A More Precise Security Type System for Dynamic Security Tests

Gregory Malecha Stephen Chong
School of Engineering and Applied Sciences
Harvard University
gmalecha@cs.harvard.edu chong@seas.harvard.edu

Abstract

The move toward publically available services that store private information has increased the importance of tracking information flow in applications. For example, network systems that store credit-card transactions and medical records must be assured to maintain the confidentiality and integrity of this information. One way to ensure this is to use a language that supports static reasoning about information flow in the type system. While useful in practice, current type systems for checking information flow are imprecise, unnecessarily rejecting safe programs. This annoys programmers and often results in increased code complexity in order to work around these artificial limitations. In this work, we present a new type system for statically checking information flow properties of imperative programs with exceptions. Our key insight is to propagate a context of exception handlers and check exceptions at the throw point rather than propagating exceptions outward and checking them at the catch sites. We prove that our type system guarantees the standard non-interference condition and that it is strictly more permissive than the existing type system for Jif, a language that extends the Java type system to reason about information flow.

1. Introduction

Modern software platforms are becoming increasingly distributed and public. Both of these properties lead to systems that are more vulnerable to breaches in the integrity and confidentiality of information that they are entrusted to protect. Software bugs place both of these properties at risk, but even seemingly correct systems can unintentionally leak information with potentially disastrous consequences. For example, SQL injection attacks [Su and Wassermann 2006] and cross-site scripting (XSS) attacks [Endler 2002] are two of the most prevalent vulnerabilities on the Internet today [van der Stock et al. 2007] and both can be viewed as failures to properly enforce the flow of information [Dalton et al. 2007; Vogt et al. 2007]. In addition to addressing these problems, information flow analyses have been applied to help reason about a variety of other problems such as distributed systems development [Liu et al. 2009] and client-server application synthesis [Chong et al. 2007a,b].

PLAS '10 Toronto, Canada

Information flow bugs are difficult to detect because, unlike most types of correctness bugs, they often require specially crafted, uncommon input. Even more formal methods (e.g., Cadar, Cristian and Dunbar, Daniel and Engler, Dawson [2008]; Chlipala et al. [2009]) will often fail to catch information flow problems because specifications must explicitly state these security properties which are often overlooked or too verbose to specify for large systems. Even when specifications are written with security in mind, the burden of manually proving these properties is often too high and the brittleness of proofs in many systems makes refactoring and evolving code difficult; we require lighter-weight, more automatic, approaches for checking these properties.

While arbitrary correctness properties are difficult to reason about, programming language research has contributed a variety of systems for reasoning about specialized properties of code. In the realm of security, much work has gone into statically certifying *non-interference* [Goguen and Meseguer 1982] properties for calculi (e.g. Heintze and Riecke [1998]; Tse and Zdancewic [2007]) as well as dynamic monitoring of program execution to enforce information flow requirements (e.g. Askarov and Sabelfeld [2009]). Many of these analyses use type-oriented techniques originally developed by Volpano et al. [1996] to achieve modular and efficient checking. Researchers have capitalized on these features adapting the technique to several mainstream languages, including Java [Myers 1999] and OCaml [Pottier and Simonet 2003].

Jif [Myers 1999] is a programming language that extends the Java type system and run-time environment with support for reasoning about information flow. Unlike smaller calculi, Jif supports a considerable chunk of the standard Java language including classes, inheritance, and exceptions. Jif also provides run-time inspection of the security policy, a feature that is necessary for writing realistic programs that are parametric with respect to the run-time policy and support run-time policy modification. For example, many applications that deal with users, such as wikis and shared calendars, require a way to check whether run-time content, such as a web page or calendar event, should be accessible to a particular user.

The complexity of dynamic policy inspection coupled with non-local control flow leads to an imprecise and unnecessarily conservative analysis in Jif. Examples of this imprecision have been encountered in real developments in previous work (e.g., Clarkson et al. [2008]; Hicks et al. [2006]). This is a major impediment to development in Jif because secure Java code must be rewritten to work in Jif. In addition, the tricks required to convince the existing type system that code is secure obscure the code's meaning making it difficult to understand and maintain.

For example, consider the following Jif code that uses a run-time test to determine if information can flow from variable y to variable z.

```
1  int{*p} y = ...; /* y protected by label p */
2  int{*q} z = 0;   /* z protected by label q */
3
4  try {
5      if (p ⊑ q) {
6          /* information can flow from y to z */
7          if (y > 0) throw new Exception();
8      }
9  }
10 catch (Exception e) {
11     z = 1;
12 }
```

The program begins by defining two variables, y and z, with different protection levels. We assume that p and q are variables that contain protection levels, and the protection level of y and z are the contents of p and q respectively. The program then checks whether the run-time policy permits information to flow from protection level p to protection level q, with the dynamic test p $\sqsubseteq$ q. If information flow is allowed, the program performs a test on the value of y and raises an exception if it is greater than 0. The catch handler for the exception assigns the constant 1 to variable z. Thus information may flow from variable y to variable z, since the assignment to z depends on the value of y.

This example program is secure: information is only allowed to flow from y to z when it is permitted by the run-time security policy, i.e. when p $\sqsubseteq$ q. However, the Jif compiler rejects this program because the assignment to z in line 11 occurs outside the lexical scope of the dynamic security test p $\sqsubseteq$ q on line 5; the compiler has forgotten that the run-time check ensures that information is allowed to flow from p to q. In this work, we present a new type system that permits this information flow.

Outline and Contributions

We begin with background on information flow research including how security policies are described and what it means to be secure (Section 2). We then introduce Limp$_{\mathcal{L}}$, a loop-less imperative language that we will use to study information flow (Section 3). Then we adapt the Jif type system for Limp$_{\mathcal{L}}$ (Section 4.1) to serve as a basis for comparison. We then cover our primary contributions:

- We define a new typing relation for Limp$_{\mathcal{L}}$ that propagates information about exception handlers rather than exceptions (Section 4.2).
- We show that our type system guarantees a standard non-interference condition for information flow (Section 5.2).
- We show that our type system is strictly more permissive than the Jif-style type system for Limp$_{\mathcal{L}}$ (Section 5.3).
- We show how our type system can be cleanly extended to facilitate a hierarchy of exceptions (Section 6).

We conclude with a discussion of the insights of our new type system (Section 7) before considering related work (Section 7.1), and future directions (Section 8).

2. Background

In this section we cover the basics of information flow that are necessary to understand our system. We begin with a brief discussion of the methods for stating policies before formally defining the main semantic condition that we are interested in: non-interference.

For our purposes, *security policies* are join semi-lattices of security levels [Denning and Denning 1977]. Let $\mathcal{L}$ be the set of security levels, and $\sqsubseteq_{\mathcal{L}}$ the partial order over $\mathcal{L}$. We use Φ to denote the security policy.

$$\text{Security policy} \quad \Phi \quad = \quad (\mathcal{L}, \sqsubseteq_{\mathcal{L}})$$

Security levels $\mathcal{L}$ define the protection levels in the program and partial order $\sqsubseteq_{\mathcal{L}}$ defines permitted information can flow between security levels. That is, information is allowed to flow from level p to level q if and only if $p \sqsubseteq_{\mathcal{L}} q$. We use $\sqcup_{\mathcal{L}}$ to denote the *join operator*. Note that for any two security levels p and q, there exists $p \sqcup_{\mathcal{L}} q \in \mathcal{L}$ that is a least upper bound of both p and q. We further assume that there is a distinguished bottom security level $\bot \in \mathcal{L}$, such that $\bot \sqsubseteq_{\mathcal{L}} p$ for all $p \in \mathcal{L}$.

We note that this model of security levels can be used to reason about both confidentiality and integrity, and our results are applicable to rich security policy models, such as the decentralized label model [Myers and Liskov 1997].

In the literature, the standard notion of information security enforcement for programs is embodied in *non-interference* [Goguen and Meseguer 1982]. A system satisfies non-interference if low security outputs of the system are independent of high-security inputs. Intuitively, non-interference requires that information does not flow from high-security inputs to low-security outputs. Many variants and extensions of non-interference have been developed for addressing different types of information channels, such as termination and timing channels (see Sabelfeld and Myers [2003] for an overview). In this paper we focus only on basic non-interference since the core problems that we address arise even in this simplified context.

To state non-interference more formally, we first need to define some symbols and judgments. Since we will be reasoning about an imperative language we will model information flows through mutable stores σ. A store is a map from variables to values. We write $\sigma[x \mapsto v]$ for the store that maps variable x to value v, and otherwise behaves like store σ. We assume that each security level $o \in \mathcal{L}$ is able to observe the values of some subset of variables in the store; this subset is determined by a *variable environment* Γ. We write $\Gamma \vdash \sigma_1 \approx_o \sigma_2$ to mean that for all variables x that security level o may observe, we have $\sigma_1(x) = \sigma_2(x)$. Moreover, we say "$\Gamma$ protects x at level o" if variable x is not observable at any security level less restrictive than security level o.

