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

Avraham Shinnar
Harvard University
shinnar@eecs.harvard.edu

Marco Pistoia
IBM T. J. Watson Research Center
pistoia@us.ibm.com

Anindya Banerjee
IMDEA Software
anindya.banerjee@imdea.org

Abstract
This paper presents $\lambda_I$, a language for dynamic tracking of information flow across multiple, interdependent dimensions of information. Typical dimensions of interest are integrity and confidentiality. $\lambda_I$ supports arbitrary domain-specific policies that can be developed independently. $\lambda_I$ 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 introduces IMPOLITE, a novel class of information-flow policies for $\lambda_I$. 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 $\lambda_I$’s unified framework.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features; D.4.6 [Operating Systems]: Security and Protection—Information flow controls, Verification

General Terms Languages, Security, Verification

Keywords Information flow control, declassification, security type system

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 [20]. Research systems, such as Jif [16], Flow Caml [6], and Information-Based Access Control (IBAC) [20], 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, and from the point of view of confidentiality, 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 defines $\lambda_I$, 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 introduces 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, $\lambda_I$ 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.

Relative-security relations are also useful to overcome some limitations of existing access-control models. For example, in Java and the CLR, permissions are assigned to classes by class loaders [21]. The provider of a class loader is implicitly granted the authority to assign any permission to any class loaded by that class loader. Ideally, the permissions assigned to a class should be trusted as much as the class-loader provider is trusted, but since Java and the CLR security models do not support relative-security relations, this restriction is not possible. For this reason, the literature [9, 22] has emphasized that the permission $p$ to instantiate a class loader is implicitly equivalent to AllPermission [21, Section 8.2.5]. To address this issue, unlike other programming languages, $\lambda_I$ allows addressing information-flow metadata as first-class information. For example, $\lambda_I$ allows keeping track of the fact that a datum $c$ may be labeled with an integrity level $R$, and $R$ itself may be labeled with integrity level $S$. In $\lambda_I$, we write this as $S[R][\langle c \rangle]$, using frames [24, 7, 10], and we say that $R$ frames $c$ and $S$ frames $R$.

Common systems are also incapable of using relatively-trusted integrity-enforcement mechanisms. In Java and the CLR, an installed security manager can enforce any policy it desires [21, Section 8.2.5] [22, Section 7.5.1]. Thus, a malicious implementation of a security manager can make any permission check succeed. This is the same as granting AllPermission to arbitrary code. Conversely, in a system supporting relative-security relations, any security decision made by a security manager would be constrained by the security manager’s integrity level, and would be trusted as much as the security manager itself is trusted.

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 $a$ of type $\forall \lambda$, and performs a callback, $a.f$. An intruder can choose to inject one of two perhaps identical implementations of $a.f$, with just 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 would have influenced the control flow of the program. $\lambda_I$ handles this situation by framing the frame of $a.f$ with the frame $A$ of the attacker, as in $A[R_1][\langle a.f \rangle]$ and $A[R_2][\langle a.f \rangle]$, 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, e.g., $A[R_1][\langle S_1 \rangle][\langle a.f \rangle]$ and $A[R_2][\langle S_2 \rangle][\langle a.f \rangle]$. This demonstrates the need for unbounded 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. In Section 2, we will show the importance of a language that can account for interdependent dimensions of integrity and confidentiality.

1.1 Overview

Section 3 presents $\lambda_I$, an expressive language for dynamic information-flow tracking in multiple, interdependent dimensions. Information-flow tracking is built into $\lambda_I$, which allows programs to access and manipulate information-flow metadata. $\lambda_I$ dynamically maintains security metadata throughout the execution of a program for subsequent policy decisions.

A storage channel arises whenever an attacker can influence the information-flow metadata of a value used in a security test, and not necessarily the value itself, thereby influencing the control flow of the program, as in the class loader, security manager, and callback examples discussed above. The ability of $\lambda_I$ to maintain frames on frames allows $\lambda_I$ to fully account for frames used as storage channels, in contrast to previous work [7, 20].

