

IBM Research Report

A Language for Information Flow: Dynamic Tracking in Multiple Interdependent Dimensions

Avraham Shinnar
Harvard University
Cambridge, MA

Marco Pistoia
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Anindya Banerjee
Kansas State University
Manhattan, KS



Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

A Language for Information Flow

Dynamic Tracking in Multiple Interdependent Dimensions

Avraham Shinnar¹, Marco Pistoia², and Anindya Banerjee³

¹ Harvard University, Cambridge, Massachusetts, USA, shinnar@eecs.harvard.edu

² IBM T. J. Watson Research Center, Hawthorne, New York, USA, pistoia@us.ibm.com

³ Kansas State University, Manhattan, Kansas, USA, ab@cis.ksu.edu

Abstract. This paper presents λ_I , a language for dynamic tracking of information flow across multiple, interdependent dimensions of information. Typical dimensions of interest are integrity and confidentiality. λ_I supports arbitrary domain-specific policies that can be developed independently. λ_I treats information-flow metadata as a first-class entity and tracks information flow on the metadata itself (integrity on integrity, integrity on confidentiality, etc.).

This paper also defines IMPOLITE, a novel class of information-flow policies for λ_I . Unlike many systems, which only allow for absolute-security relations, IMPOLITE can model more realistic security policies based on relative-security relations. IMPOLITE demonstrates how policies on interdependent dimensions of information can be simultaneously enforced within λ_I 's unified framework.

1 Introduction

This paper addresses the need for general information-flow systems that allow for expressive policy specifications. Security-enforcement mechanisms in existing commercial languages, such as Java and the Common Language Runtime (CLR), are imprecise and unsound [1]. Research systems, such as Jif [2], Flow Caml [3], and Information-Based Access Control (IBAC) [1], are sound, but restrict the class of policies that can be enforced. In particular, existing systems can only encode *absolute-security relations*—from the point of view of integrity, all the principals responsible for the value of an expression must be equally trusted with respect to any security-sensitive use of that expression, while in a confidentiality setting, it is only possible to control who has access to sensitive data, without being able to control who has access to the confidentiality policy itself. Additionally, static enforcement methodologies generally require the program to be statically labeled with information-flow-policy annotations—a significant burden on the developer, which may limit the portability of the program and restrict who can configure the information-flow policy of the program.

This paper introduces λ_I , a language that can precisely track information flow in multiple dimensions, such as integrity and confidentiality, without restricting the type of tracked data or the enforceable policies. Next, this paper

presents IMPOLITE, a class of policy-enforcement systems that can simultaneously enforce integrity and confidentiality policies on both the data manipulated by a program and the information-flow metadata kept by the systems.

IMPOLITE supports *relative-security relations*. From the point of view of integrity, different principals often have varying degrees of responsibility for a given value v . For example, a principal p may be responsible for having defined v , but the identity of p is only trusted up to the Certificate Authority a that signed p 's certificate. Systems that only support absolute-security relations typically require that not only p , but also a be sufficiently trusted to define v . Conversely, λ_I allows policies to make security decisions based on whether or not p is trusted to define v and a is trusted to certify p 's identity. Security decisions can be based on the history and structure of influences.

Access-control-based security models, such as those adopted by Java and the CLR, assign permissions to classes via class loaders [4]. In Java, every new Class c is assigned a `ProtectionDomain`, which encapsulates c 's permissions. A malicious class loader can easily escalate c 's privileges by assigning the provider of c (modeled as a `CodeSource` object cs) a `ProtectionDomain` with `AllPermission` in it, as follows:

```
new ProtectionDomain(cs, new AllPermission().newPermissionCollection())
```

The integrity level R of c is not constrained by the integrity level S of the class loader that assigned R to c . For this reason, the literature [5, 6] has emphasized that the permission q to instantiate a class loader is implicitly equivalent to `AllPermission` [4, Section 8.2.5]. Thus, any program using Java's built-in `URLClassLoader` to load code over the network—a very common operation—is implicitly granted `AllPermission`. The fact that c has integrity level R should only be trusted up to S —the integrity level of c 's class loader. To address this issue, unlike other programming languages, λ_I allows addressing information-flow metadata as information. For example, R is the integrity *metadata* on the underlying datum c and S is the integrity metadata on the integrity metadata R . In λ_I , this is written $S\{R\}\{c\}$, using *frames* [7–9]. Frame R denotes the integrity of c ; frame S denotes the integrity of $R\{c\}$.

Common systems are also incapable of using relatively-trusted integrity-enforcement mechanisms. An installed `SecurityManager` can enforce any policy it desires [4, Section 8.2.5] [6, Section 7.5.1]. Thus, a malicious implementation, by simply doing nothing, can make any permission check succeed:

```
public void checkPermission(Permission perm) {}
```

This is the same as granting `AllPermission` to arbitrary code! Clearly, security decisions need to be constrained by the enforcer's integrity level.

More generally, programs make decisions based on the integrity levels of the data they use. However, an intruder can affect a program *by influencing the*

integrity level of a value—not necessarily the value itself. Consider the case of a library method m that takes a parameter $A\ a$ and performs a callback, $a.f$. An intruder can choose to inject implementations of $a.f$ with different integrity levels, for example $R_1[a.f]$ or $R_2[a.f]$. On a subsequent security check involving $a.f$, R_1 may be sufficiently trusted, whereas R_2 may not. Thus, the intruder will have been able to influence the control flow of the program. λ_I handles this situation by *framing the frame* of $a.f$ with the frame A of the attacker, as in $A[R_1][a.f]$ and $A[R_2][a.f]$, tracking the influences on the metadata R_1 and R_2 , respectively. This problem can affect more than two levels of integrity since the values injected by the attacker may already have longer histories of influences, resulting in $A[R_1][S_1][a.f]$ and $A[R_2][S_2][a.f]$. This demonstrates the need for *potentially unbounded* (but finite) levels of framing.

Similarly, confidentiality levels may themselves need to be confidential, requiring multiple (unbounded) levels of frames. Integrity and confidentiality levels can also be interdependent; confidentiality levels can have integrity levels, which can have confidentiality levels, etc.

1.1 Contributions

Section 3 presents λ_I , an expressive language for dynamic information-flow tracking in multiple, interdependent dimensions. Information-flow tracking is built into λ_I , which allows programs to access and manipulate information-flow metadata. λ_I dynamically maintains security metadata throughout the execution of a program for subsequent policy decisions. Unlike previous work [8, 1], λ_I allows frames on frames, fully accounting for frames used as storage channels.

λ_I tracks information-flow dynamically, which potentially allows it to accept more programs than static systems—such as type-based information-flow systems—at the cost of greater overhead. Dynamic systems must be careful to address *implicit flows*: preventing an action can be as harmful as causing it. Handling these flows correctly requires the use of a *write oracle*, as discussed in Section 3.3. A write oracle essentially calculates to what locations code might write—a *modifies* set—which is required by many program-verification tools, such as Java Modeling Language (JML) [10].

λ_I separates a unified information-flow tracking mechanism from domain-specific policies via *lazy policy enforcement*. λ_I delays making policy decisions until interactions with the outside world arise. At that point, it presents the policy enforcer with a structured view of the relevant influences. A program is allowed to continue execution even when an untrusted value v_1 has influenced a value v_2 , which *may* be later used in a trusted computation, or when a confidential value v'_1 has influenced a value v'_2 , which *may* later become publicly observable. The policy will only reject the program if it tries to actually use v_2 or reveal v'_2 .

λ_I unifies the way information flow is tracked across domains, neither interpreting nor constraining the data or policies. In traditional systems, integrity and confidentiality are generally enforced by separate mechanisms, despite their well recognized duality [11, 12]. Additionally, non-lazy systems, generally constrain policies, requiring the labels to form a lattice [13]. λ_I treats integrity and confidentiality uniformly and does not constrain the allowable policies.