To reason about the execution of a program s, we use the transitive closure of the small step operational semantics, $\Phi \vdash s, \sigma \rightarrow^* v, \sigma'$. This states that under security policy Φ and with store σ, the program s evaluates to value v and store σ'. In subsequent sections we will make both of these definitions more precise.

Based on the above definitions, we can formally state the non-interference property that we will be interested in:

Definition (Non-interference)**.** Program s *satisfies non-interference with respect to level o under variable environment* Γ if for all security policies Φ, for all stores σ, and for all variables h such that Γ protects h at level o' and $o' \not\sqsubseteq_{\mathcal{L}} o$, and for all values v_1, v_2 of the same type, if $\Phi \vdash s, \sigma[h \mapsto v_1] \rightarrow^* v_1', \sigma_1'$ and $\Phi \vdash s, \sigma[h \mapsto v_2] \rightarrow^* v_2', \sigma_2'$ then $\Gamma \vdash \sigma_1' \approx_o \sigma_2'$ and $v_1' = v_2'$.

This definition states that a program satisfies non-interference if an attacker with security level o, who can observe the values of some variables in the final store, can not learn anything about the high security input.

3. The Language

In this section we present Limp$_{\mathcal{L}}$, an imperative calculus for reasoning about security. Limp$_{\mathcal{L}}$ is based on *IMP* [Winskel 1993], but omits loops and adds named exceptions and first-class security levels. This choice of language constructs allows us to focus on the key differences between our type system and Jif's. Formally, the

$\boxed{\Phi \vdash e, \sigma \Downarrow v}$ **Expression Evaluation Semantics**

$$\frac{}{\Phi \vdash v, \sigma \Downarrow v}\ \text{E-VALUE} \qquad \frac{x \mapsto v \in \sigma}{\Phi \vdash x, \sigma \Downarrow v}\ \text{E-VAR}$$

$$\frac{\Phi \vdash e_j, \sigma \Downarrow i_j \qquad i = [\![i_1 \oplus i_2]\!]}{\Phi \vdash e_1 \oplus e_2, \sigma \Downarrow i}\ \text{E-OP}$$

$$\frac{\Phi = (\mathcal{L}, \sqsubseteq_{\mathcal{L}}) \qquad o_1 \sqsubseteq_{\mathcal{L}} o_2 \implies i = 1 \qquad \Phi \vdash e_i, \sigma \Downarrow o_i \qquad o_1 \not\sqsubseteq_{\mathcal{L}} o_2 \implies i = 0}{\Phi \vdash e_1 \sqsubseteq e_2, \sigma \Downarrow i}\ \text{E-FLOWS}$$

Figure 1. $\text{Limp}_{\mathcal{L}}$ expression evaluation semantics.

language is described by the following BNF:

Integers	i	$\in$	$\mathbb{Z}$
Variables	x, y	$\in$	Countably infinite set of names
Exceptions	C, D	$\in$	Finite set of names
Security Level	o	$\in$	$\mathcal{L}$
Expressions	e	$::=$	$x \mid i \mid o \mid e \oplus e \mid e \sqsubseteq e$
Statements	s	$::=$	$\textsf{skip} \mid x := e \mid s; s$
			$\mid \textsf{if}\ e\ \textsf{then}\ s\ \textsf{else}\ s$
			$\mid \textsf{throw}\ (C, e)$
			$\mid \textsf{try}\ s\ \textsf{catch}\ (C\, x)\ s$
			$\mid \textsf{try}\ s\ \textsf{finally}\ s$

Metavariables x and y range over the set of program variables, which are drawn from a countably infinite set of strings. Stores map variables to integers and security levels. Expressions in the language, ranged over by e, include variables, integers i, security levels o, and pure binary operations on integers $e \oplus e$. In addition, the language includes a dynamic security level test $e_1 \sqsubseteq e_2$, which evaluates e_1 and e_2 to security levels o_1 and o_2, and tests whether the run-time security policy allows information to flow from o_1 to o_2.

In addition to the standard statement constructs **skip**, assignment, sequence, and **if** statements, $\text{Limp}_{\mathcal{L}}$ supports simple named exceptions. Metavariables C and D range over exception names, and exceptions can carry a single integer value. The statement $\textsf{throw}\ (C, e)$ throws the exception named C, and the associated value is the result of evaluating expression e. The construct $\textsf{try}\ s_1\ \textsf{catch}\ (C\, x)\ s_2$ evaluates s_1, and if s_1 throws exception C with associated value v, then it executes s_2 with variable x bound to v. The construct $\textsf{try}\ s_1\ \textsf{finally}\ s_2$ evaluates s_1, and, regardless of whether s_1 terminates normally or exceptionally, evaluates s_2.

3.1 Semantics

The operational semantics of $\text{Limp}_{\mathcal{L}}$ are based on the operational semantics of *IMP* extended with exceptions and omitting **while** loops. Values in $\text{Limp}_{\mathcal{L}}$ are broken into two categories: expression values and statement values. We use metavariable v to range over both of these categories; it will be clear from the context whether an expression or statement value is intended.

Expression Values	v	$::=$	$i \mid o$
Statement Values	v	$::=$	$\textsf{skip} \mid \textsf{throw}\ (C, v)$

Expression values are either integers or security levels. Note that we are treating exception names as second-class so they are not values. Statement values include **skip**, which corresponds to normal termination of a statement, and $\textsf{throw}\ (C, v)$, which corresponds to exceptional termination with the exception name C carrying the expression value v.

Since expressions have very simple semantics that don't include side-effects, we use a big-step operational semantics to define them while using a small-step operational semantics for statements. The $\text{Limp}_{\mathcal{L}}$ evaluation relations have the following forms:

Expression Evaluation	$\Phi \vdash e, \sigma \Downarrow v$
Statement Evaluation	$\Phi \vdash s, \sigma \to s', \sigma'$

Both evaluation relations are parameterized by the run-time security policy Φ. It should be noted that, unlike in Jif, Φ can not change at run time. We make this simplifying assumption because allowing the policy to change complicates the definition of non-interference in a way that should be orthogonal to the aspects that we are considering. Both relations are parameterized by store σ, which we treat as a map from variables to values.

The semantics of expressions are presented in Figure 1. Values reduce to themselves and variables reduce to the value that the store assigns to them. We leave the set of binary operators abstract, requiring only that they are restricted to integer arguments and return values and are eager in both arguments, i.e., there is no short-circuit evaluation. We distinguish the flows binary operator ($\sqsubseteq$) which consults the run-time security policy Φ, returning 1 if the flow is permitted and 0 if it is not.

Figure 2 gives the small-step operational semantics for $\text{Limp}_{\mathcal{L}}$ statements. The semantics of assignment, sequence, and conditionals are standard. We describe the semantics of exceptions in more detail. Sequences beginning with a $\textsf{throw}\ (C, v)$ absorb the next statement (E-SEQTHROW). Statement values that are **throw**s are consumed at **catch** blocks reducing to the **catch** handler if the type of the exception matches the guard (E-CATCHCATCH). To void variable shadowing, we enforce that the names of variables bound in a **catch** block are disjoint from the domain of σ; we encode this implicitly with $\uplus$. If the exception name does not match the guard or the body results in a **skip**, the handler is ignored and body result is propagated (E-CATCHPASS). The **finally** construct is used to specify a statement that should execute regardless of whether or not an exception is thrown in the body. We describe this by stepping the body to a value, and then reducing the **try**...**finally** construct to a sequence of the **finally** statement and the value (E-FINALLY). Thus, if the **finally** block terminates normally the result is the result of the body and if the **finally** block terminates with an exception, then the value of the body is ignored and the exception is propagated.

4. Typing Information Flow

In this section we consider two type systems for checking information flow. First, we adapt the type system of Jif to $\text{Limp}_{\mathcal{L}}$, and then present our modified type system for the same language, highlighting the differences between out type system and Jif's.

Fundamentally, information flow tracks the security level of expressions and statements. The security level of an expression value is the least upper-bound of the security levels of all of the values that contributed to its construction. For statements, the security level is more subtle: the security level of a statement is an upper-bound on the information that may be gained by knowing whether the statement executes.