$\lambda_I$ tracks information-flow dynamically, potentially allowing it to accept more programs than static systems—such as type-based information-flow systems—at the cost of greater overhead. The overhead arises because of the need to handle implicit flows. For example, the fact that an action has been prevented might itself constitute a leak. The correct handling of these flows requires the use of a write oracle that essentially calculates what locations a code may write—similar to the modifies specification required by tools such as JML [12]. Details are available in the research report [25]. $\lambda_I$ separates a unified information-flow tracking mechanism from domain-specific policies via lazy policy enforcement. $\lambda_I$ 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 $v_2'$.

$\lambda_I$ unifies the way information flow is tracked across domains, neither interpreting nor constraining the data or policies. Only at the point in which a security test on a value necessary, is the information-flow metadata attached to the value extracted and interpreted by the test function, which is encoded in $\lambda_I$. Furthermore, $\lambda_I$ treats integrity and confidentiality uniformly and does not constrain the allowable policies. In contrast, non-lazy systems generally constrain policies, requiring the labels to form a lattice [4].

As discussed in Section 3.2, $\lambda_I$ supports endorsement, declassification, and other essential information-flow primitives. $\lambda_I$ 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. An explicit encoding is included as Appendix A.

Section 4 introduces Information Management Policies in a Limited Trust Environment (IMPOLITE), a class of security policies, enforceable on $\lambda_I$, that allow relative-security relations on multiple interdependent dimensions of information. Existing systems, such as IBAC [20], 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. A non-interference theorem and proof for IMPOLITE appear in the research report [25].

1.2 The Attacker Model

We assume a trusted-memory model; all memory accesses are mediated by the runtime. This allows us to support an active-attacker model [18]. Outside observers can inject code into a program and monitor its public interactions. Modulo timing- and termination-attacks, an active attacker cannot compromise a system with an IMPOLITE policy[25].

2. Motivating Example

Figure 1 models a medical record scenario. In a medical record each field’s value may have its own integrity and confidentiality requirements. These in turn may have their own integrity and confidentiality requirements, necessitating a system that can model the complex interactions of multiple, interdependent dimensions of information.

In the scenario of Figure 1, Mr. Doe was seen by Dr. Smith. 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 given by Mr. Doe, yielding 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 integrity label of Medical Notes with a confidentiality label, Payroll.

**Integrity on Integrity.** The integrity label on Dr. Smith 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 enforced immediately prior to a security-sensitive event. The programmer can choose when to enforce their desired policies. The language simply tracks influences.

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

3.1 Core Frame Language: $\lambda_F$

Figure 2 presents the syntax of $\lambda_F$, an A-normalized language with references, structures, conditional, and recursive first-class functions. $\lambda_F$ also includes frames, a mechanism for associating information-flow metadata with data. $\lambda_F$ 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(b) so that only atomic denotations are framed. null is used as a terminator and is absorbed by framing.

Canonicalization emphasizes the underlying data, ascribing a clear, useful meaning to constructs such as $\mathcal{R}([S])$[3] + $T$[4]. It succinctly describes the interplay between frames and structures, allowing $\lambda_I$ to track multiple dimensions of information in an interdependent fashion. Structures can also be used to encode structured information. A plus function, given arguments $\mathcal{R}([S])$[3] and $T$[4] as above could return $\{\text{part1} = \mathcal{R}([S]), \text{part2} = T\}[7]$.

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

Figure 3 describes the semantics of $\lambda_F$. $d[d_1/x]$ is standard capture avoiding substitution of $d_1$ for $x$ in $d$. $\lambda_F$ provides recursive closures as a form of fix frame with,
frameof, and valueof are the intro and elim forms for framed constructs. passive, as expressed in Figure 3(a), is a predicate on denotations indicating that they do not contain closures. 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 discussed in Section 3.2, which introduces \( \lambda_I \).

3.2 Information-Flow Language: \( \lambda_I \)

\( \lambda_I \) is defined by translation to \( \lambda_F \), tracking control- and data-flow and framing every value with its dependencies. Figure 4 presents excerpts of this translation; the full translation is in the tech report[25].