As discussed in Section 3.4, λ_I is a rich language that supports essential information-flow primitives such as endorsement and declassification. Additionally, λ_I can encode the Java `doPrivileged(Assert)` in the CLR and `doAs` constructs, which allow trusted code to ignore the permissions of its callers and run methods with different permissions, respectively.

Section 4 introduces Information Management POLicies in a Limited Trust Environment (IMPOLITE), a novel class of security policies, enforceable on λ_I , that allow relative-security relations on multiple interdependent dimensions of information. Existing systems, such as IBAC [1], can be modeled as instantiations of IMPOLITE. In IBAC, integrity labels are sets of permissions, and policy decisions treat all frames equivalently by taking their intersection. IMPOLITE supports relative-security relations, which can depend on the structure of the influences, whereas IBAC can only model absolute-security relations.

Section 4 states a non-interference result [14] for IMPOLITE. Appendix A presents a full proof, which includes conditionals, closures, and the heap.

1.2 The Attacker Model

We assume a *trusted-memory model*; untrusted intruders can read and write to memory, but the run-time system mediates all such accesses and maintains information-flow metadata for subsequent inspection at enforcement time. This allows us to support an *active-attacker model* [15]. Outside observers can inject code into a program and monitor its public interactions. From the point of view of integrity, trusted code can call untrusted code, and this can modify the heap. However, if this affects security-critical events, it will be detected. Dually, from the point of view of confidentiality, λ_I allows arbitrary code to read all of memory, but detects attempts to reveal secrets to outside public observers.

Our model is *timing-* and *termination-insensitive*; an outside observer can monitor system calls and the program's return value, but cannot measure time between these events or detect non-termination. We equate detected information-flow violations with non-termination. Modulo these restrictions, we prove in Appendix A that, for IMPOLITE policies, an active attacker cannot compromise the security of the system.

2 Motivating Example

Figure 1 models a medical-record scenario. A medical record is a structure where each field's value may have its own integrity and

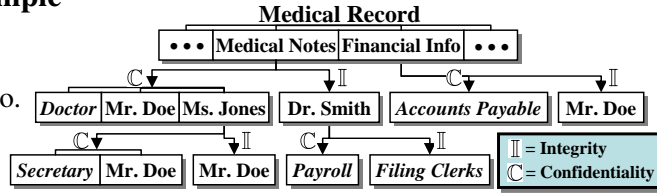


Fig. 1. Model of the Medical-Record Scenario

confidentiality requirements. Those requirements may in turn have their own integrity and confidentiality requirements, necessitating a systems that can model the complex interactions of multiple, interdependent dimensions of information.

In the scenario of Figure 1, Dr. Smith was seen by Mr. Doe. The resulting Medical Record datum contains several fields, including some Medical Notes and Financial Info. Integrity and confidentiality edges represent trust levels and privacy requirements, respectively. Principals written in italic represent roles.

Integrity. The value of the Medical Notes field has Dr. Smith's integrity stamp on it. We model this property as Dr. Smith{(Medical Notes)}. Similarly, the Financial Info was proffered by Mr. Doe, resulting in Mr. Doe{(Financial Info)}.

Confidentiality. In an emergency, Dr. Smith's Medical Notes must be accessible to every *Doctor*. Mr. Doe may access the Medical Notes, and has also chosen to grant his fiancée, Ms. Jones, access to the Medical Notes. The resulting confidentiality label is a structure with three fields: *Doctor*, Mr. Doe, and Ms. Jones.

Confidentiality on Integrity. If Dr. Smith is an HIV specialist, knowing that he was consulted could lead people to infer that Mr. Doe is HIV positive. Thus, it is necessary to protect the Medical Notes with a confidentiality label, *Payroll*.

Integrity on Integrity. The integrity label on the Medical Notes is *Filing Clerks*, as they certify that the Medical Notes were submitted by Dr. Smith.

Confidentiality on Confidentiality. Mr. Doe may not want his relatives to know that he has granted Ms. Jones access to the Medical Notes. Thus, the Ms. Jones confidentiality label is itself confidential, with a structured label containing fields *Secretary* and Mr. Doe.

Integrity on Confidentiality. The Ms. Jones confidentiality label has an integrity level of Mr. Doe, as he granted her access to the Medical Notes.

3 Language

In this section, we present a language that provides primitives for tracking and manipulating information-flow metadata. As discussed in Section 1.2, our system mediates all access to memory, which is assumed local. Policies are *lazily*

Variables	x, m		
Field Names (\mathcal{F})	f		
Commands (\mathcal{C})	C	$::= v \mid \mathbf{let} \ x = C \ \mathbf{in} \ C \mid \mathbf{ref} \ v \mid !v \mid v := v \mid$ $\mathbf{true} \mid \mathbf{false} \mid n \mid \mathbf{unit} \mid \mathbf{null} \mid$ $\mathbf{struct} \ \{\bar{f} = \bar{v}\} \mid v.f \mid$ $\mathbf{frame} \ v \ \mathbf{with} \ v \mid \mathbf{frameof} \ v \mid \mathbf{valueof} \ v \mid$ $\mathbf{fix} \ m \ x \Rightarrow C \mid v \ v \mid \mathbf{if} \ v \ \mathbf{then} \ C \ \mathbf{else} \ C$	values, let, refs primitive values records frames functions, conditionals
Atomic Den. (\mathcal{A})	a	$::= \mathbf{true} \mid \mathbf{false} \mid n \mid \mathbf{unit} \mid \mathbf{null} \mid \ell \mid \langle m, x, C \rangle$	primitives, locations, closures
Denotations (\mathcal{D})	d, R	$::= a \mid \{\bar{f} = \bar{d}\} \mid R[a]$	atomics, structs, frames
Value (\mathcal{V})	v	$::= x \mid d$	variables, denotations

Fig. 2. Frame Language Syntax

enforced immediately prior to a security-sensitive event. These policies are part of the general system; they are not specified by the language.

We first present a core language, λ_F , for manipulating the metadata, and then define the full language, λ_I , via a translation to λ_F . This separation helps provide a clean interface to the metadata.

3.1 Core Frame Language: λ_F

Figure 2 presents the syntax of λ_F , an A-normalized language with references, structures, conditionals, and recursive first-class functions. λ_F also includes frames, a mechanism for associating information-flow metadata with data. λ_F provides primitives to manipulate these frames, but does not enforce policies on them. A frame can be any denotation, and frames can themselves be framed. Denotations include atomic denotations, structures, and framed constructs. Framed constructs, as seen in Figure 2, are *canonicalized* according to the rules in Figure 3(a) so that only atomic denotations are framed. Canonicalization emphasizes the *underlying data*, ascribing a clear, useful meaning to constructs such as $R[S][3] + T[4]$. Note that canonicalization could be done explicitly by the **frameof** rule, but is broken out as separate rules to simplify the theory.

As formalized by the Null Absorption property, *null* is used as a form of frame terminator, and is absorbed by framing. Struct Lifting is an assumption common to many languages, including ML: denotations are identical to the singleton tuple (in our language a simple struct) containing that denotation.

The Struct Lifting and Distribution properties of Figure 3(a) imply a form of associativity on frames, $d_1[d_2][d_3] \equiv d_1[d_2[d_3]]$. This interpretation is compatible with the ABLP calculus for access control in distributed systems by Abadi, et al. [16] as will be discussed in Section 5.