4.1 Type Checking *á la* Jif

The Jif type system was first published in Myers' PhD thesis [Myers 1999]. Since then the Jif language has undergone several simplifications [Chong and Myers 2006], though the spirit of the type system has remained mostly unchanged. At a high-level, Jif typing rules have a computational nature where the type of a term is built bottom-up by combining the types of subterms. We adapt the Jif type system for $\text{Limp}_{\mathcal{L}}$, preserving the computational nature. Both this adapted type system and our type system (presented in the next

$\boxed{\Phi \vdash s, \sigma \to s', \sigma'}$ **Statement Evaluation Semantics**

$$\frac{\Phi \vdash e, \sigma \Downarrow v \qquad \sigma' = \sigma[x \mapsto v]}{\Phi \vdash x := e, \sigma \to \mathsf{skip}, \sigma'} \text{ E-ASSIGN} \qquad \frac{\Phi \vdash e, \sigma \Downarrow v \qquad e \neq v}{\Phi \vdash \mathsf{throw}\,(C, e), \sigma \to \mathsf{throw}\,(C, v), \sigma} \text{ E-THROW}$$

$$\frac{\Phi \vdash s_1, \sigma \to s_1', \sigma'}{\Phi \vdash s_1; s_2, \sigma \to s_1'; s_2, \sigma'} \text{ E-SEQSTEP}$$

$$\frac{}{\Phi \vdash \mathsf{skip}; s_2, \sigma \to s_2, \sigma} \text{ E-SEQSKIP} \qquad \frac{\Phi \vdash e, \sigma \Downarrow v}{\Phi \vdash \mathsf{throw}\,(C, v); s_2, \sigma \to \mathsf{throw}\,(C, v), \sigma} \text{ E-SEQTHROW}$$

$$\frac{\Phi \vdash e, \sigma \Downarrow i \qquad e \neq i}{\Phi \vdash \mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2, \sigma \to \mathsf{if}\, i\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2, \sigma} \text{ E-IFSTEP} \qquad \frac{\begin{array}{c} i \neq 0 \implies s = s_1 \\ i \in \mathbb{Z} \qquad i = 0 \implies s = s_2 \end{array}}{\Phi \vdash \mathsf{if}\, i\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \to s, \sigma} \text{ E-IF}$$

$$\frac{\Phi \vdash s, \sigma \to s', \sigma'}{\Phi \vdash \mathsf{try}\, s\, \mathsf{catch}\,(C\,x)\, s_c, \sigma \to \mathsf{try}\, s'\, \mathsf{catch}\,(C\,x)\, s_c, \sigma'} \text{ E-CATCHSTEP}$$

$$\frac{}{\Phi \vdash \mathsf{try}\, \mathsf{throw}\,(C, v)\, \mathsf{catch}\,(C\,x)\, s_c, \sigma \to\ s_c, \sigma \uplus \{x \mapsto v\}} \text{ E-CATCHCATCH} \qquad \frac{v \neq \mathsf{throw}\,(C, v')}{\Phi \vdash \mathsf{try}\, v\, \mathsf{catch}\,(C\,x)\, s_c, \sigma \to v, \sigma'} \text{ E-CATCHPASS}$$

$$\frac{\Phi \vdash s, \sigma \to s', \sigma'}{\Phi \vdash \mathsf{try}\, s\, \mathsf{finally}\, s_c, \sigma \to \mathsf{try}\, s'\, \mathsf{finally}\, s_c, \sigma'} \text{ E-FINALLYSTEP} \qquad \frac{}{\Phi \vdash \mathsf{try}\, v\, \mathsf{finally}\, s_c, \sigma \to s_c; v, \sigma} \text{ E-FINALLY}$$

Figure 2. $\text{Limp}_{\mathcal{L}}$ statement evaluation semantics.

section) have the same structure of types.

Raw Types	τ	::=	$\mathsf{int} \mid \mathsf{level}$
Labels	l, m	::=	$o \mid {*x} \mid l \sqcup l$
Labeled Types	T	::=	$\tau\{l\}$

Raw types include integers and security levels. A *labeled type* is a pair of a raw type and a *security label*. A security label is either a security level o, a dynamic security level $*x$, or the symbolic join of two security labels $l_1 \sqcup l_2$. Dynamic security label $*x$ refers to the security level stored in variable x. For example, the type $\mathsf{int}\{*x\}$ is the type of integer values protected by the security level stored in the variable x. A dynamic security level is a simple kind of dependent type. The type system will ensure that variables that store security levels are immutable, that is, they will not be modified during the execution of the program. This restriction is analogous the requirement in Jif that any label variable is declared **final**, and is needed for soundness of the type system.

A symbolic join $l_1 \sqcup l_2$ represents a security level that is an upper bound of labels l_1 and l_2. Since l_1 and l_2 may be dynamic security levels, their value may not be known statically. Symbolic joins allow precise static reasoning about upper bounds of labels. The decentralized label model [Myers and Liskov 1997], used in the Jif type system, has an uninterpreted join operation that enables precise static reasoning about upper bounds.

We begin with the Jif typing rules for expressions. To introduce the expression typing judgment, we first introduce variable environments and label constraint environments.

Variable Environment	Γ	::=	$x \mapsto T :: \Gamma \mid \bullet$
Label Constraint Environment	δ	::=	$l \sqsubseteq m \mid \mathsf{True} \mid \delta \wedge \delta$
Expression Typing			$\Gamma, \delta \vdash^J e : T$

Variable environment Γ is a partial map from variables to labeled types. The empty environment is written $\bullet$. Label constraint environment δ encodes statically known information about the run-time security policy. It is constructed as a conjunction of flows facts. The flow fact **True** corresponds to knowing nothing about the run-time security policy; flow fact $l \sqsubseteq m$ means that the run-time security policy allows information to flow from the security level represented by l to the security level represented by m.

Finally, the Jif expression typing judgment $\Gamma, \delta \vdash^J e : T$ states that expression e has labeled type T under the variable environment Γ and label constraint environment δ. Inference rules for this judgment are presented in Figure 3. For variables, we simply look up the type in the environment (TJIF-VAR). Integer constants are typed to $\mathsf{int}\{\bot\}$ (TJIF-INT) and level constants are typed to $\mathsf{level}\{\bot\}$ (TJIF-LEVEL) because we assume that an observer has access to the source code and can therefore read the constants. Operations, both $\oplus$ and $\sqsubseteq$, require the appropriate raw types for their arguments and compute a result protected by the join of the argument labels, since the resulting value depends on both of the input values.

Note that rules TJIF-OP and TJIF-FLOWS use the judgment $\delta \vdash l \sqsubseteq l'$, which means that using label constraint environment δ, we can prove that security label l' is an upper bound of security label l. The inference rules for this judgment are given in Figure 4.

To present the Jif typing rules for statements, we must first introduce *program counter labels* and *path maps*. A program counter label represents the information that may be gained by knowing that a statement executes. To prevent illegal information flows, security type systems generally use the program counter label as a lower bound on the side effects of a statement. That is, the label of

$\boxed{\Gamma, \delta \vdash^J e : T}$ **Jif Typing Expressions**

$$\frac{x \mapsto \tau\{l\} \in \Gamma}{\Gamma, \delta \vdash^J x : \tau\{l\}} \text{ TJIF-VAR}$$

$$\frac{}{\Gamma, \delta \vdash^J i : \textbf{int}\{\bot\}} \text{ TJIF-INT}$$

$$\frac{}{\Gamma, \delta \vdash^J o : \textbf{level}\{\bot\}} \text{ TJIF-LEVEL}$$

$$\frac{\Gamma, \delta \vdash^J e_i : \textbf{int}\{l_i\} \qquad \delta \vdash l_1 \sqcup l_2 \sqsubseteq l}{\Gamma, \delta \vdash^J e_1 \oplus e_2 : \textbf{int}\{l\}} \text{ TJIF-OP}$$

$$\frac{\Gamma, \delta \vdash^J e_i : \textbf{level}\{l_i\} \qquad \delta \vdash l_1 \sqcup l_2 \sqsubseteq l}{\Gamma, \delta \vdash^J e_1 \sqsubseteq e_2 : \textbf{int}\{l\}} \text{ TJIF-FLOWS}$$

Figure 3. Jif typing rules for expressions.

$\boxed{\delta \vdash l \sqsubseteq l}$ **Flows Derivation Under Assumptions**

$$\frac{}{\delta \vdash l \sqsubseteq l} \text{ F-REFL}$$

$$\frac{\delta \vdash l_1 \sqsubseteq l_2 \qquad \delta \vdash l_2 \sqsubseteq l_3}{\delta \vdash l_1 \sqsubseteq l_3} \text{ F-TRANS}$$

$$\frac{}{\delta \wedge l \sqsubseteq l' \vdash l \sqsubseteq l'} \text{ F-CONTEXT}$$

$$\frac{\delta \vdash l \sqsubseteq l'' \qquad \delta \vdash l' \sqsubseteq l''}{\delta \vdash l \sqcup l' \sqsubseteq l''} \text{ F-JOIN}$$

$$\frac{}{\delta \vdash l \sqsubseteq l \sqcup l'} \text{ F-JOINL}$$

$$\frac{}{\delta \vdash l' \sqsubseteq l \sqcup l'} \text{ F-JOINR}$$

Figure 4. Flows derivation.

any variable updated by a statement must be at least as restrictive as the program counter label.

