quad e_j = e'_j \qquad \{i,j\} = \{1,2\}}{\langle (e_1 \mid e_2),\, M\rangle \longmapsto \langle (e'_1 \mid e'_2),\, M'\rangle}$

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

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

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

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

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

[E16] $\quad \langle \texttt{let}\ (x,y) = ((x = v_1[C],\ v_2 : \tau) \mid (x = v'_1[C'],\ v'_2 : \tau'))\ \texttt{in}\ e,\, M\rangle \longmapsto \langle e[(v_2 \mid v'_2)/y][(v_1 \mid v'_1)/x],\, M\rangle$

[Auxiliary functions]

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

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

Lemma 5.3 (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 5.4 (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 5.5 (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 5.6 (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 $pc \vdash \texttt{read}_i\, v : \lfloor\tau\rfloor_i$, $pc \vdash \texttt{new}_i\, v : \tau$ and $pc \vdash \texttt{update}_i\, vv' : \tau$.

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

Lemma 5.7 (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 5.1 (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). Suppose the derivation of $pc \vdash e : \tau$ ends with using (SUB). Then there exists τ' such that $pc \vdash e : \tau'$, and $\tau' \leq \tau$, and the derivation of $pc \vdash e : \tau'$ does not end with using (SUB). If we can show $pc \vdash e : \tau'$, which satisfies the assumption, then $pc \vdash e : \tau$ by (SUB). Therefore, the assumption does not lose generality.

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 5.6, $pc \vdash \texttt{new}_i v : \tau'$. 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 5.4, $pc'[v/x] \vdash e'[v/x] : \tau_1[v/x]$. 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 5.4 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 straightforward 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 5.5, $\lfloor pc \rfloor_i \vdash v_i : ((x : \lfloor \tau' \rfloor_i) \xrightarrow{\lfloor C' \rfloor_i ; \lfloor pc' \rfloor_i} \lfloor \tau \rfloor_i)_{\lfloor \ell \rfloor_i}$, 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 $e_1 e_2$ 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 5.5, $\lfloor pc \sqcup \ell_1 \sqcup \ell_2 \rfloor_i \vdash \lfloor e \rfloor_i : \lfloor \tau \rfloor_i$. $H \sqsubseteq \ell_j$ 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 let $(x, y) = ((x = v_1[C],\ v_2 : \tau) \mid (x = v_1'[C'],\ v_2' : \tau'))$ 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.

□

5.4 Noninterference proof

Theorem 5.2 (Noninterference). Suppose $L \not\sqsubseteq H$, and $\Gamma \vdash e : \texttt{int}_L$. Given two input maps A_1 and A_2 such that $\Gamma \vdash A_1 \approx_H A_2$, if if $\langle e[A_i],\ M\rangle \longmapsto^* \langle v_i,\ M_i'\rangle$ for $i \in \{1, 2\}$, then $v_1 = v_2$.

Proof. First, we incorporate A_1 and A_2 into a λ^2_{DSec} input map A such that for any x in *dom*(Γ), $A(x) = A_1(x)$ if $A_1(x) = A_2(x)$, and $A(x) = (A_1(x) \mid A_2(x))$ if otherwise. Since $\Gamma \vdash A_1 \approx_H A_2$, $A(x)$ is a bracket construct only if $H \sqsubseteq \Gamma(x)[A_1]$ and $H \sqsubseteq \Gamma(x)[A_2]$, or equivalently, $H \sqsubseteq \Gamma(x)[A]$. Therefore, A is a well-typed input map with respect to Γ. By Lemma 5.7, $\vdash e[A] : \texttt{int}_L$.