PRIMITIVES $(\mathbf{true}, h) \Downarrow (\mathbf{true}, h) \dots$	FRAMING $(\mathbf{frame } d \mathbf{ with } R, h) \Downarrow (R[d], h)$	VALUE PROJECTION $(\mathbf{valueof } R[d], h) \Downarrow (d, h)$
FRAME PROJECTION $(\mathbf{frameof } R[d], h) \Downarrow (R, h)$	STRUCTURE CREATION $(\mathbf{struct } \{\bar{f} = \bar{d}\}, h) \Downarrow (\bar{f} = \bar{d}, h)$	FIELD PROJECTION $(\{f_i = d_i\}.f_i, h) \Downarrow (d_i, h)$
REF $\frac{\ell \notin \text{dom}(h) \quad \text{passive } d}{(\mathbf{ref } d, h) \Downarrow (\ell, [h \mid \ell \mapsto d])}$	ASSIGNMENT $\frac{\text{passive } d}{(\ell := d, h) \Downarrow (\text{unit}, [h \mid \ell \mapsto d])}$	DEREFERENCING $(!R[\ell], h) \Downarrow ((h \ \ell), h)$
IF TRUE $\frac{(C_1, h) \Downarrow (d, h') \quad \text{passive } d}{(\mathbf{if } R[\mathbf{true}] \mathbf{ then } C_1 \mathbf{ else } C_2, h) \Downarrow (d, h')}$	IF FALSE $\frac{(C_2, h) \Downarrow (d, h') \quad \text{passive } d}{(\mathbf{if } R[\mathbf{false}] \mathbf{ then } C_1 \mathbf{ else } C_2, h) \Downarrow (d, h')}$	
LET $\frac{(C_1, h) \Downarrow (d_1, h_1) \quad (C_2[d_1/x], h_1) \Downarrow (d, h')}{(\mathbf{let } x = C_1 \mathbf{ in } C_2, h) \Downarrow (d, h')}$	METHOD APPLICATION $\frac{C[d/x][\langle m, x, C \rangle/m], h) \Downarrow (d', h')}{(R[\langle m, x, C \rangle] d, h) \Downarrow (d', h')}$	
	METHOD DEFINITION $(\mathbf{fix } m \ x \Rightarrow C, h) \Downarrow (\langle m, x, C \rangle, h)$	

NULL ABSORPTION: $null[\langle d_1 \rangle][\langle null \rangle][\langle d_2 \rangle] \equiv d_1[\langle d_2 \rangle]$

STRUCT LIFTING: $d \equiv \{.1 = d\}$

DISTRIBUTION: $d[\langle \{\bar{f}_i = \bar{d}_i\} \rangle] \equiv [\langle \{\bar{f}_i = \bar{d}[\langle d_i \rangle] \} \rangle]$

(a) Frame Canonicalization

passive $d \equiv d$ contains no closures.

(b) **passive** Denotations

Fig. 3. Frame Language Semantics

Throughout the paper, we will write $d_1[\langle d_2 \rangle]$ for all denotations d_2 , and assume canonicalization is implicitly performed as per Figure 3(a).

Figure 3 describes the semantics of λ_F . $d[d_1/x]$ is standard capture avoiding substitution of d_1 for x in d . λ_F provides recursive closures as a form of `fix`. **frame with**, **frameof**, and **valueof** are the introduction and elimination forms for framed constructs. **passive**, as expressed in Figure 3(b), is a predicate on denotations indicating the absence of closures within them. Only **passive** denotations can be put in the heap and returned from conditionals. Additionally, writes are restricted to *bare* (unframed) locations. These restrictions will be explained and motivated in Section 3.2, which introduces λ_I .

3.2 Information-Flow Language: λ_I

Figure 4 defines λ_I by translation to λ_F . λ_I compiles to λ_F by tracking control- and data-flow, framing every value with its dependencies. It includes all of the λ_F commands and a few more, as described in Figure 4(a). *Programs* (top-level commands) are translated by prepending two let bindings for special heap locations, *pc* and *meth*, created and used by the system, as defined in Figure

<p style="text-align: center;">Commands $C(\lambda_I \supset \lambda_F) ::= \dots \mid \mathbf{getpc}$ $\mid \mathbf{getmeth} \mid v.\bar{f} := v \mid \mathbf{assert} R \text{ in } C$</p> <p style="text-align: center;">(a) Information Flow Language Syntax</p>	<p style="text-align: center;">$T_p(C) = \mathbf{let} pc = \mathbf{ref} \text{ null in}$ $\mathbf{let} meth = \mathbf{ref} \text{ null in } \llbracket C \rrbracket$</p> <p style="text-align: center;">(b) Translation of the top level program</p>
$\llbracket v \rrbracket \equiv v$ $\llbracket \mathbf{frameof} v \rrbracket \equiv \mathbf{frameof} v$ $\llbracket \mathbf{fix} m x \Rightarrow C \rrbracket \equiv \mathbf{fix} m x \Rightarrow \llbracket C \rrbracket$ $\llbracket \mathbf{let} x = C_1 \text{ in } C_2 \rrbracket \equiv \mathbf{let} x = \llbracket C_1 \rrbracket \text{ in } \llbracket C_2 \rrbracket$ $\llbracket \mathbf{ref} v \rrbracket \equiv \mathbf{ref} \text{ taint}(v)$ $\llbracket \mathbf{valueof} v \rrbracket \equiv \mathbf{valueof} v$ $\llbracket v.f \rrbracket \equiv v.f$ $\llbracket \mathbf{getpc} \rrbracket \equiv !pc$ $\llbracket !v \rrbracket \equiv \mathbf{frame} !v \text{ with } (\mathbf{frameof} v)$ $\llbracket \mathbf{frame} v_1 \text{ with } v_2 \rrbracket \equiv \mathbf{frame} v_1 \text{ with } v_2$ $\llbracket \mathbf{struct} \{ \bar{f} = \bar{v} \} \rrbracket \equiv \mathbf{struct} \{ \bar{f} = \bar{v} \}$ $\llbracket \mathbf{getmeth} \rrbracket \equiv !meth$ $\llbracket v_1.f_1.f_2 \dots f_n := v_2 \rrbracket$ $\equiv v_1 := \mathbf{taint_field}(!v_1, [f_1, f_2, \dots, f_n], v_2)$	$\llbracket v_1 v_2 \rrbracket \equiv \mathbf{let} R = \mathbf{frameof} v_1 \text{ in}$ $\mathbf{let} v'_2 = \mathbf{frame} v_2 \text{ with } !meth \text{ in}$ $\mathbf{set}(meth = R) \text{ in}$ $\mathbf{set}(pc = \text{taint}(R)) \text{ in}$ $\mathbf{frame} (v_1 v'_2) \text{ with } R$ $\llbracket \mathbf{assert} R \text{ in } C \rrbracket \equiv$ $\mathbf{let} R' = \mathbf{frame} R \text{ with } !meth \text{ in}$ $\mathbf{set}(pc = R') \text{ in } \llbracket C \rrbracket$ $\llbracket \mathbf{if} v \text{ then } C_1 \text{ else } C_2 \rrbracket \equiv$ $\mathbf{let} R = \mathbf{frameof} v \text{ in}$ $\mathbf{set}(pc = \text{taint}(R)) \text{ in}$ $\mathbf{let} y = \mathbf{if} v \text{ then } \llbracket C_1 \rrbracket \text{ else } \llbracket C_2 \rrbracket \text{ in}$ $\mathbf{tainter}(\text{wo}(C_1) \cup \text{wo}(C_2));$ $\mathbf{frame} y \text{ with } R$

With the following helper functions:

$C_1; C_2 \equiv \mathbf{let} _ = C_1 \text{ in } C_2$ $\mathbf{set}(var = R) \text{ in } C \equiv \mathbf{let} old = !var \text{ in}$
 $\mathbf{taint}(v) \equiv \mathbf{frame} v \text{ with } !pc$ $var := \mathbf{frame} \text{ null with } R; \mathbf{let} ret = C \text{ in } var := old; ret$
 $\mathbf{taint_field}(v_1, f :: fl, v_2) \equiv \mathbf{let} \bar{x}_k = \overline{v_1.f_k} \text{ in}$ $\mathbf{taint_field}(v_1, [], v_2) \equiv \mathbf{taint}(v_2)$
 $\mathbf{let} z = \mathbf{taint_field}(x_i, fl, v_2) \text{ in } \mathbf{struct} \{ f_0 = x_0, \dots, f_{i-1} = x_{i-1}, f_i = z, f_{i+1} = x_{i+1}, \dots \}$
 $\mathbf{tainter}(set) \equiv \mathbf{foldl} (_ ; \text{unit} (\lambda(v, fl) \rightarrow v := \mathbf{taint_field}(v, fl, !v.f_1 \dots f_n)) set)$

Fig. 4. Information Flow Language \rightarrow Frame Language Translation

4(b). pc is used to track the current control dependencies. $meth$ records the frame of the currently executing function; it is changed upon function invocation and restored upon function return. The translation then proceeds recursively on the program's command, propagating influences as needed. For example, the **ref** rule *taints* (meaning, frames with pc) the value being written to memory, recording the influence of the current control dependencies. Dereferencing a location frames the looked-up value with the frame of the location.