A path map describes the information that may be gained by observing that a statement terminated with a particular exception.[1]

$$\text{Path Map} \quad A \quad ::= \quad \{C \mapsto (l, T)\} \uplus A \mid \bullet$$

An empty path map is written $\bullet$. Path map $\{C \mapsto (l, T)\} \uplus A$ extends path map A by associating exception name C with the pair (l, T), where label l is an upper bound on the information that caused C to be thrown (i.e., the program counter at the point of the **throw** statement), and T is the labeled type of the value that the exception carries. (The type system will restrict T to being labeled **int**s; we use labeled types to clarify the distinction between the label of the decision to throw C and the label of the value carried by the exception.) While we describe path maps concretely as an association list with unique keys, we abuse notation and also treat them as functions from exception names to labels, defined as follows:

$$A(C) = \begin{cases} l & \text{if } C \mapsto (l, T) \in A \\ \emptyset & \text{otherwise} \end{cases}$$

Here, if a mapping does not occur in the list, we use a special label $\emptyset$ (pronounced "not taken") that is used to describe impossible paths. When used as a program counter label for a statement s, $\emptyset$ means that s is unreachable. Label $\emptyset$ is necessary for typing dead code, such as code sequences after a **throw** statement. Like security level $\bot$, $\emptyset$ is a lower bound of all labels; we distinguish $\emptyset$ from $\bot$ because label $\emptyset$ is an artifact of the type system, and is not a security level like $\bot$.

$$\text{Statement Typing} \qquad \Gamma, \delta \vdash^J_l s : l', A$$

The Jif typing judgment for statements has the form $\Gamma, \delta \vdash^J_l s : l', A$, where Γ is a variable environment, δ is a label constraint environment, l is the program counter label, s is a statement, l' is a label that indicates what information may be gained by knowing that s terminated normally (referred to as the "normal termination label"), and A is a path map that describes what information may be gained by knowing that s terminated with an exception. Inference rules for this judgment are given in Figure 5.

The rule for **skip** (TJIF-SKIP) says that **skip** can be typed at any program counter label l and the normal termination label is the same as the program counter label. Assignment statements $x := e$ are checked using the TJIF-ASSIGN rule, which determines the labeled type of e and checks that values with that label can be stored in variable x assuming the constraints on δ hold. Since **skip** and assignment statements always terminate normally in well-typed programs, the resulting path map is empty in both cases. Note that the assignment rule enforces the immutability of label variables by requiring the raw types to be **int**.

The sequencing rule TJIF-SEQ for $s_1; s_2$ recursively constructs a type for s_1, and uses the normal termination label of s_1 as the program counter label of s_2. This is because if an observer learns that s_2 executes, she knows that s_1 terminated normally. The normal termination label of the sequence is the normal termination label of s_2, and the resulting path map is an upper bound of the path maps from s_1 and s_2, denoted $\delta \vdash A_1 \sqcup A_1 \sqsubseteq A$ and defined in Figure 5.

To check conditional statements, we determine the label of the test expression and taint the program counter with it when checking the branches. This is because knowing which branch executed may allow an observer to determine the evaluation of the test expression. In addition, we extract information about the run-time security policy that can be learned from the evaluation of the test expression, using the function $implies(-)$, defined in Figure 5 Specifically, if the expression $e_1 \sqsubseteq e_2$ evaluates to 1, then we add the corresponding flow fact to the label constraint environment δ when checking the consequent. Soundness will only require that the *implies* function be a conservative approximation of the run-time policy; i.e.

$$\forall \Phi, e, \sigma, v.\ \Phi \vdash e, \sigma \Downarrow v \wedge v \neq 0 \rightarrow \Phi \vdash implies(e).$$

Including this allows the Jif type system to accept programs whose security depends on the run-time security policy, such as the following program where the assignment to y is allowed because it will only be executed if the run-time security policy permits information flow from x to y.

```
1 // Γ(x) = int{*p}
2 // Γ(y) = int{*q}
3 if (p ⊑ q) then
4     y := x
5 else
6     skip
```

[1] Path maps as presented by Myers [1999] included the normal termination label; we distinguish the normal termination label in the typing judgment.

$\boxed{\Gamma, \delta \vdash_l^J s : l', A}$ **Jif Typing Statements**

$$\frac{}{\Gamma, \delta \vdash_l^J \ \mathsf{skip} : l, \bullet}\ \textsc{TJif-Skip} \qquad \frac{\Gamma, \delta \vdash^J e : \mathsf{int}\{l_e\} \qquad \delta \vdash l \sqsubseteq l' \qquad \Gamma(x) = \mathsf{int}\{l'\} \qquad \delta \vdash l_e \sqsubseteq l'}{\Gamma, \delta \vdash_l^J x \ := \ e : l, \bullet}\ \textsc{TJif-Assign}$$

$$\frac{\Gamma, \delta \vdash_l^J s_1 : l', A_1 \qquad \Gamma, \delta \vdash_{l'}^J s_2 : l'', A_2 \qquad \delta \vdash A_1 \sqcup A_2 \sqsubseteq A}{\Gamma, \delta \vdash_l^J s_1;\ s_2 : l'', A}\ \textsc{TJif-Seq}$$

$$\frac{\Gamma, \delta \vdash^J e : \mathsf{int}\{l'\} \qquad \delta \vdash A_1 \sqcup A_2 \sqsubseteq A \qquad \Gamma, \delta \wedge \mathit{implies}(e) \vdash_{l \sqcup l'}^J s_1 : l_1', A_1 \qquad \Gamma, \delta \vdash_{l \sqcup l'}^J s_2 : l_2', A_2}{\Gamma, \delta \vdash_l^J \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : l_1' \sqcup l_2', A}\ \textsc{TJif-If}$$

$$\frac{\Gamma, \delta \vdash^J e : \mathsf{int}\{l'\} \qquad \delta \vdash l \sqcup l' \sqsubseteq l''}{\Gamma, \delta \vdash_l^J \mathsf{throw}\ (C, e) : \emptyset, \{C \mapsto (l, \mathsf{int}\{l''\})\}}\ \textsc{TJif-Throw}$$

$$\frac{\Gamma, \delta \vdash_l^J s : l_1', A_1 \qquad x : \tau\{l_x\} :: \Gamma, \delta \vdash_{l_C}^J s_e : l_2', A_2 \qquad C \mapsto (l_C, \tau\{l_x\}) \in A_1 \qquad \delta \vdash A_1 \backslash C \sqcup A_2 \sqsubseteq A \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, \delta \vdash_l^J \mathsf{try}\ s\ \mathsf{catch}\ (C\,x)\ s_e : l_1' \sqcup l_2', A}\ \textsc{TJif-Catch}$$

$$\frac{\Gamma, \delta \vdash_l^J s : l_1, A_1 \qquad \Gamma, \delta \vdash_l^J s_f : l_2, A_2 \qquad A_1' \equiv A_1 \sqcup l_2 \qquad \delta \vdash A_1' \sqcup A_2 \sqsubseteq A}{\Gamma, \delta \vdash_l^J \mathsf{try}\ s\ \mathsf{finally}\ s_f : l_1 \sqcup l_2, A}\ \textsc{TJif-Finally}$$

$$\frac{\Gamma, \delta \vdash_l^J s : l', \bullet}{\Gamma, \delta \vdash_l^J s : l, \bullet}\ \textsc{TJif-SinglePath} \qquad \frac{\Gamma, \delta \vdash_l^J s : \emptyset, \{C \mapsto (l_C, T)\}}{\Gamma, \delta \vdash_l^J s : \emptyset, \{C \mapsto (l, T)\}}\ \textsc{TJif-SinglePathEx}$$

$\boxed{\mathit{implies}(e)}$ **Expression Implications**

$$\mathit{implies}(e) = \begin{cases} l_1 \sqsubseteq l_2 & \text{if } e = e_1 \sqsubseteq e_2, l_1 = \mathit{exprToLabel}(e_1), \text{ and } l_2 = \mathit{exprToLabel}(e_2) \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathit{exprToLabel}(e) = \begin{cases} o & \text{if } e = o \\ *x & \text{if } e = x \end{cases}$$

$\boxed{A \backslash C}$ **Path map removal**

$$A \backslash C = \begin{cases} \bullet & \text{if } A = \bullet \\ A' & \text{if } A = \{C \mapsto (l, T)\} \uplus A' \\ \{D \mapsto (l, T)\} \uplus A' \backslash C & \text{if } A = \{D \mapsto (l, T)\} \uplus A' \text{ and } D \neq C \end{cases}$$

$\boxed{\delta \vdash A \sqcup A \sqsubseteq A}$ **Path Map Upper Bounds**

$$\frac{\forall i \in 1..2.\ \forall C \mapsto (l, \tau\{m\}) \in A_i.\ \exists D \mapsto (l', \tau\{m'\}) \in A.\ \delta \vdash l \sqsubseteq l' \ \wedge \ \delta \vdash m \sqsubseteq m'}{\delta \vdash A_1 \sqcup A_2 \sqsubseteq A}$$

$\boxed{A \sqcup l}$ **Lifting Path Maps by Labels**

$$A \sqcup l = \begin{cases} \bullet & \text{if } A = \bullet \\ \{C \mapsto (l' \sqcup l, \tau\{m' \sqcup l\})\} \uplus (A' \sqcup l) & \text{if } A = \{C \mapsto (l', \tau\{m'\})\} \uplus A' \end{cases}$$

Figure 5. Jif statement typing rules.