Because $\langle e[A_i],\ M\rangle \longmapsto^* \langle v_i,\ M_i'\rangle$ and $e[A_i] = \lfloor e[A]\rfloor_i$ for $i \in \{1, 2\}$, $\langle e[(v_1 \mid v_2)/x],\ M\rangle \longmapsto^* \langle v,\ M'\rangle$ by Lemma 5.2. Moreover, $\vdash v : \texttt{int}_L$ by Theorem 5.1. Thus, v is not a bracket value, and $\lfloor v\rfloor_1 = \lfloor v\rfloor_2$. By Lemma 5.1, $\langle e[A_i],\ M\rangle \longmapsto^* \langle\lfloor v\rfloor_i,\ \lfloor M'\rfloor_i\rangle$ for $i \in \{1, 2\}$. Since the operational semantics of λ_{DSec} is deterministic, we have $v_1 = \lfloor v\rfloor_1$ and $v_2 = \lfloor v\rfloor_2$, which imply $v_1 = v_2$. □

6 Dynamic labels in practice

The simplicity of λ_{DSec} helps proving the correctness of its dynamic label mechanism, but makes λ_{DSec} impractical to use. This section shows that the dynamic label mechanism of λ_{DSec} can be applied to a practical programming language such as Jif. First, we show that the existing dynamic label mechanism of Jif can be interpreted using λ_{DSec}. Second, we propose an extension to the dynamic label mechanism of Jif based on λ_{DSec}.

6.1 Dynamic labels in Jif

Jif [24] is the only implemented security-typed language supporting dynamic labels. Jif extends the Java language [14] with static information flow control. Jif aims to provide a usable programming model, in which the dynamic label mechanism plays an important role.

In Jif, labels can also be used as first-class values, and a variable of type `label` may be used as a label for other values. Jif provides the `switch label` statement for run-time label tests. For example, the following code uses the `switch label` statement to send a value through a communication channel with a dynamic label:

(A)
```
final label{L} x;
Channel{*x} c;
int{H} y;
switch label(y) {
  case (int{*x} z) c.send(z);
  else throw new UnsafeTransfer();
}
```

The `send` operation is secure only if `x` is a high-security label, which has to be determined at run time. The notation `*x` refers to the label value of `x`; it can be used as a label only if `x` is declared as a `final` variable, to prevent assignments from changing the meaning of types that mention it. In the example, the `switch label` statement executes the first of the cases whose associated label is at least as restrictive as that of `y`. The value of `y` is assigned to the corresponding variable (for example, `z`). In this example, the first case will be executed only if $H \sqsubseteq *\texttt{x}$, guaranteeing that `c` is a high-security channel.

In general, the statement $\texttt{switch label}(e)\{\texttt{case}\ (\texttt{int}\{\ell\}\ x)\ S_1;\ \texttt{else}\ S_2\}$ can be encoded as the following pseudo-code in λ_{DSec}:

$$\texttt{if}\ \ell_e \sqsubseteq \ell\ \texttt{then}\ (\lambda(x\!:\!\texttt{int}_\ell)[pc].\, S_1)\, e\ \texttt{else}\ S_2$$

where ℓ_e represents the label of e, and *pc* is an upper bound to the labels of the effects of S_1. By rule (IF), $\ell_1 \sqcup \ell_2 \sqsubseteq pc$ needs to hold, where ℓ_1 and ℓ_2 are the labels of ℓ_e and ℓ, respectively. Indeed, the type system of Jif ensures $\ell_1 \sqcup \ell_2 \sqsubseteq pc$.

In Jif, labels are specified using the *decentralized label model* [23]. These labels may explicitly mention principals. For example, a value with type `int{Alice:Bob}` is an integer owned by principal `Alice` and readable by `Alice` and `Bob`. Like labels, principals may also be used as first-class values at run time. The statement `actsFor(p1, p2)`S executes the statement S if the principal `p1` can act for the principal `p2`. This acts-for relationship between `p1` and `p2` is equivalent to `{p2:}` $\sqsubseteq$ `{p1:}`. Thus the `actsFor` statement essentially implements a run-time label test, and can be encoded as:

$$\texttt{if}\ \texttt{\{p2:\}} \sqsubseteq \texttt{\{p1:\}}\ \texttt{then}\ S\ \texttt{else}\ ()$$

The Jif type system checks S with a program counter label *pc* such that $\ell_1 \sqcup \ell_2 \sqsubseteq pc$, where ℓ_1 and ℓ_2 are the labels of `p1` and `p2`, respectively. This is consistent with the type system of λ_{DSec}.