Influence is relative. In Figure 3.2, `getName` completely trusts the string `log`. Method `g` trusts the string as much as it trusts `getName`. This trust is independent of `g`'s callers; `f` does not influence the trust level. The application rule of λ_I frames the function's argument with the frame of the caller (its $meth$ frame) and the return value with the frame of the invoked function.

```

f () { g (); }
g () {
  name=getName ();
  ... }
getName () {
  return "log"; }

```

Fig. 5. Relative Trust

λ_I provides access to the underlying **frame with**, **frameof**, and **valueof** commands in λ_F , allowing code to access and manipulate the information-flow metadata. In particular, this allows code to endorse

and declassify data in integrity and confidentiality environments, respectively. λ_I also adds in some new commands. **assert in** allows the programmer to explicitly ignore control dependencies. This is akin to `doPrivileged` in Java and `Assert` in the CLR. If a library wants to write to a log file, it can use **assert in** to do so, even if the client does not have the required permissions. **getpc** and **getmeth** help the programmer selectively ignore *some* control dependencies.

λ_I also adds in an assignment command that modifies *part* of a structure. As witnessed by the translation, this can be done in λ_F . If this command were not provided, however, its encoding would conservatively taint the entire structure.

3.3 Restrictions

To ensure soundness, λ_I restricts conditionals, closures, and the heap.

Conditionals and the Heap: The Write Oracle. In Figure 3.3, if an attacker can set `b` to *false*, `Shire` will be destroyed instead of `Mordor`. To prevent this *indirect flow* of information, the translation in

```
location = "Shire";
if (b) location="Mordor";
destroy(location);
```

Fig. 6. Branch Not Taken

`wo(C)` returns a set that contains a pair (v, \bar{fl}) for each path $v.f_1 \dots f_n$ that C may modify. `wo` may be conservative

Fig. 7. Write Oracle: `wo`

Figure 4(a) employs an online write oracle `wo`, defined in Figure 3.3. `wo` returns the locations that may be written by a conditional's branches. The translation marks these locations as influenced by `b`, correctly accounting for these indirect flows.

For expository purposes, Figure 4 simplifies the way λ_I uses `wo` because locations modified in a conditional will be re-tainted by `wo`. If we track modified locations in *written*, we only want to taint $(wo(C_1) \cup wo(C_2)) \setminus written$. If `wo` were perfect, and C_2 is the branch not taken, we could just taint $wo(C_2) \setminus written$. However, given a conservative (but still sound) `wo`, tainting $(wo(C_1) \cup wo(C_2)) \setminus written$ is necessary. Consider in fact a `wo` for Figure 3.3 that believes that the *true* branch writes to another variable, `tower`. If `b` is *true*, `tower` will not be tainted; if `b` is *false*, `tower` will be tainted by `wo`. Thus, by controlling `b`, an attacker could influence the frame on the value of `tower` without detection.

The Heap: Bare (Unframed) Locations. λ_F , and hence λ_I , disallows writes to framed locations, obviating the need to taint most of the heap to correctly account for indirect flows. A location must only be trusted by the writing code, forcing code to *endorse* (with **valueof**) locations before writing to them.

Closures, the Heap and Conditionals: Passivity. λ_F , and hence λ_I , also prevent closures from being in the heap or returned from conditionals. An attacker can prevent a closure in the heap from being invoked by overwriting it, and

similarly for conditionals. Figure 3.3 presents pseudo-code for a command that returns one of two closures, depending on b . If trusted code invokes the returned function, then by falsifying b an attacker can cause n to not be written, a potential information-flow violation. `wo` does not return n , as n is not modified in the conditional. A similar problem exists for the heap, which necessitates the `passive` restriction on denotations in the heap. If a location l has `(fix a x => n := 5)` stored in it, and an attacker overwrites l with `(fix a x => unit)`, then when trusted code dereferences and invokes l , n will retain its old value, but the attacker’s influence would not be tracked.

```
if(b) (fix a x=>n:=5)
else (fix a x=>unit)
```

Fig. 8. `if` and Closures

3.4 Properties

The Frame Canonicalization rules of Figure 3(a) succinctly describe the interplay between frames and structures, allowing λ_I to track multiple dimensions of information in an interdependent fashion. Structures can also be used to encode structured information. A `plus` function, given arguments $A\{3\}$ and $B\{4\}$ could return $\{part1 = A, part2 = B\}\{7\}$, encoding the set containing A and B .

Using `valueof`, λ_I can model many useful security-related primitives, such as *endorsement* and *declassification*, which allow trusted code to trust an (otherwise) untrusted value and to reveal private data to public parties, respectively.

Another common primitive in security-related systems (see, for example, Java’s `doAs` [4]) allows executing a method with an authenticated principal’s permissions. In λ_I , this is encoded by endorsing a closure with the principal’s permissions. When invoked, the closure will run with the altered permissions.

Section 1 discussed how existing systems, such as Java and the CLR, are unable to provide relatively-trusted class loaders or security enforcers. While it does not support the object model used in those examples, λ_I does solve the problems they illustrate. When modeled in λ_I , a class-loader’s permission assignments will be framed by its integrity level. Similarly, λ_I allows anyone to install a `SecurityManager`, but that will be framed with the integrity level of its setter. In λ_I , an access-control test would be encoded using `frameof` and conditionals, and the system automatically tracks all the dependencies.

4 IMPOLITE: A Novel Security Model

In this section, we present the IMPOLITE class of policies.⁴ The goal is to test diagrams such as Figure 1 against a specified policy. For IMPOLITE, we introduce

⁴ Some mathematical conventions: unbound variables are assumed to be quantified with a top level \forall . Types are omitted where easily inferable.

dimensions of information. `joiner` recurses on frames. When given a framed denotation, `joiner` checks for validity of the contents with respect to f_{cur} (what it, in turn, is framing), and then uses `tester` to recurse on the frame. When given a structure, it uses `joiner` to test all of the structure's components, and then uses `join` to determine the result. Structures are used to encode dimensions of information as well as the internal complexity of the information. The mutual recursion arises from this alternating use of structures in frames, and differentiates between these usages. When the structure represents the different dimensions of information, the recursion is in `tester`; when the structure represents the internal complexity of the information, the recursion is in `joiner` (and uses `join` to parameterize how the internal structure is interpreted).

In the scenario, `tester` would be invoked on the structures that encode different dimensions, whereas `joiner` would be invoked on the aggregate $\{part1 = Doctor, part2 = Mr. Doe, part3 = Ms. Jones\}$.

To state the non-interference result, we need the following definition:

Definition 4 (Completely Invalid).

$(f, a) \in \mathcal{F} \times \mathcal{A}$ is completely invalid iff $(f, a) \not\models^* (" ", null)$.

$(f, R) \in \mathcal{F} \times \mathcal{D}$ is completely invalid iff all of its parts are completely invalid.

\models^* is the reflexive transitive closure of \models

In a relative-security-relation environment, non-interference does not generally hold. By definition, a relative-security relation depends on unproven assumptions about the behavior of the code. This would involve some sophisticated form of program verification and is beyond the scope of our work. Nevertheless, we can prove non-interference for a stylized program which only allows for attackers that are framed by a *completely invalid* denotation. Nothing these attackers do will ever pass the IMPOLITE test.