The path map of a conditional statement is an upper bound of the path maps of the consequent and the alternative, since the information gained by knowing the conditional terminated with an exception may reveal that either the consequent or the alternative terminated with an exception.

Exceptional control structures use the path map to track the information that may be learned by observing that a particular exception was thrown. TJIF-THROW produces a path map that maps the raised exception to the program counter label and specifies a normal termination label of $\emptyset$ since **throw** (C, v) never terminates normally. For the **try** s **catch** $(C\,x)$ s_e construct (TJIF-CATCH), we type-check the body of the **try**, s, and use the label associated with exception C as the program counter label for the **catch** handler s_e. Note that if the exception can not be thrown from within s, then the program counter label for s_e is $\emptyset$, indicating that the catch handler is unreachable. The path map for the **try**...**catch** construct is obtained by removing C from A_1 (denoted $A \backslash C$) and joining it with A_2; this corresponds to propagating non-C exceptions from s and all exceptions from s_e. The function $A \backslash C$ is defined in Figure 5. The normal termination label the **try** ... **catch** statement is the join of the normal termination labels of s and s_e, since the construct terminates normally if either s or s_e terminates normally. Finally, we require that the variable x is not currently in the variable environment and that it binds an integer value. This ensures that there is no shadowing of variables, and that the labeled type of a variable cannot refer to variables introduced in catch handlers. These restrictions simplify type-checking, but are not fundamental limitations.

The **try** s **finally** s_f construct is similar to a combination of sequencing and catch. We check both s and s_f with the initial program counter label since we know that s_f is guaranteed to execute regardless of the behavior of s. Normal termination of the **try**...**finally** block requires normal termination of both s and s_f, and the normal termination label is thus the join of the normal termination labels of s and s_f. According to the operational semantics, an exception thrown by s only propagates if s_f terminates normally. Thus, any exception thrown by s that propagates reveals that s_f terminated normally. We thus join the path map of s with the normal termination label of s_f, denoted $A \sqcup l$.

The single path rules TJIF-SINGLEPATH and TJIF-SINGLEPATHEX state that if a statement can terminate in only one way (either normally, or with some particular exception), then the (normal or exceptional) termination label can be lowered to be the same as the program counter label of the statement. This allows, for example, the program (**if** h **then skip else skip**); l := 7 to type check (where l and h have different security labels), since the **if** command can only terminate normally. The single path rules are important for expressiveness.

4.2 A Revised Type System

By propagating exceptions outward and checking them at their handlers, the Jif type system loses contextual information from where the exception was thrown. For example, if exception C is only thrown in contexts where $l \sqsubseteq m$ is known through a runtime test, then at the catch handlers for C, the flow fact $l \sqsubseteq m$ will always be satisfied, but may not be in the label constraint environment.

This could be addressed by augmenting path maps with label constraint information, and extending the path map join operation to merge this information intelligently. However, the same precision can be obtained by regarding path maps as constraints that statements must satisfy, rather than summarizations of the behavior of statements. Our type system propagates information about catch handlers inwards, and **throw** statements may only throw exceptions for which there is an appropriate enclosing catch handler. Thus, we will use path maps to describe the environment in which a statement occurs. It is interesting to note that the Jif type system already treats the label constraint environment in this way, as a description of the enclosing context of a statement; our type system provides a more uniform treatment of path maps and label constraint environments.

$\boxed{\Gamma, \delta, l \vdash e : \tau}$ **Modified typing rules for Limp$_\mathcal{L}$ expressions**

$$\frac{\Gamma(x) = \tau\{l'\} \qquad \delta \vdash l' \sqsubseteq l}{\Gamma, \delta, l \vdash x : \tau}\ \text{T-VAR}$$

$$\frac{}{\Gamma, \delta, l \vdash i : \textbf{int}}\ \text{T-INT}$$

$$\frac{}{\Gamma, \delta, l' \vdash o : \textbf{level}}\ \text{T-LEVEL}$$

$$\frac{\Gamma, \delta, l \vdash e_i : \textbf{int}}{\Gamma, \delta, l \vdash e_1 \oplus e_2 : \textbf{int}}\ \text{T-OP}$$

$$\frac{\Gamma, \delta, l \vdash e_i : \textbf{level}}{\Gamma, \delta, l \vdash e_1 \sqsubseteq e_2 : \textbf{int}}\ \text{T-FLOWS}$$

Figure 6. Modified Limp$_\mathcal{L}$ typing rules for expressions.

With this philosophy in mind, we present new typing rules for expressions and statements. The Jif typing judgment for expressions had the form $\Gamma, \delta \vdash^J e : \tau\{l\}$. Our typing judgment has the same entities, but we emphasize that label l is a constraint on the label of the expression by moving l to the left of the turnstile. Our typing judgment for expressions has the form $\Gamma, \delta, l \vdash e : \tau$, and the rules are given in Figure 6. The rules are mostly similar to the Jif rules presented earlier. The differences are highlighted by T-VAR which checks that looking up the variable in the environment returns the same type as τ and a label that protects information that can flow into the desired result label. In the spirit of checking in the most permissive context, the rules for integer constants and constant labels use a free label l to express that a constant can be used in any context. Finally, rather than combining labels in the T-OP and T-FLOWS rules we simply propagate the upper-bound constraint label into the checking of the subterms.

The Jif typing rules for expressions and the new rules presented here are essentially equivalent in expressiveness: if $\Gamma, \delta \vdash^J e : \tau\{l\}$ then $\Gamma, \delta, l \vdash e : \tau$; and if $\Gamma, \delta, l \vdash e : \tau$ then there exists some l' such that $\delta \vdash l' \sqsubseteq l$ and $\Gamma, \delta \vdash^J e : \tau\{l'\}$.

We make similar changes to the typing rules for statements. The form of the Jif typing judgment for statements is $\Gamma, \delta \vdash^J_l s : l', A$. Since we now regard path maps and the normal termination label as expressing constraints that s must satisfy (or alternatively, as describing the context in which s appears), we move path map A and normal termination label l' to the left of the turnstile, resulting in a new judgment of the following form.

$$\Gamma, \delta, A, l' \vdash_l s\ \textbf{ok}$$

Since path maps are now used to describe the context in which a statement occurs, we refer to them as *exception environments* in this section.

The rules for our typing relation are given in Figure 7. For **skip**, we require that the normal termination label is an upper bound of the program counter label and place no restrictions on the exception environment A. Rule T-ASSIGN for assignment $x := e$ ensures that both the label of expression e and the program counter label are bounded above by the label of the variable. The normal termi-

$\boxed{\Gamma, \delta, A, l' \vdash_l s \textbf{ ok}}$ **Modified typing rules for Limp$_\mathcal{L}$ statements**

$$\frac{\delta \vdash l \sqsubseteq l'}{\Gamma, \delta, A, l' \vdash_l \textbf{ skip ok}} \text{ T-SKIP}$$

$$\frac{\Gamma(x) = \textbf{int}\{l_x\} \qquad \delta \vdash l \sqsubseteq l_x \qquad \Gamma, \delta, l_x \vdash e : \textbf{int} \qquad \delta \vdash l \sqsubseteq l'}{\Gamma, \delta, A, l' \vdash_l x := e \textbf{ ok}} \text{ T-ASSIGN}$$

$$\frac{\Gamma, \delta, A, l' \vdash_l s_1 \textbf{ ok} \qquad \Gamma, \delta, A, l'' \vdash_{l'} s_2 \textbf{ ok}}{\Gamma, \delta, A, l'' \vdash_l s_1;\ s_2 \textbf{ ok}} \text{ T-SEQ}$$

$$\frac{\Gamma, \delta \wedge \mathit{implies}(e), A, l'' \vdash_{l \sqcup l'} s_1 \textbf{ ok} \qquad \Gamma, \delta, l' \vdash e : \textbf{int} \qquad \Gamma, \delta, A, l'' \vdash_{l \sqcup l'} s_2 \textbf{ ok}}{\Gamma, \delta, A, l'' \vdash_l \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ ok}} \text{ T-IF}$$