6.2 The Jif-DX language

The original Jif dynamic label mechanism appears to be sound but has several limitations. First, label checking of the clauses of a `switch label` statement does not fully exploit the label constraint enforced by the run-time test. Second, Jif supports only one kind of statically specified label constraints: `actsFor` constraints, which give information about principals but are not as powerful as general label constraints. Third, in Jif only variables or fields of the enclosing class declaration can be used as dynamic labels, but in practice other expressions may be useful in dynamic labels.

These limitations of Jif make it difficult or awkward to write some applications that need to manipulate dynamic labels. Therefore, we propose the Jif-DX language, which extends Jif with a better dynamic label mechanism, including the label-test statement, method and field label constraints, and more general label expressions [1]. These new language features are based on the constructs of λ_{DSec}. In particular, the label-test statement resembles the label-test expression of λ_{DSec}; a method label constraint corresponds to the label constraint component of a lambda term; a field label constraint corresponds to the label constraint component in a pair value.

6.2.1 The label-test statement

Jif-DX provides the label-test statement, which is a more flexible way to implement run-time label checks than the `switch label` statement. The label-test statement resembles the conditional label-test expression of λ_{DSec}, except that the conditional branches are statements instead of expressions: "$\texttt{if}\ (\ell_1\texttt{<=}\ell_2)\ S_1\ \texttt{else}\ S_2$". Intuitively, S_1 is executed if $\ell_1 \sqsubseteq \ell_2$ is true at run time; otherwise, S_2 is executed. As in λ_{DSec}, $\ell_1 \sqsubseteq \ell_2$ can be assumed to hold when type-checking S_1.

Both the `switch label` statement and the `actsFor` statement in Jif can be encoded with the label-test statement. For example, the statement "`actsFor(p1, p2)` S" is equivalent to "`if ({p2:} <= {p1:})` S".

[1]Some of the proposed features have been incorporated into Jif version 3.0.

6.2.2 Method label constraints

Jif-DX allows general label constraints to be specified in method signatures, whereas Jif only provides `actsFor` constraints. The following example shows a use of a label constraint on a method:

(B)
```
class Key[principal p] {
    int{} encrypt(label{} lb, int{*lb} x) where {*lb} <= {p:}   { ...  }
}
```

The class `Key[principal p]` represents a key belonging to principal `p`. The `encrypt` method takes in a label `lb` and an integer `x` labeled with `{*lb}`, and attempts to encrypt `x` with the key of principal `p` and return the encrypted result as a public integer. This method should only encrypt the data owned by principal `p`, because the result can be decrypted by `p`. This requirement is captured by the method label constraint `{*lb}` $\sqsubseteq$ `{p:}`. The compiler ensures that the constraint is satisfied wherever this method is called.

Another way to write this code would be to insert a run-time check in the method body and make the method throw an exception if `{*lb}` $\sqsubseteq$ `{p:}` is not satisfied at run time. This code would incur some unnecessary run-time label checks, and the caller would have to handle the exception somehow. Indeed, one advantage of the method label constraint is its ability to exploit information available at the caller side to reduce the number of run-time checks. For example, in the following Jif-DX code the compiler can determine that the method constraint is satisfied without a run-time check:

(C)
```
Key[Alice]{} k;
int{Alice:Bob} x;
k.encrypt({Alice:Bob}, x);
```

6.2.3 Field label constraints

In Jif-DX, label constraints can also be specified on class fields of type `label`. The compiler ensures that the field label constraints of a class are satisfied whenever a new instance of the class is created. All fields appearing in a label constraint must be final, so field label constraints that are satisfied when an object is created will hold for the lifetime of the object.

Like method label constraints, field label constraints can be used to reduce the number of run-time label checks. For example, sending an integer through a multilevel communication channel with label ℓ requires sending the exact label of the integer through the channel. The natural way to implement it is to wrap the integer and its label in an object of the `Labeled` class and send the object through the channel.