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 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 5, 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. \( \lambda_I \)’s application rule frames \( f \)’s argument, \( dir \), with the meth frame of \( f \)’s caller and \( f \)’s return value with \( f \)’s frame.
\(\lambda_I\) provides access to the underlying \texttt{frame with}, \texttt{frameof}, and \texttt{valueof} commands in \(\Lambda_F\), 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. \(\lambda_I\) also adds in some new commands. \texttt{assert in} allows the programmer to explicitly ignore control dependencies. This is akin to \texttt{doPrivileged} in Java and Assert in the CLR. If a library wants to write to a log file, it can use \texttt{assert in} to do so, even if the client does not have the required permissions. \texttt{getpc} and \texttt{getmeth} help the programmer selectively ignore some control dependencies. Appendix A has some examples.

### 3.3 Restrictions

To ensure correct influence tracking, \(\lambda_I\) restricts conditionals, closures, and the heap. These restrictions are all related to implicit flows. An attacker can influence the behavior of a program by preventing it from executing some code that would otherwise have been executed. More sophisticated static analysis could remove these restrictions by conservatively approximating the code that did not execute. For simplicity, we choose to simply prohibit some of the more difficult to analyze situations. To handle if statements, \(\lambda_I\) assumes a write oracle that approximates the locations that would have been written in the branch not taken. This is similar to that employed by Le Guernic et al. [14, 13]. More information concerning the motivation for these restrictions as well as approaches for removing them can be found in [25].

### 4. Security Policy Interpretation

In this section we discuss the IMPOLITE class of policies and the interpretation of such policies.

Our overall goal is to check diagrams such as Fig. 1 with regards to policies such as: Payroll is allowed to view Dr. Smith was consulted by Mr. Doe, but Joe the Plumber is not allowed to view this information. Here is another policy: Payroll cannot view Dr. Smith’s notes.

An IMPOLITE policy has two components. The first component is a relative trust relationship, termed validfor, between (field name, atomic denotation) pairs. For example, in the scenario of Section 2, the trust relation can ask: Do Payroll employees believe that Dr. Smith is the integrity label on Medical Notes? In our formulation this is represented as (C, Payroll)=(I, Dr. Smith), read: Is (C, Payroll) valid for (I, Dr. Smith)? (Were one to define a relation that asserted (I, Dr. Smith) for all pairs \((f, a)\), where \(f\) is a label and \(a\) is an atomic notation, one obtains the IBAC [20] policy.)

The second component of a policy specifies how a policy is interpreted on general structures (see Section 3.2). For 1, Doctor, Mr. Doe and Ms. Jones can be represented as a structure \{\textit{part}1 = Doctor, \textit{part}2 = Mr. Doe, \textit{part}3 = Ms. Jones\}; that they are all given access to Medical Notes is represented by the fact that the interpretation of this structure in dimension C is true if one of Doctor, Mr. Doe or Ms. Jones attempts access to medical notes. More complex policies such as always requiring the presence of Mr. Doe along with Ms. Jones or a Doctor can also be interpreted.

Given a policy specified as above we show [25] that a \(\lambda_I\) program that does not use endorsement or declassification satisfies non-interference. This means the following: Consider two runs of the program from two initial environments set up by completely untrusted (not even partially trusted) attackers. Run the program from these two environments. If the respective denotations of the program satisfy the security policy then the two denotations are the same. Note that the formulation does not preserve violation of the policy in intermediate states of computation as long as the violations have no influence on the final denotations.

### 5. Related Work

Since Denning and Denning [5], there has been a large volume of work on static checking of information flow policies [23]. Goguen and Meseguer [8] introduce non-interference based on earlier work by Cohen [3]. Volpano,
et al. [28] 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. [15]. Based on the premise that many software attacks subvert the execution of machine code, Abadi, et al. perform a comprehensive study of control-flow integrity [1].

Myers’ Jif [16] and Pottier and Simonet’s Flow Caml [6] use type-based static analysis to track information flow. Neither Jif nor Flow Caml address the interactions amongst different dimensions of information. Jif is based on the Decentralized Label Model [17]. 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. Jif thereby requires the programmer to specify the security policy. In contrast, \( \lambda_I \) can encode declarative policies, such as those of Java and the CLR. \( \lambda_I \)'s memory model assumes a core trusted memory that 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. Furthermore, \( \lambda_I \) supports a very flexible policy-enforcement mechanism, with arbitrary values as labels, and arbitrary information-flow tests.