$$\frac{C \mapsto (l_C, \textbf{int}\{l_x\}) \in A \qquad \delta \vdash l \sqsubseteq l_x \qquad \Gamma, \delta, l_x \vdash e : \textbf{int} \qquad \delta \vdash l \sqsubseteq l_C}{\Gamma, \delta, A, l' \vdash_l \textbf{throw } (C, e) \textbf{ ok}} \text{ T-THROW}$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma, \delta, A[C \mapsto (l_C, \textbf{int}\{l_x\})], l' \vdash_l s \textbf{ ok} \qquad x : \textbf{int}\{l_x\} :: \Gamma, \delta, A, l' \vdash_{l_C} s_e \textbf{ ok}}{\Gamma, \delta, A, l' \vdash_l \textbf{try } s \textbf{ catch } (C\, x)\ s_e \textbf{ ok}} \text{ T-CATCH}$$

$$\frac{\Gamma, \delta, A', l' \vdash_l s \textbf{ ok} \qquad \Gamma, \delta, A, l'' \vdash_l s_f \textbf{ ok} \qquad \delta \vdash A \backslash l'' = A' \qquad \delta \vdash l'' \sqsubseteq l'}{\Gamma, \delta, A, l' \vdash_l \textbf{try } s \textbf{ finally } s_f \textbf{ ok}} \text{ T-FINALLY}$$

$$\frac{\Gamma, \delta, \bullet, l'' \vdash_l s \textbf{ ok} \qquad \delta \vdash l \sqsubseteq l'}{\Gamma, \delta, A, l' \vdash_l s \textbf{ ok}} \text{ T-SINGLEPATH}$$

$$\frac{C \mapsto (l_C, \textbf{int}\{l_x\}) \in A \qquad \Gamma, \delta, \{C \mapsto (l'_C, \textbf{int}\{l'_x\})\}, \emptyset \vdash_l s \textbf{ ok} \qquad \delta \vdash l \sqsubseteq l_C \qquad \delta \vdash l'_x \sqsubseteq l_x}{\Gamma, \delta, A, l' \vdash_l s \textbf{ ok}} \text{ T-SINGLEPATHEX}$$

$\boxed{\delta \vdash A \backslash l = A'}$ **Exception environment filter**

$$\frac{}{\delta \vdash \bullet \backslash l = \bullet}$$

$$\frac{\delta \vdash A \backslash l = A' \qquad \delta \not\vdash l \sqsubseteq l_C}{\delta \vdash \{C \mapsto (l_C, T)\} \uplus A \backslash l = A'}$$

$$\frac{\delta \vdash A \backslash l = A' \qquad \delta \vdash l \sqsubseteq l_C}{\delta \vdash \{C \mapsto (l_C, T)\} \uplus A \backslash l = \{C \mapsto (l_C, T)\} \uplus A'}$$

Figure 7. Modified typing rules for Limp$_\mathcal{L}$ statements.

nation label must be at least as restrictive as the program counter label, and there are no restrictions on the exception environment since, like **skip** assignment statements never throw exceptions. For sequences, T-SEQ passes the exception environment downward to the subterms and ensures that the normal termination label of s_1 is bounded above by the program counter label of s_2.

The intuition behind the new typing rule for conditional statements is the same as for the Jif typing rule: we require that the consequent and alternate be typable in contexts where the program counter is tainted by the label of the test expression. As with the Jif rule, we extend label constraint environment δ for the consequent with the additional flow facts implied by non-zero evaluation of the test expression, using the same *implies* function. The only difference with the Jif typing rule is that the exception environment is propagated inwards, using the same exception environment for both the consequent and the alternative.

Exceptional control flow structures constitute the most considerable differences between the type systems. Now that we interpret A as constraints on the exceptions thrown by the statement, the side condition for checking whether an exception can be thrown has moved from the rule for catch handlers to the rule for **throw**. In rule T-THROW, we check that a handler $(l_C, \textbf{int}\{l_x\})$ is in the exception environment, and ensure that the value thrown is bounded above by l_x, and that the program counter label is bounded above by l_C. Note that the label constraint environment used to check these upper bounds is the label constraint environment at the **throw** statement, which may contain more flow facts that the label constraint environment at the corresponding catch handler.

In rule T-CATCH for construct **try** s **catch** $(C\, x)$ s_e, we check statement s using an exception environment that is extended with a catch handler: $A[C \mapsto (l_C, \textbf{int}\{l_x\})]$, where l_C is the program counter label of the catch handler. (Path map extension $A[C \mapsto (l, T)]$ is defined as $\{C \mapsto (l, T)\} \uplus (A \backslash C)$.)

In rule T-FINALLY for statement **try** s **finally** s_f, since s_f is executed regardless of the normal or exceptional termination of s, the program counter label is the same for both s and s_f. Since an exception thrown by s is only propagated if s_f terminates normally, we need to restrict the exception environment given to s. We use judgment $\delta \vdash A \backslash l'' = A'$ to ensure that A', the exception environment given to s, is derived from exception environment A by removing any handler whose label is not provably an upper bound of l'', the normal termination label of s_f. This restriction of the exception environment is required to rule out programs with illegal information flow via the **finally** construct. For example, consider the following program, assuming that lo and hi have different security levels and information flows is not allowed to flow from hi to lo. Rule T-FINALLY would reject this program, since the catch handler for exception C must be removed from the exception environment used to check the **throw** statement on line 3. Without removing the exceptions, the **throw** is checked in a context that can throw C allowing the implicit flow.

```
1  try {
2      try {
3          throw (C, 0)
4      }
5      finally {
6          if (hi) then throw (D, 0) else skip
7      }
8  }
9  catch (C x) {
10     lo := 1 /* here, we know that hi is non−zero */
11 }
```

Rule T-SINGLEPATH states that if a statement type checks with an empty exception environment (i.e., it can only terminate nor-

mally), then the statement typechecks in any context where the normal termination label is bounded below by the program counter label. Rule T-SINGLEPATHEX is similar, but allows us to specify a distinguished exception from the exception environment. Note that in this case, the normal termination program counter label is free since statement s will never terminate normally.

Since our rules are fundamentally non-structural it could be difficult to determine when to apply these rules without exception propagation information, which the Jif type system collects implicitly in its typing rules. This can be addressed by performing a simple analysis to determine the ways by which a statement can terminate, and using the results of this analysis to guide the guesses for applications of single path rules.

5. Type System Properties

The goal of our type system is to ensure that well-typed $\text{Limp}_{\mathcal{L}}$ programs neither get stuck nor leak information. We also show that our type system is strictly more permissive than the Jif type system adapted to our calculus. Here we present only high-level proof sketches; full proofs are given in the companion technical report [Malecha and Chong 2010]. Before stating our theorems, we define some judgments that relate variable environments and stores and clarify some notations described earlier.

We say that store σ is typed by variable environment Γ (written $\Gamma \vdash \sigma$) if for every variable x the type of value $\sigma(x)$ is $\Gamma(x)$. More formally,

$$\Gamma \vdash \sigma \triangleq \forall x,\ \tau,\ l.\ x : \tau\{l\} \in \Gamma \Rightarrow \exists v : \tau, \sigma(x) = v$$

Observational equivalence of two stores, $\Gamma \vdash \sigma_1 \approx_o \sigma_2$, which was described in Section 2, is defined as follows.

$$\Gamma \vdash \sigma_1 \approx_o \sigma_2 \triangleq \forall x.\ \tau\{o\} \in \Gamma \Rightarrow \sigma_1(x) = \sigma_2(x)$$

In Section 2, we used the phrase "Γ protects x at level o" to mean variable x is not observable at any security level less restrictive than security level o. More formally, we say Γ *protects* x *at level* o if $\Gamma(x) = \tau\{o\}$.

5.1 Type Safety

The simplicity of the types in our calculus make proving type safety simple. Based on our small-step semantics, we prove progress and preservation lemmas that show that well-typed terms do not get stuck during evaluation and that evaluation preserves well-typedness.

Lemma 1 (Progress)**.** *If* $\Gamma, \delta, A, l' \vdash_l s$ ***ok*** *and* $\Gamma \vdash \sigma$*, then either* s *is a value or* $\Phi \vdash s, \sigma \to s', \sigma'$*.*

Proof. Induction on the typing derivation. □

Lemma 2 (Preservation)**.** *If* $\Gamma, \delta, A, l' \vdash_l s$ ***ok****,* $\Gamma \vdash \sigma$*, and* $\Phi \vdash s, \sigma \to s', \sigma'$*, then* $\Gamma, \delta, A, l' \vdash_l s'$ ***ok****.*

Proof. Induction on the typing derivation. □

Appropriately adapted forms of these lemmas are also true for the Jif type system since the two type systems have the same rules when labeled types are erased to raw types. Combined these theorems suggest that ignoring the labels in our type system leads to a standard type system for loopless *IMP* with exceptions.