(D)
```
class Labeled {
    public final label{ℓ} lb;
    public int{*lb} content;
    public Labeled(label{ℓ} x, int{*x} y) { lb = x;  content = y; }
}
```

The label of field `lb` is ℓ, ensuring that `lb` itself can be sent through the channel. But the label of field `content` is dynamic, and the constraint `{*lb}` $\sqsubseteq \ell$ needs to hold for field `content` to be sent safely through the channel. This constraint can be enforced by a run-time label check, but it can also be enforced statically by specifying a field label constraint `{*lb}` $\sqsubseteq \ell$, as in the `UBLabeled` (“`UB`” stands for upper bound) class. Sending a `UBLabeled` object through a channel with label ℓ is always safe.

(E)
```
class UBLabeled {
    public final label{ℓ} lb where {*lb} <= ℓ;
    public int{*lb} content;
    public UBLabeled(label{ℓ} x, int{*x} y) where {*x} <= ℓ {
```

```
        lb = x;  content = y;
    }
}
```

6.2.4 Path-expression labels

Consider the `Labeled` class again, and suppose `o` is a `Labeled` object. Then what is the type of `o.content`? According to the `Labeled` class, the precise type would be `int{*o.lb}`, which cannot be expressed in Jif because Jif does not allow *path expressions* such as `o.lb` to appear in labels.

In Jif-DX, a path expression with the type `label` can be used in label expressions as long as all the identifiers in the path expression are final, ensuring that the path expression always has the same value. For example, if `o` is a final variable, then `{*o.lb}` is a legitimate label, and the following code can be used to access `o.content` while preserving its precise type.

```
(F)  int{*o.lb} y = o.content;
```

If `o` were not a final variable, then `o.content` would not be well-typed in Jif-DX. But there is an easy workaround: assign `o` to a final variable `fo` and access the `content` field by `fo.content`, which has a well-formed type `int{*fo.lb}`. This workaround is like unpacking a pair value in λ_{DSec}.

6.2.5 Example: bounded dynamic labeling

In this section, we show how to use the new dynamic label constructs in Jif-DX to implement a MAC mechanism, which would be much harder and unintuitive to implement in Jif. The MAC mechanism in the MITRE CMW system [34] associates two labels with each object: a *floating label* and a fixed *mandatory label*. The floating label is updated accordingly when the content of the object is updated, but is bounded by the fixed mandatory label in order to prevent the covert channel caused by label updates. The doubly labeled object can be represented by a `UBLabeled` (see code fragment E) object in Jif-DX, and the policy that the floating label be bounded by the mandatory label is represented by the field constraint `{*lb}` $\sqsubseteq \ell$, where `{*lb}` is the floating label, and ℓ is the mandatory label.

The following code shows how to update the label and access the content of a `UBLabeled` object. Simple as it is, this example demonstrates several subtle issues related to manipulating dynamic labels.

```
(G)  UBLabeled o;
     final label{} x, y;
     int{*x} data;
     ...
(1)  if ({*x} <= ℓ) o = new UBLabeled(x, data);
     final UBLabeled{} fo = o;
(2)  if ({*fo.lb} <= {*y})
         if ({*y} <= ℓ) o = new UBLabeled(y, fo.content);

(3)  int{ℓ} output = fo.content;
     int{Alice:} output2;
(4)  if ({*fo.lb} <= {Alice:}) output2 = fo.content;
```

The first label-test statement (1) attempts to update the content of `o`, and the constraint `{*x} <=` ℓ guarantees the label of the new value is bounded by the mandatory label ℓ. The constructor call `new UBLabeled(x, data)` is well-typed because of the constraint `{*x}` $\sqsubseteq \ell$ enforced by the label test.

The second label-test statement (2) attempts to just update the label field of `o` to `y`. The first test `{*fo.lb} <= {*y}` is necessary for `new UBLabeled(y, fo.content)` to be well-typed, because the type of `fo.content` (`int{*fo.lb}`) must be a subtype of `int{*y}`. Essentially, the constraint prevents downgrading the label of the object content. Furthermore, this example shows that the immutability requirement for label fields is not a fundamental limitation because adding a level of indirection makes it possible to update `o.lb` even though the field `lb` is final.