Zheng and Myers [29] describe a language that, like \( \lambda_I \), allows first-class labels. Their language assumes that the labels form a lattice and use the associated join operation to merge labels. In contrast, \( \lambda_I \) does not require that the labels form a lattice, and maintains the structure of the labels. This allows \( \lambda_I \) to account for interdependencies between different dimensions of information. This also allows \( \lambda_I \) to support policies such as the relative-security relations of IMPOLITE.

Pistoia, et al. [20] describe IBAC, a unified access-control and information-flow system that uses permission sets as labels for information flow. The IBAC language supports a subset \( \lambda_I \)'s features. IBAC’s non-interference can be viewed as an instantiation of the IMPOLITE non-interference result. IBAC does not support any relative-security relations. The use of the intersection operator by IBAC can be viewed as a policy-driven optimization.

With robust declassification, Myers, et al. enforce that only high-integrity data be declassified, and declassification be performed only in high-integrity contexts [18]. Qualified robustness provides an attacker a limited ability to affect what information may be released by programs [19]. An endorse primitive is used to upgrade the integrity of data. The RX language allows integrity and confidentiality metapolicy labels on roles [27, 11]. Deeper interactions between integrity and confidentiality are not investigated.

Swamy et al. [26] design the dependently-typed language, Fable, wherein a variety of security policies can be expressed. Such policies include access control, information flow and provenance. Data is protected by way of security labels based on dependent types. These labels, like our frames-on-frames, capture security policy. Policy enforcement happens in a separate part of the code by way of an interpretation of labels provided by the programmer, and can be checked statically. In contrast, \( \lambda_I \) tracks information flow dynamically and does not handle access control and provenance presently. Although Fable allows programs in which multiple policies may be at play simultaneously, the notion of composition of the policies forbids any interaction between policies. In contrast, \( \lambda_I \) allows interaction between confidentiality and integrity policies.

6. Discussion

This paper presented \( \lambda_I \), a language for dynamic tracking of information flow in multiple, interdependent dimensions. A promising research direction is to apply \( \lambda_I \) to other dimensions of information, such as non-repudiation, provenance, and concurrency.

As presented in this paper, \( \lambda_I \) has a number of restrictions to ensure that influences are correctly tracked. We would like to remove these restrictions by designing a static analysis or type system to conservatively approximate or constrain implicit flows.

We are also interested in exploring policy-driven optimizations. \( \lambda_I \) 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 \( \lambda_I \)'s tracking mechanism.

Acknowledgments

The authors would like to thank Greg Morrisett, John Field and David Naumann for their helpful suggestions.

References

A. Encoding Security Primitives in $\lambda_I$

Many common security primitives can be encoded in $\lambda_I$. We present here encodings for three primitives found in Java and the CLR: `doPrivileged`, `Assert`, and `doAs`. They all take a closure $m$ encoded in Java and the CLR as an object with a single `run` method. For readability, the examples are not in A-normal form; A-normalization is straightforward.

Java's `doPrivileged` prevents stack inspection from traversing the rest of the stack; the caller of `doPrivileged` is checked, but the code above it is not. In $\lambda_I$ this is encoded by adjusting the control dependencies using `assert`.  

$$\text{doPrivileged}(m) \equiv \text{assert \ null \ in \ (m \ \text{unit})}$$

The CLR's `Assert` stops stack inspection from looking at the rest of the stack above the caller of `Assert` for a given permission. If frames are just simple permissions, then, assuming an equality test, this can be encoded by selectively removing problematic frames.

$$\text{Assert}(p, m) \equiv \text{let \ filter =}
\begin{align*}
&\text{fix \ f l \Rightarrow} \\
&\text{let \ v = valueof \ l \ in} \\
&\text{let \ r = frameof \ l \in} \\
&\text{if \ v = \null \ then \ null} \\
&\text{else if \ v = p \ then \ f r} \\
&\text{else frame \ v \ with \ f r \in} \\
&\text{assert \ (filter \ getpc) \ in \ (m \ \text{unit})}
\end{align*}$$

Java's `doAs` takes a subject $s$ and a closure $m$, and runs $m$ after adding the permissions of $s$ to the permissions already owned by $m$. Assuming that `perms` returns $s$’s permissions, `doAs` in $\lambda_I$ alters $m$’s frames appropriately.

$$\text{doAs}(s, m) \equiv (\text{frame \ m \ with \ perms(s)}) \ \text{unit}$$