5.2 Non-interference

The focus of this work is proving that well-typed terms are secure, that is, they satisfy non-interference. We prove the following theorem for our type system.

Theorem 3 (Non-interference)**.** *For all statements* s*, contexts* Γ*, and security labels* o*, if*

$$\Gamma, \textsf{True}, \bullet, o \vdash_l s\ \textbf{\textit{ok}}$$

then for all security policies Φ*, for all stores* σ*, and for all variables* h *such that* $\Gamma(h) = \tau\{o'\}$ *and* $o' \not\sqsubseteq_{\mathcal{L}} o$*, and for all values* v_1*,* v_2 *of type* τ*, if*

$$\Phi \vdash s, \sigma[h \mapsto v_1] \to^* v_1', \sigma_1'$$

and

$$\Phi \vdash s, \sigma[h \mapsto v_2] \to^* v_2', \sigma_2'$$

then $\Gamma \vdash \sigma_1' \approx_o \sigma_2'$ *and* $v_1' = v_2'$*.*

Proof. Using the technique of Pottier and Simonet [2003], we define a language $\text{Limp}^2_{\mathcal{L}}$ that is capable of modeling two different executions of a program. We then prove by induction that the two input stores will produce two observationally equivalent final stores. □

5.3 Precision

To understand the relationship between our type system and the standard Jif type system, we show that our type system accepts strictly more programs than the Jif type system adapted for our calculus.

Theorem 4 (Inclusion)**.** *If* $\Gamma, \delta \vdash^J_l s : l', A$ *then* $\Gamma, \delta, A, l' \vdash_l s$ ***ok****.*

Proof. Induction on the Jif typing relation. The key insight is that we can pick all of the same labels as the Jif type system picked. The label constraint environment that our type system uses to check conditions of the form $\delta \vdash l_1 \sqsubseteq l_2$ contains at least as much information as the corresponding label constraint environment that the Jif type system uses to check the same condition. □

Theorem 5 (Strict Inclusion)**.** *There exists* s, Γ, δ, l, l'*, and* A *such that* $\Gamma, \delta, A, l' \vdash_l s$ ***ok*** *and not* $\Gamma, \delta \vdash^J_l s : l', A$*.*

Proof. Adapting the Jif program from the introduction to $\text{Limp}_{\mathcal{L}}$ we have:

```
1 try
2   if (p ⊑ q) then
3     if (y > 0) then throw (C, 0) else skip
4   else skip
5 catch (C x)
6   z = 1
```

Let $\Gamma = \{\mathsf{p} : \textsf{level}\{\bot\}, \mathsf{q} : \textsf{level}\{\bot\}, \mathsf{y} : \textsf{int}\{*\mathsf{p}\}, \mathsf{z} : \textsf{int}\{*\mathsf{q}\}\}$. When checking the **throw** statement, the flow fact $\mathsf{p} \sqsubseteq \mathsf{q}$ is in the label constraint environment. Our type system uses this flow fact to conclude that the **throw** is legal in the context. The Jif type system, on the other hand, checks whether the exception can be thrown with the empty label constraint environment, and is therefore unable prove that the program is secure. □

The witness used in the proof of Theorem 5 is by no means the only witness, but is a rather simple one. In general, our reformulation of the Jif type system is more precise in checking programs that throw exceptions under conditions that test the label lattice in **try** blocks.

6. Exceptions with Subtyping

The calculus presented here is very simple, in order to present the key innovation of our type system. However, the full Jif programming language contains many additional language features. In this section, we bring our calculus slightly closer to the full expressiveness of the Jif language by extending our calculus with exceptions with subtyping and show how to adapt our type system. We will assume that it is not always possible to determine the exact type of an exception in a purely syntax-directed manner. We will, however, omit programmatic constructs that would hide the concrete class (for example, variables with exception type).

The syntax of this extended calculus, which we call $\text{Limp}^{\leq:}_{\mathcal{L}}$, is the same as the syntax for $\text{Limp}_{\mathcal{L}}$. The subtyping over exceptions is expressed in the semantics by parameterizing statement evaluation by the subtyping relation, denoted $\leq:$. The only requirements on this relation is that it must form a partial order. Figure 8 gives the new $\text{Limp}^{\leq:}_{\mathcal{L}}$ semantics for the **try** ... **catch** construct, which are the only notable differences to the semantics for $\text{Limp}_{\mathcal{L}}$. An exception is caught if the type of the exception is a subtype of the exception declared in the **catch** block (rule $\text{E}^{\leq:}$-CATCHCATCH). If the body of the **try** throws an exception C that is not a subtype of D, then the exception is propagated.

We can support exception subtyping by converting our flat exception environments into a stack of exception handlers. Exception environments are now described by the following type:

$$\text{Exception environment} \quad A \quad ::= \quad C \mapsto (l, T) :: A \mid \mathcal{F} \mapsto (l, -) :: A \mid \bullet$$

As before, T is the labeled type that binds the value carried by the exception and l is the program counter label of the catch handler. We denote **finally** handlers by the distinguished name $\mathcal{F}$, which does not carry a value since **finally** blocks do not bind the exception value.

Figure 9 presents the necessary changes to the type system to support exception subtyping. The changes are similar to mechanisms in the Jif type system to handle subtyping for exceptions. The primary difference is the additional exception propagation relation which walks the exception environment and checks that information flow is permitted at all handlers that might catch a particular exception. This relation has the following form:

$$\delta, C\ m \vdash_l A$$

which states that an exception with name C and value protected by label m can be thrown from a statement with program counter label l under the exception environment A and the label constraint environment δ.

Propagation stops when an exception reaches a handler that must catch it (P-MUSTCATCH). This requires that the propagating exception is a subtype of the handled exception. In this case, the label of the program counter at the throw site must be able to flow to the exception handler's program counter label. If the exception type of the handler is a subtype of the thrown exception type, then it may catch the exception so we must check against this handler as well as the rest of the chain (P-MAYCATCH). This rule uses $<:$ to mean a strict subclass of, i.e. $C <: D$ if $C \leq: D$ and $C \not\leq: D$.

If the exception type and the handler type are not related then we know that the exception won't be caught by this handler and it can be skipped (P-PASS). The P-FINALLY block raises the program counter label of the exception and checks the rest of the chain with the augmented exception. This corresponds to the case when the **finally** block terminates normally and the exception is re-thrown. Raising the program counter label accomplishes the same thing as not permitting the lower exceptions from being thrown since the $l \sqsubseteq l'$ requirement in P-MUSTCATCH and P-MAYCATCH will no longer hold. In the special case where a **finally** block can not terminate normally (P-FINALLYEND), we can ignore the rest of the chain since it will be checked when checking the **finally** block itself.

$\boxed{\Gamma, \delta, A, l' \vdash_l s \textbf{ ok}}$ **Exception subtyping type rules**

$$\frac{\begin{array}{c} x \notin \text{dom}(\Gamma) \\ \Gamma, \delta, C \mapsto (l_C, \textbf{int}\{l_e\}) :: A, l' \vdash_l s \textbf{ ok} \\ x \mapsto \textbf{int}\{l_e\} :: \Gamma, \delta, A, l' \vdash_{l_C} s_e \textbf{ ok} \end{array}}{\Gamma, \delta, A, l' \vdash_l \textbf{try } s \textbf{ catch } (C\ x)\ s_e \textbf{ ok}}\ \text{T-CATCH}$$

$$\frac{\begin{array}{c} \Gamma, \delta, A, l'' \vdash_l s_f \textbf{ ok} \\ \delta \vdash l'' \sqsubseteq l' \\ \delta \vdash A \backslash l'' = A' \\ \Gamma, \delta, \mathcal{F} \mapsto (l'', -) :: A', l' \vdash_l s \textbf{ ok} \end{array}}{\Gamma, \delta, A, l' \vdash_l \textbf{try } s \textbf{ finally } s_f \textbf{ ok}}\ \text{T-FINALLY}$$

$$\frac{\Gamma, \delta \vdash e : \textbf{int}\{l''\} \qquad \delta, C\ l'' \vdash_l A}{\Gamma, \delta, A, l' \vdash_l \textbf{throw } (C, e) \textbf{ ok}}\ \text{T-THROW}$$

$\boxed{\delta, C\ m \vdash_l A}$ **Exception Propagation Relation**

$$\frac{\begin{array}{c} \delta \vdash l \sqsubseteq l' \\ D \leq: C \qquad \delta \vdash m \sqcup l \sqsubseteq m' \end{array}}{\delta, C\ m \vdash_l D \mapsto (l', \tau\{m'\}) :: A}\ \text{P-MUSTCATCH}$$

$$\frac{\begin{array}{c} \delta \vdash l \sqsubseteq l' \\ C <: D \qquad \delta \vdash m \sqcup l \sqsubseteq m' \qquad \delta, C\ m \vdash_l A \end{array}}{\delta, C\ m \vdash_l D \mapsto (l', \tau\{m'\}) :: A}\ \text{P-MAYCATCH}$$

$$\frac{C \not\leq: D \qquad D \not\leq: C \qquad \delta, C\ m \vdash_l A}{\delta, C\ m \vdash_l D \mapsto (l', m') :: A}\ \text{P-PASS}$$

$$\frac{\delta, C\ m \vdash_{l \sqcup l'} A \qquad l' \neq \emptyset}{\delta, C\ m \vdash_l \mathcal{F} \mapsto (l', -) :: A}\ \text{P-FINALLY}$$

$$\frac{}{\delta, C\ m \vdash_l \mathcal{F} \mapsto (\emptyset, -) :: A}\ \text{P-FINALLYEND}$$

Figure 9. Typing rules for exception propagation with subtypes.