The last two statements (3,4) attempt to access `o.content`. The assignment to `output` is well-typed because of the field label constraint `{*fo.lb}` $\sqsubseteq \ell$. The assignment to `output2` might appear secure because a label test is used to ensure the label of `output2` is at least as restrictive as the label of `fo.content`. However, there is an implicit flow from `fo.lb` to `output2` in the label-test statement. The implicit flow is legal only if $\ell \sqsubseteq$ `{Alice:}`, which prevents a possible covert channel caused by dynamic labeling.

7 Related Work

Dynamic information flow control mechanisms [33, 34] 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 [18, 30, 12] 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 [8]; later work phrased the analysis as type checking (e.g., [25]). Noninterference was later developed as a more semantic characterization of security [13], followed by many extensions. Volpano, Smith and Irvine [32] 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 [15] proved the noninterference theorem for the SLam calculus, a purely functional language. Zdancewic and Myers [37] investigated a secure calculus with first-class continuations and references. Pottier and Simonet [26] 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 [28, 40].

One problem with type-based static information flow analyses is that they tend to be conservative and may identify information flows that do not exist. For example, consider the following code:

$$\texttt{if s <= } 0 \texttt{ then x} := 0 \texttt{ else x} := 0$$

in which `x` does not depend on `s`, but most security type systems still ensure $\ell_{\mathtt{s}} \sqsubseteq \ell_{\mathtt{x}}$. Some recent work [2, 16] partially addresses this problem by using flow-sensitive static analyses.

The Jif language [21, 24] 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 [5] proved a noninterference result for a Java-like language with simple access control primitives. Unlike in λ_{DSec}, run-time access control in their language is separate from information flow control in the sense that the result of a run-time access check does not affect the security of any information flow in a program.

Concurrent to our work, Tse and Zdancewic proved a noninterference result for a security-typed lambda calculus ($\lambda_{\mathtt{RP}}$) with run-time principals [31]. Run-time principals are closely related to dynamic labels, as labels are composed of principals in the decentralized label model of Jif. However, λ_{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. In addition, support for references makes λ_{DSec} more powerful than λ_{RP} computationally. The λ_{RP} type system uses singleton types (types containing only one value [3]) for relating type information to term-level constructs. 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 proof in any substantial way. In general, we feel that the choice between dependent types and singletons is a matter of taste.

Other work [36, 35] 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.

8 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 is proved to enforce noninterference. Based on the dynamic label mechanism of λ_{DSec}, we propose an extension to Jif, making it easier to write programs manipulating dynamic labels efficiently.

An important direction for future work is to investigate the interaction between dynamic labels and parametric polymorphism.

Acknowledgements

The authors would like to thank Greg Morrisett, Steve Zdancewic and Amal Ahmed for their insightful suggestions. Many thanks also to Steve Chong, Nate Nystrom, Michael Clarkson, and the anonymous reviewers, who all provided useful feedbacks on earlier drafts of this paper.

This work was supported by the Department of the Navy, Office of Naval Research, under ONR Grant N00014-01-1-0968. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect views of the Office of Naval Research. This work was also supported by the National Science Foundation under grants 0208642, 0133302, and 0430161, and by an Alfred P. Sloan Research Fellowship.

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] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *The Eleventh International Symposium on Static Analysis Proceedings*, pages 100–115, Verona, Italy, 2004.

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

[4] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. 15th IEEE Computer Security Foundations Workshop*, June 2002.

[5] 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.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: mathematical foundations and model. Technical Report M74-244, MITRE Corp., Bedford, MA, 1973.

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

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

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

[10] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[11] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[12] 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.

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

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.

[15] 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.

[16] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proc. 33th ACM Symp. on Principles of Programming Languages (POPL)*, pages 79–90, Charleston, South Carolina, USA, January 2006.

[17] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

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

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

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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

[29] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. In *The Eighth USENIX Security Symposium Proceedings*, pages 123–139, August 1999.

[30] 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.

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

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

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

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

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

[36] 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.

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

[38] 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.

[39] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

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