Definition 5 (Safe). `safe C` iff `C` does not contain **valueof**, **frameof**, **assert in**.

valueof, **frameof**, and **assert in** can all be used to violate information flow. **valueofin** in particular, allows the core system to forget attacker influences. Endorsement and declassification, common forms of (deliberate) information-flow violations in integrity and confidentiality contexts, respectively, can be built on top of **valueof**. **frameof** allows code to detect the influences on a value. The following example illustrates the possible violations that can arise. An attacker can choose to write to a location in memory. The trusted code can then use **frameof** to make a decision based on whether or not the attacker tainted the location stored in that value. Note that using the value itself is not a problem, but **frameof** allows the trusted code to detect that the attacker chose to write something to the location, leaking a single bit of information. **assert v in C'**

allows core code to ignore that it was called (and hence influenced) by an attacker. If the code writes to a location in memory, then this allows the attacker to cause that write to happen without the attacker’s influence being tracked.

We can now formulate our non-interference result, proven in Appendix A.

Theorem 1 (Non-Interference). *Consider a safe (Def 5) λ_I command C , completely invalid (Def 4) denotations U_i, U'_i, U''_i, U'''_i , and commands C_i, C'_i .*

Suppose
$$\begin{aligned} & (\llbracket \text{let } \bar{x}_i = \text{frame } (\text{assert } \overline{U}_i \text{ in } \overline{C}_i) \text{ with } \overline{U}''_i \text{ in } C \rrbracket, \emptyset) \downarrow (d_1, h_1) \\ & (\llbracket \text{let } \bar{x}_i = \text{frame } (\text{assert } \overline{U}'_i \text{ in } \overline{C}'_i) \text{ with } \overline{U}'''_i \text{ in } C \rrbracket, \emptyset) \downarrow (d_2, h_2) \end{aligned}$$

Then, up to locations, $\text{test } d_1 \wedge \text{test } d_2 \implies \text{valueof } d_1 = \text{valueof } d_2$

We look at two runs of a core program C , in different environments. We assume that completely-invalid attackers are allowed to set up the initial environments in the two runs. Note that attacker code (and any closures it puts into the environment) can be written in the full language; they do *not* have to be *safe*. The theorem then states that if the IMPOLITE test succeeds in both cases, then the underlying values were not influenced by the attacker code.

As described above, **valueof**, **frameof**, and **assert in** potentially violate information flow, and Theorem 1 therefore only applies to programs where the trusted code is *safe*. To be more precise, if using one of these operations affects the final result, the information-flow policy may be violated, otherwise there is no problem. To formalize this, assume the existence of a completely-invalid ϵ . **valueof** and **frameof** could be modified to taint their return value with ϵ , and **assert in** to taint pc with ϵ . Theorem 1 would then hold without needing any restriction on the safety of the trusted code. This would allow information-flow violations as long as they have no influence on the final result, using the information-flow-tracking mechanism of λ_I to precisely identify and check for possible influences.

5 Related Work

Since Denning and Denning [17], there has been a large volume of work on static checking of information flow policies [18]. Goguen and Meseguer [14] introduce non-interference based on earlier work by Cohen [19]. Volpano, et al. [20] are the first to show a type-based algorithm that certifies implicit and explicit flows and also guarantees non-interference. Most of these works focus on confidentiality. Integrity is explored by Li, et al. [21]. Based on the premise that many software attacks subvert the execution of machine code, Abadi et al. perform and develop a comprehensive study of control-flow integrity [22].

Myers’ Jif [2] and Pottier and Simonet’s Flow Caml [3] use type-based static analysis to track information flow. Neither Jif nor Flow Caml allows simultaneous tracking of interdependent dimensions of information. Jif is based on the

Decentralized Label Model [23]. Section 1 has already discussed a few key differences between Jif and our system. Another difference is that Jif considers all memory as a channel of information, which requires that every variable, field, and parameter used in the program be statically labeled. Labels can either be declared or inferred. In contrast, λ_I 's memory model allows for a core trusted memory, which does not act as a channel and saves the programmer from the burden of labeling channels. This is a realistic model assuming that the operating system enforces memory protection across processes. Unlike static-enforcement systems, λ_I only requires labeling channels that represent communications with the outside world. Furthermore, λ_I supports a very flexible policy-enforcement mechanism, with arbitrary values as labels, and arbitrary information-flow tests.

Pistoia, et al. [1] describe IBAC, a unified access-control and information-flow system that uses access-control labels for information flow. The IBAC language supports a subset of the features available in λ_I . IBAC's non-interference can be viewed as an instantiation of the IMPOLITE non-interference result, with a restricted `validfor` relation in which (1) `join` is \wedge , and (2) if a frame S is `validfor` another frame R , then it is `validfor` all frames. Thus, IBAC does not support any relative-security relationships. The use of the intersection operator by IBAC can be viewed as a policy-driven optimization of this test.

With robust declassification, Myers, et al. enforce the principle that only high-integrity data be declassified, and declassification be performed only in high-integrity contexts [15]. Qualified robustness provides an attacker a limited ability to affect what information may be released by programs [24]. An `endorse` primitive is used to upgrade the integrity of data. The RX language allows integrity and confidentiality metapolicy labels on roles [25, 26]. Deeper interactions between integrity and confidentiality are not investigated.

Le Guernic, et al. [27] consider dynamic, automaton-based, monitoring of information flow for a single execution of a sequential program. The mechanism is based on a combination of dynamic and static analyses. The dynamic analysis accepts or rejects a single execution of a program without necessarily doing the same for all other executions. The automaton guarantees confidentiality of secret data and takes into account both direct and implicit flows. The static analysis overapproximates implicit indirect flows and generates corresponding branch-not-taken inputs to the automaton—similar to `wo` in our semantics. That work has been recently extended to handle concurrent programs [28].

Abadi, et al. [16] introduce the *says* and *quotes* relations, and prove a form of associativity between them. This is similar to the frames in our system and the associativity property of our Frame Canonicalization. The *speaks-for* relation introduced by them is an instance of the `validfor` relation used by IMPOLITE.

6 Discussion

This paper presented λ_I , a language for dynamic tracking of information flow in multiple, interdependent dimensions. We will implement λ_I and apply it to complex systems. A promising research direction is to apply λ_I to other dimensions of information, such as non-repudiation, provenance, and concurrency.

We are also interested in exploring policy-driven optimizations. λ_I is less efficient than more specialized, less expressive systems since it needs to maintain the entire structure of every frame. For a given policy, it should be possible to automatically optimize λ_I 's tracking mechanism. For example, IBAC only needs to maintain the intersection of all the frames. This would be useful in a Just In Time compiler environment, as the policy would already be known.

References

1. Pistoia, M., Banerjee, A., Naumann, D.A.: Beyond Stack Inspection: A Unified Access Control and Information Flow Security Model. In: 28th IEEE Symposium on Security and Privacy, Oakland, CA, USA (May 2007) 149–163
2. Myers, A.C.: JFlow: Practical Mostly-static Information Flow Control. In: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999), San Antonio, TX, USA (January 1999) 228–241
3. Flow Caml: <http://cristal.inria.fr/~simonet/soft/flowcaml/>.
4. Pistoia, M., Nagaratnam, N., Koved, L., Nadalin, A.: Enterprise Java Security. Addison-Wesley, Reading, MA, USA (February 2004)
5. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 Platform Security: Architecture, API Design, and Implementation. Second edn. Addison-Wesley, Reading, MA, USA (May 2003)
6. Pistoia, M., Reller, D., Gupta, D., Nagnur, M., Ramani, A.K.: Java 2 Network Security. Second edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (August 1999)
7. Sewell, P., Vitek, J.: Secure Composition of Untrusted Code: Wrappers and Causality Types. In: 13th IEEE Computer Security Foundations Workshop (CSFW 2000), Cambridge, England, UK (July 2000) 269–284
8. Fournet, C., Gordon, A.D.: Stack Inspection: Theory and Variants. In: 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), Portland, OR, USA, ACM Press (January 2002) 307–318
9. Grossman, D., Morrisett, J.G., Zdancewic, S.: Syntactic Type Abstraction. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(6) (2000) 1037–1080
10. Java Modeling Language (JML): <http://www.eecs.ucf.edu/~leavens/JML/>.
11. Biba, K.J.: Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, USA (4 1977)
12. Bell, D.E., LaPadula, L.: Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, MITRE Corporation, Bedford, MA, USA (1973)
13. Denning, D.E.: A Lattice Model of Secure Information Flow. Communications of the ACM **19**(5) (May 1976) 236–243
14. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, IEEE Computer Society Press (May 1982) 11–20

15. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing Robust Declassification. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14 2004), Pacific Grove, CA, USA, IEEE Computer Society (June 2004) 172–186
16. Abadi, M., Burrows, M., Lampson, B.W., Plotkin, G.D.: A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **15**(4) (1993) 706–734
17. Denning, D.E., Denning, P.J.: Certification of Programs for Secure Information Flow. *Communications of the ACM* **20**(7) (July 1977) 504–513
18. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* **21**(1) (January 2003) 5–19
19. Cohen, E.S.: Information Transmission in Sequential Programs. In DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J., eds.: *Foundations of Secure Computation*, Academic Press (1978) 297–335
20. Volpano, D., Irvine, C., Smith, G.: A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* **4**(2-3) (January 1996) 167–187
21. Li, P., Mao, Y., Zdancewic, S.: Information Integrity Policies. In: *Workshop on Formal Aspects in Security and Trust (FAST 2003)*, Pisa, Italy (September 2003)
22. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity: Principles, Implementations, and Applications. In: *12th ACM Conference on Computer and Communications Security (CCS 2005)*, Alexandria, VA, USA, ACM (2005) 340–353
23. Myers, A.C., Liskov, B.: A Decentralized Model for Information Flow Control. In: *16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, Saint-Malo, France (October 1997) 129–142
24. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security* **14**(2) (May 2006) 157–196
25. Swamy, N., Hicks, M., Tse, S., Zdancewic, S.: Managing Policy Updates in Security-Typed Languages. In: *19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*, Venice, Italy, IEEE Computer Society (July 2006) 202–216
26. Hosmer, H.H.: Metapolicies I. *SIGSAC Review* **10**(2-3) (1992) 18–43
27. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-based Confidentiality Monitoring. In: *Proceedings of 11th Annual Asian Computing Science Conference (ASIAN 2006)*, Tokio, Japan (December 2006)
28. Le Guernic, G.: Automaton-based Confidentiality Monitoring of Concurrent Programs. In: *20th IEEE Computer Security Foundations Symposium (CSF-20 2007)*, Venice, Italy, IEEE Computer Society (July 2007)

A Non-Interference for IMPOLITE

We present here a proof of non-interference, Theorem 1, via some relations with convenient properties including *Reflexivity*, *Symmetry*, and *Transitivity*.

Definition 6 (fails). For some (join, \models) :

$$\text{fails } d \equiv \neg \text{test } d \quad \text{where } \text{join} = \text{join and } (\models) = (\models \cup \models^+)$$

\models^+ is \models 's transitive closure restricted to elements of the form $(f, d) \models^* (\text{null}, d')$

Lemma 1 (Properties of fails).

- *Atomic Success:* $\neg \text{fails } a$
- *Atomic Success:* $\text{fails } d\{a\} = \text{fails } d\{a'\}$
- *Left Extension:* $\text{fails } v \implies \text{fails } R\{v\}$

- *Right Extension*: $\text{fails } R[a] \implies \text{fails } R[S][a]$
- *Test Failure*: $\text{fails } d \implies \neg \text{test } d$
- *Completely Invalid*: $d \text{ completely invalid} \implies \text{fails } d[\text{null}]$.

Remark 1. β will be a partial injection between heap domains.

Definition 7 (Substituting free locations under β : $d/h/C[\beta]$). $d[\beta]$ is the substitution of locations in d under β (β total on the free locations in d). This is extended to heaps (by substitution on the domain and range), and commands.

Definition 8 (Failure Preservation (on \mathcal{D}): $d_1 \dashv\vdash d_2$). (d_2 failure preserves d_1)

ATOMS $a_1 \dashv\vdash a_2$	STRUCTS $\frac{\forall i, d_i \dashv\vdash d'_i}{\{f_i = d_i\} \dashv\vdash \{f_i = d'_i\}}$	FRAMES $\frac{\text{fails } d_1 \implies \text{fails } d_2 \quad R_1 \dashv\vdash R_2}{R_1[d_1] \dashv\vdash R_2[d_2]}$	FAILURE $\frac{\text{fails } d_2}{d_1 \dashv\vdash d_2}$
---------------------------------	---	--	---

Lemma 2 (Properties of Failure Preservation(on denotations)).

- *R, T, • Failure Preservation*: $\text{fails } d_1 \wedge d_1 \dashv\vdash d_2 \implies \text{fails } d_2$

Definition 9 ($\mathcal{R}_{\mathcal{D}}$: β -indistinguishable denotations). $\mathcal{R}_{\mathcal{D}}$ holds when two denotations are either the same (modulo β) or will ultimately fail the final test.

ATOMS $\mathcal{R}_{\mathcal{D}} \beta a \ a[\beta]$	STRUCTS $\frac{\forall i, \mathcal{R}_{\mathcal{D}} \beta d_i \ d'_i}{\mathcal{R}_{\mathcal{D}} \beta \{f_i = d_i\} \ \{f_i = d'_i\}}$	FRAMES $\frac{\mathcal{R}_{\mathcal{D}} \beta P_1 \ P_2 \quad \mathcal{R}_{\mathcal{D}} \beta a_1 \ a_2}{\mathcal{R}_{\mathcal{D}} \beta P_1[a_1] \ P_2[a_2]}$
MUTUAL FAILURE $\frac{\text{fails } d_1 \quad \text{fails } d_2}{\mathcal{R}_{\mathcal{D}} \beta d_1 \ d_2}$	PASSIVE FAILURE $\frac{\text{fails } d_1 \vee \text{fails } d_2 \quad \text{passive } d_1 \quad \text{passive } d_2}{\mathcal{R}_{\mathcal{D}} \beta d_1 \ d_2}$	

Lemma 3 (Properties of $\mathcal{R}_{\mathcal{D}}$).

- *Reflexivity*: $\mathcal{R}_{\mathcal{D}} \beta d \ d[\beta]$ • *Symmetry*: $\mathcal{R}_{\mathcal{D}} \beta d_1 \ d_2 \implies \mathcal{R}_{\mathcal{D}} \beta^{-1} d_2 \ d_1$
- *Trans*: $d_2 \dashv\vdash d_3 \wedge \mathcal{R}_{\mathcal{D}} \beta d_1 \ d_2 \wedge \mathcal{R}_{\mathcal{D}} \beta' d_2 \ d_3 \implies \mathcal{R}_{\mathcal{D}} \beta' \cdot \beta d_1 \ d_3$
- *Extension*: $\beta' \supseteq \beta \wedge \mathcal{R}_{\mathcal{D}} \beta d_1 \ d_2 \implies \mathcal{R}_{\mathcal{D}} \beta' d_1 \ d_2$
- *Testable Eq.*: $\mathcal{R}_{\mathcal{D}} \beta d \ d' \wedge \text{test } d \wedge \text{test } d' \implies \text{valueof } d[\beta] = \text{valueof } d'$

Proof by induction. *Testable Equality* follows from the Test Failure property of Lemma 1, the struct case by the Top Conjunction property of Def. 2.

Definition 10 (Failure Preservation (on heaps): $h_1 \dashv\vdash h_2$).

$$\forall \ell \in \text{dom } \beta, (h_1 \ \ell) \dashv\vdash (h_2 \ (\beta \ \ell))$$

Lemma 4 (Properties of Failure Preservation (on heaps)).