Using the exception propagation relation, modifications to the exception handling rules are mostly syntactic. T-CATCH is just adapted to use the new path map. T-FINALLY adds a finally marker to the top of the exception environment for checking s, with the label equal to the normal termination label of s_f. Finally, the flows test in T-THROW changes to the exception propagation relation with the appropriate parameters.

We are currently working on proving that this extended type system enforces non-interference.

7. Discussion

We have proven that our type system enforces non-interference and that it permits strictly more programs than the existing Jif type system (adapted for our calculus). It is important to note that this improvement in expressivity is not of purely theoretical interest. Our type system would help developers write provably secure code

$\boxed{\leq:, \Phi \vdash s, \sigma \to s, \sigma}$ **Limp$^{\leq:}_{\mathcal{L}}$ Semantics**

$$\frac{\leq:, \Phi \vdash s, \sigma \to s', \sigma'}{\leq:, \Phi \vdash \textsf{try}\, s\, \textsf{catch}\, (C\, x)\, s_c, \sigma \to \textsf{try}\, s'\, \textsf{catch}\, (C\, x)\, s_c, \sigma'}\ \text{E}^{\leq:}\text{-CatchStep}$$

$$\frac{C \leq: D}{\leq:, \Phi \vdash \textsf{try throw}\, (C, v)\, \textsf{catch}\, (D\, x)\, s_c, \sigma \to\ s_c, \sigma \uplus \{x \mapsto v\}}\ \text{E}^{\leq:}\text{-CatchCatch}$$

$$\frac{C \not\leq: D}{\leq:, \Phi \vdash \textsf{try throw}\, (C, v)\, \textsf{catch}\, (D\, x)\, s_c, \sigma \to \textsf{throw}\, (C, v), \sigma'}\ \text{E}^{\leq:}\text{-CatchPass}$$

$$\frac{}{\leq:, \Phi \vdash \textsf{try skip catch}\, (D\, x)\, s_c, \sigma \to \textsf{skip}, \sigma'}\ \text{E}^{\leq:}\text{-CatchSkip}$$

Figure 8. Small-step operational semantics for Limp$^{\leq:}_{\mathcal{L}}$.

without resorting to awkward coding idioms to convince the type system that the desired security property holds.

We note that the additional expressivity of our system is derived from the standard formalism that type systems are relations rather than computations. This re-formulation may have implications for the implementation which we have not yet had the chance to explore, especially in its interaction with the mechanism for inferring security labels in Jif. Label inference is essential to adapting existing Java code and even writing new Jif programs due to the verbosity of explicitly labeling variables.

The similarity of our type system with respect to almost all constructs except for exceptions is interesting. Specifically, note that the problem of maintaining all of the static information known by the system is only complicated by the non-local control associated with exceptions. If we remove exceptions, then the two type systems are equally expressive. This insight justifies our choice of a very minimal calculus which lacks even basic constructs such as loops and functions. Since the semantics of these constructs do not exhibit the kind of non-local control that exceptions introduce, we expect it to be relatively straightforward to adapt the existing Jif rules for our new type system.

While our formalization ignores the complexities introduced from potential non-termination [Sabelfeld and Myers 2003], it is not difficult to see how the rules could be extended to yield a termination sensitive or insensitive information flow analysis with the inclusion of loops. Neither timing channels nor concurrency are dealt with in our system, nor are they addressed by the Jif system, though existing work suggests several ways to deal with these problems [Smith and Volpano 1998].

7.1 Related Work

In addition to the background work on information flow type systems and the Jif language, our contributions draw on work from a variety of sources. Non-interference in the presence of first-class labels is studied by both Zheng and Myers [2004] and Grabowski and Beringer [2009]. They consider languages that permit dynamic checks on security labels, and show non-interference results. Grabowski and Beringer [2009] consider the possibility that the run-time security policy may be chosen after the program has been analyzed. This mechanism is closely related to the more general feature of dependent types. Seen in this way, run-time label tests can be viewed as a form of run-time-type analysis which is a basic component of the more general reflective programming techniques.

This insight is not new, Tse and Zdancewic [2007] define the λ_{RP} calculus which is modeled on the lambda calculus and supports run-time reflection on the **actsFor** relation, which is another part of the run-time security policy. They note that, while seemingly intertwined, reflection on **actsFor** and reflection on label tests are mostly orthogonal; this justifies our focus on security levels, and not on security principals. Supporting such an extension in our type system requires supporting, and reasoning about, **actsFor** facts in the constraint environment δ and adding principals that, like labels, need to be immutable.

Nanevski [2004] argues for a co-monadic formulation of exceptions rather than the, probably more well known, monadic formulation. This distinction parallels the shift from propagating exceptions outward (the monadic style) to propagating the capability to raise an exception (the co-monadic style).

8. Conclusions

We have described a new type system for an imperative calculus with named exceptions that enforces non-interference and is strictly more permissive than the existing Jif type system when restricted to our fragment. Our key insight is to relax the computational flavor the existing type system in favor of a constraint-based system that checks side conditions at the point of most knowledge. We proved our modified type system enforces non-interference. In addition, we discussed how our type system can be extended to handle a hierarchy of exceptions and the addition of a single path rule for safely lowering labels when statements can be proven to terminate in a single way.

Future Work

Our work suggests two interesting avenues for future work. The first is to determine the empirical cost of our constraint-based type system compared to the Jif type system. If our type system is proven feasible for our calculus it would be useful to extend out type system to handle the full Jif language. This would require extending the type system to handle looping constructs, including **break** and **continue**, functions, objects, and declassification as well as converting our simplified label model to the decentralized label model.

Once the type system addresses the full Jif language, it will be interesting to see the effect that it has on programming in Jif. We have shown that our type system permits more programs but we do not yet understand how our type system interacts with label inference or how it compares to the current type system in terms of being efficiently checkable.

References

Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages

43–59, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3712-2. doi: http://dx.doi.org/10.1109/CSF.2009.22.

Cadar, Cristian and Dunbar, Daniel and Engler, Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of OSDI*, 2008.

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596565.

Stephen Chong and Andrew C. Myers. Decentralized robustness. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 242–256, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2615-2. doi: http://dx.doi.org/10.1109/CSFW.2006.11.

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating System Principles*, pages 31–44, 2007a.

Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium*, pages 1–16. USENIX Association, August 2007b.

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society, May 2008.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 482–493, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: http://doi.acm.org/10.1145/1250662.1250722.

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977. ISSN 0001-0782. doi: http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/359636.359712.

David Endler. The evolution of cross site scripting attacks. Technical report, iDEFENSE Labs, 2002.

Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

Robert Grabowski and Lennart Beringer. Noninterference with dynamic security domains and policies. *13th Asian Computing Science Conference, Focusing on Information Security and Privacy*, 5913, 2009.

Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: http://doi.acm.org/10.1145/268946.268976.

Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. Understanding practical application development in security-typed languages. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, October 2009.

Gregory Malecha and Stephen Chong. A more precise security type system for dynamic security tests. Technical Report TR-05-10, Harvard University, May 2010.

Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 228–241, New York, NY, USA, January 1999. ACM Press.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

Alexander Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, Carnegie Mellon University, 2004.

François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems*, 25 (1):117–158, 2003. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/596980.596983.

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

Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/268946.268975.

Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Notices*, 41(1):372–382, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1111320.1111070.

Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.*, 30, November 2007. ISSN 0164-0925. doi: 10.1145/1290520.1290526. URL `http://portal.acm.org.ezp-prod1.hul.harvard.edu/citation.cfm?id=1290520.1290526`.

Andrew van der Stock, Jeff Williams, and Dave Wichers. OWASP top 10, 2007.

P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium (NDSS '07)*, February 2007.

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

Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.

Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Formal Aspects in Security and Trust*, Toulouse, France, August 2004.