- *R, T, • Failure Pres.*: $\text{fails } h_1 \ \ell \wedge h_1 \dashv\vdash h_2 \implies \text{fails } h_2 \ (\beta \ \ell)$

Definition 11 ($\mathcal{R}_{\mathcal{H}}$: β -indistinguishable heaps).

$$\mathcal{R}_{\mathcal{H}} \beta h_1 \ h_2 \iff \forall \ell \in \text{dom } \beta, \mathcal{R}_{\mathcal{D}} \beta (h_1 \ \ell) \ (h_2 \ (\beta \ \ell))$$

Lemma 5 (Properties of $\mathcal{R}_{\mathcal{H}}$). • *Reflexivity, Symmetry*

- *Trans*: $h_2 \dashv\vdash h_3 \wedge \mathcal{R}_{\mathcal{H}} \beta h_1 \ h_2 \wedge \mathcal{R}_{\mathcal{H}} \beta' h_2 \ h_3 \implies \mathcal{R}_{\mathcal{H}} \beta' \cdot \beta h_1 \ h_3$

Definition 12 (Simultaneous substitution: $\frac{C_1}{C_2} \beta \gg \frac{C'_1}{C'_2}$).

$$\frac{\text{ID} \quad \frac{C_1}{C_2} \beta \gg \frac{C_1}{C_2}}{\text{SUBST} \quad \frac{\mathcal{R}_D \beta \ d_1 \ d_2 \quad \frac{C_1[d_1/x] \ C'_1}{C_2[d_2/x] \ \beta \gg \frac{C'_1}{C'_2}}{C_1 \ \beta \gg \frac{C'_1}{C'_2}}}{C_1 \ \beta \gg \frac{C'_1}{C'_2}}}$$

Lemma 6 (Properties of Simultaneous Substitution).

- Reflexivity, (Vertical) Symmetry, Transitivity
- Extension: $\beta' \supseteq \beta \wedge \frac{C_1}{C_2} \beta \gg \frac{C'_1}{C'_2} \implies \frac{C_1}{C_2} \beta' \gg \frac{C'_1}{C'_2}$
- Decomposition: Suppose $\frac{\text{let } x=C_1 \text{ in } D_1 \ \beta \gg \frac{E_1}{E_2}}{\text{let } x=C_2 \text{ in } D_2 \ \beta \gg \frac{E_1}{E_2}}$. Then $\exists C'_1, D'_1, C'_2, D'_2$
 $E_1 = \text{let } x = C'_1 \text{ in } D'_1, E_2 = \text{let } x = C'_2 \text{ in } D'_2, \frac{C_1}{C_2} \beta \gg \frac{C'_1}{C'_2}, \frac{D_1}{D_2} \beta \gg \frac{D'_1}{D'_2}$.

Definition 13 (Failure-Preserving Evolution: $d_1 \rightsquigarrow d_2, h_1 \rightsquigarrow h_2$).

$$d_1 \rightsquigarrow d_2 \iff d_1 \neg d_2 \wedge \mathcal{R}_D \text{ id } d_1 \ d_2 \quad h_1 \rightsquigarrow h_2 \iff h_1 \neg \text{id} \neg h_2 \wedge \mathcal{R}_H \text{ id } h_1 \ h_2$$

Lemma 7 (Properties of Failure-Preserving Evolution).

- R, T • Transitivity with \mathcal{R}_H : $\mathcal{R}_H \beta \ h_1 \ h_2 \wedge h_2 \rightsquigarrow h_3 \implies \mathcal{R}_H \beta \ h_1 \ h_3$

Lemma 8 (FP Evolution Preserves Indistinguishable Heaps).

$$\mathcal{R}_H \beta \ h_1 \ h_2 \wedge h_1 \rightsquigarrow h'_1 \wedge h_2 \rightsquigarrow h'_2 \implies \mathcal{R}_H \beta \ h'_1 \ h'_2$$

Proof: $\mathcal{R}_H \beta \ h_1 \ h_2 \implies \mathcal{R}_H \beta^{-1} \ h_2 \ h_1 \implies \mathcal{R}_H \beta^{-1} \ h_2 \ h'_1 \implies \mathcal{R}_H \beta \ h'_1 \ h_2 \implies \mathcal{R}_H \beta \ h'_1 \ h'_2$

We now introduce write confinement: untrusted code's writes are confined (tainted), so the final heap is a failure-preserving evolution of the initial one.

Lemma 9 (Write Confinement For `taint_field`).

$$\begin{aligned} (!pc, h) \downarrow (p, h) \wedge \text{fails } p \wedge \text{passive } d \wedge \text{passive } d_2 \wedge \text{if } !pc \text{ fails,} \\ (\text{taint_field}(d, fl, d_2), h) \downarrow (d', h') \quad \text{taint_field run with passive } d, d_2, d' \\ \implies d \rightsquigarrow d' \wedge h = h' \quad \text{yields a f.p. evolution of } d \text{ and the heap is unchanged} \end{aligned}$$

Definition 14 ($\overset{n}{\downarrow}$). $\overset{n}{\downarrow}$ non-deterministically extends the \downarrow relation to open commands via any series of (well-formed) substitutions.

$$\frac{(C, h_1) \downarrow (v, h_2)}{(C, h_1) \overset{n}{\downarrow} (v, h_2)} \quad \frac{(C[w/x], h_1) \overset{n-1}{\downarrow} (v, h_2)}{(C, h_1) \overset{n}{\downarrow} (v, h_2)}$$

Lemma 10 (Write Confinement).

$$(!pc, h_1) \downarrow (p, h_1) \wedge \text{fails } p \wedge (\llbracket C \rrbracket, h_1) \overset{n}{\downarrow} (v, h_2) \quad \text{if } !pc \text{ fails, and a translated command} \\ \implies h_1 \rightsquigarrow h_2 \quad \text{runs with some substitutions, } h_2 \text{ is a f.p. evolution of } h_1$$

Proof by structural induction on $\llbracket C \rrbracket$. For the *application* case, the translation

gives the following series of heaps:

$$h_1 \xrightarrow[\text{set}]{\text{meth}} h_2 \xrightarrow[\text{set}]{pc} h_3 \xrightarrow{v_1 \ v'_2} h_4 \xrightarrow[\text{unset}]{pc} h_5 \xrightarrow[\text{unset}]{\text{meth}} h_6$$

In heap h_3 , $!pc$ fails, by def of `taint` and the On Frames and Right Extension properties of Lemma 1. For $v_1 \ v'_2$, it follows by inversion that $v_1 = R[\langle f, x, C \rangle] C[d/x][\langle m, x, C \rangle/m], h) \overset{n}{\downarrow} (d', h')$, so by Def. 14, $(C, h) \overset{n+2}{\downarrow} (d', h')$. Then $h_3 \rightsquigarrow$

h_4 by the IH. Going from h_2 to h_3 only changes pc , and going from h_4 to h_5 resets it, so $h_2 \rightsquigarrow h_5$, and similarly for $meth$, giving $h_1 \rightsquigarrow h_6$. The *conditional* case is similar. For the *let* case, weaken $(C_2[d_1/x], h_1) \downarrow (d_2, h_2)$ to $(C_2, h_1) \downarrow (d_2, h_2)$, and then apply the IH. The *assignment* case follows from Lemma 9.

Definition 15 (\mathcal{R}_C).

$$\begin{aligned} \mathcal{R}_C \beta \ C_1 \ C_2 &\iff C_1, C_2 \text{ are } \beta\text{-equal if for all } \beta\text{-equal heaps and sim. sub.} \\ &\forall h_1, h_2. \mathcal{R}_H \beta \ h_1 \ h_2 \wedge \forall C'_1, C'_2. C_1 \ C_2 \beta \ C'_1 \ C'_2 \wedge \text{ related } C'_1, C'_2 \text{ that run in those heaps} \\ &\quad (C'_1, h_1) \downarrow (d_1, h'_1) \wedge (C'_2, h_2) \downarrow (d_2, h'_2) \quad \text{that run in those heaps, the results are} \\ &\implies \exists \beta'. \beta' \supseteq \beta \wedge \mathcal{R}_D \beta' \ d_1 \ d_2 \wedge \mathcal{R}_H \beta' \ h'_1 \ h'_2 \ \beta'\text{-equal for some } \beta' \text{ extension of } \beta \end{aligned}$$

Lemma 11 (Reflexivity of \mathcal{R}_C). $\text{safe } C \implies \mathcal{R}_C \beta \llbracket C \rrbracket \llbracket C[\beta] \rrbracket$

Proof by induction on C . For commands that do not modify the heap, set $\beta' = \beta$.

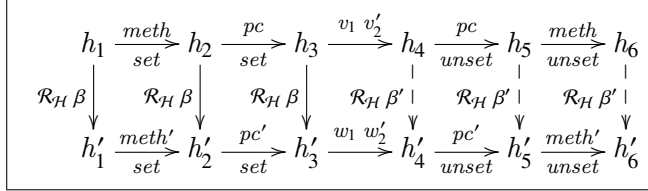
For the *ref* case, choose $\beta' = \beta[\ell_1 \mapsto \ell_2]$, where ℓ_1, ℓ_2 are the new locations chosen in the two heaps, and use the Extension properties of Lemmas 3 and 5.

For the *let* case (**let** $x = C_1$ **in** C_2), use the Decomposition prop of Lemma 6. Since sim. sub. is transitive (Lemma 6), weaken $C_2[d/x]$ to C_2 , apply the IH twice, then the Transitivity and Extension props. of Lemmas 3 and 5.

For the *application* case, consider $v_1 \ v_2$ and $w_1 \ w_2$. The translation gives

the following heaps:

The solid lines are trivial. The dashed lines proceed as follows. By inversion on

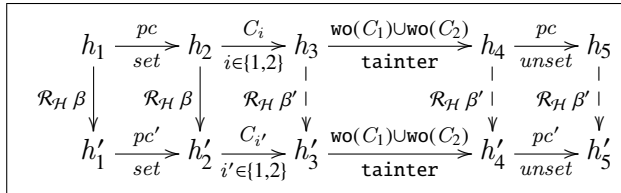


$\mathcal{R}_D \beta \ v_1 \ w_1$, either the closures in both are β -equal or they both fail (v_1 is not passive). If they both fail then $h_3 \rightsquigarrow h_4$ and $h'_3 \rightsquigarrow h'_4$ by Lemma 10. Then by Lemma 8 $\mathcal{R}_H \beta \ h'_4 \ h'_5$, and the tainted return values will be \mathcal{R}_D as they will both fail. Otherwise, by Def. 9 the closures must be β -equal, and so their bodies are of the form $C, C[\beta]$. By the transitivity of simultaneous substitution (Lemma 6) it follows that $C \llbracket \beta \rrbracket \llbracket C[d/x][\langle m.x, C' \rangle / m] \rrbracket$. The result then follows by IH.

Since the old values of pc and $meth$ in both heaps were \mathcal{R}_D under β , by the Extension property of Lemma 3 they are \mathcal{R}_D under β' , and h_5, h'_5 and h_6, h'_6 are obtained by respectively restoring the old values of pc and $meth$; it follows from Def. 11 that $\mathcal{R}_H \beta' \ h_5 \ h'_5$ and finally, $\mathcal{R}_H \beta' \ h_6 \ h'_6$.

For the *conditional* case, the translation of **if** v **then** C_1 **else** C_2 and **if** v' **then** C_1 **else** C_2

gives the following heaps: The solid lines are trivial, noting that by Def. 11 the pcs were \mathcal{R}_D beforehand and it is clear by Def. 9 that the pcs will still be \mathcal{R}_D . The dashed lines proceed as follows.



By inversion, for some $b, b', v=R[[b]], v'=R'[[b']]$, and $\mathcal{R}_D \beta R[[b]] R'[[b']]$. So either $\mathcal{R}_D \beta R R'$ and $b = b'$, or one (or both) of v, v' fails.

In the former case, the same C_i is executed in both runs, so by the IH, $\mathcal{R}_H \beta' h_3 h'_3$ and the return values are similarly $\mathcal{R}_D \beta'$. Since the pcs are \mathcal{R}_D , by Def. 9, the return denotations are also \mathcal{R}_D . Since the **tainter** commands are also the same, it follows by the IH that $\mathcal{R}_H \beta'' h_4 h'_4$ for some further extension β'' . The result follows by the Extension properties of Lemmas 3 and 5.

If both v, v' fail, then the pcs will also fail. By Lemma 10, $h_2 \rightsquigarrow h_4$ and $h'_2 \rightsquigarrow h'_4$, so by Lemma 8, $\mathcal{R}_H \beta h_4 h'_4$. As the pcs fail, the result follows from the def of **taint** and the On Frames and Right Extension props of Lemma 1.

If only one of v, v' fails, they must both be **passive**. WLOG, assume that v' fails. Then pc' in h'_2 fails, and so $\mathcal{R}_H \beta h_2 h'_2$ by Defs 9 and 11. By Lemma 10 $h'_2 \rightsquigarrow h'_3$, and so $\mathcal{R}_H \beta h_2 h'_3$ by the Transitivity property of Lemma 7. Let L be the set of (location, field list) pairs written to by C_i . By Def. 3.3, $\text{wo}(C_i) \supseteq L$ and so $\text{wo}(C_1) \cup \text{wo}(C_2) \supseteq L \cup \text{wo}(C_i)$ and $\text{wo}(C_1) \cup \text{wo}(C_2) \supseteq L \cup \text{wo}(C_1) \cup \text{wo}(C_2)$. Since $\text{wo}(C_1) \cup \text{wo}(C_2) \subseteq L \cup \text{wo}(C_1) \cup \text{wo}(C_2)$ it follows that $\text{wo}(C_1) \cup \text{wo}(C_2) = L \cup \text{wo}(C_1) \cup \text{wo}(C_2)$. The lhs of the equality represents every (location, field list) pair modified in the first execution, and the rhs represents those tainted by the second execution. By Def. 9, all such locations will still be \mathcal{R}_D of each other, so $\mathcal{R}_H \beta h_4 h'_4$. Resetting the pcs preserves \mathcal{R}_H , so $\mathcal{R}_H \beta h_5 h'_5$ holds. The return denotations and the pcs must be **passive**, so the tainted return denotations are **passive** and by Def. 9 the tainted return denotations are \mathcal{R}_D .

Proof of Theorem 1: The initial heaps are empty, and by Def. 11 are $\mathcal{R}_H \emptyset$. Every command in the let bindings will be run in a context where the pc fails, by inspection of the translation of **assert in** and the Completely Invalid property of Lemma 1. Let h_3, h_4 (and h'_3, h'_4 in the other run) be the heaps before and after such a command. Then by Lemma 10, $h_3 \rightsquigarrow h_4$ and $h'_3 \rightsquigarrow h'_4$. Given $\mathcal{R}_H \emptyset h_3 h_4$, $\mathcal{R}_H \emptyset h'_3 h'_4$ follows by Lemma 8. Since the initial heaps are $\mathcal{R}_H \emptyset$, the heaps will still be $\mathcal{R}_H \emptyset$ after all the let bound commands have run. Since U_i, U'_i are completely invalid, **fails** U_i and **fails** U'_i and so **fails** d_i and **fails** d'_i by the Completely Invalid, On Frames, and Right Extension properties of Lemma 1. By Def. 9, $\mathcal{R}_D \emptyset d_i d'_i$, and so by Def. 12, $\frac{C}{C} \emptyset \gg \frac{C[d_i/\bar{x}_i]}{C[d'_i/\bar{x}_i]}$.

By hypothesis, **safe** $C = C[\emptyset]$. By Lemma 11, $\mathcal{R}_C \emptyset C C$. Denoting the return values of the entire programs d, d' respectively, by Def. 15 $\exists \beta. \mathcal{R}_D \beta d d'$. Since **test** d and **test** d' both succeed, by the Test. Eq. property of Lemma 3, **valueof** $d[\beta] = \text{valueof } d'$, so **valueof** the resulting denotations are β -equal. \square