A Survey of Language-Based Information Flow Control

Lucas Du Zhuoli Huang Qianqian Tian

June 6, 2025

1 Introduction

Preserving the security and privacy of data processed by computer systems is a persistent issue and one that is becoming increasingly important. There are various approaches to providing privacy and security guarantees in software systems: one such approach is **information flow control** (abbreviated IFC), which tracks the propagation of information through a program to verify that all information adheres to certain security or privacy requirements.

In particular, a programming language-based approach—often using security type systems—has been seen as an effective method to apply fine-grained information flow control in practice. This project aims to survey the existing literature on information flow control and, more specifically, the language-based approach to information flow control.

In particular, we will:

1. give general theoretical background on information flow security,
2. outline core tensions and research directions in language-based IFC,
3. and finally, explore and analyze different methods of integrating IFC into existing languages, guiding future work on making language-based IFC practical.

2 Foundational Background

A major issue in computing systems, particularly as they are increasingly used to store and process sensitive data, is ensuring information *confidentiality*. Confidentiality has roots in military and governmental security applications and prioritizes *information secrecy*. As such, it relates to both notions of *security* and *privacy* (and these terms may be used interchangeably in this survey). Information should only be disclosed to entities that are allowed to access it and should never flow to unauthorized entities [3].

Information flow control represents a technique that enables *fine-grained* enforcement of confidentiality requirements by tracking the flow of information throughout a system[1]. In fact, confidentiality policies are often also called *information flow* policies, since confidentiality is essentially about controlling the way information propagates through a system [3].

[1] Information flow control techniques can also be applied to the modelling and enforcement of *integrity*—often seen as the dual property of confidentiality—which prioritizes *information trustworthiness*. However, the bulk of work on information flow control to-date has focused on confidentiality.

Central to information flow security and control is **non-interference**: a program satisfies the non-interference property (or, in some formulations: a program is *non-interference secure*) if the values of its public outputs do not depend on values of its secret inputs [9, 6].

This is an informal description, but serves to capture the intuition behind the property—after all, if a public output depends on some secret input, then there is clearly an information leak violating confidentiality. Non-interference is often seen as the gold-standard in information flow security and comes in a number of different flavors, which will be outlined later in this section.

2.1 Access Control vs. Information Flow Control

Information flow control may seem like a natural and precise way to ensure information confidentiality, but historically, *access control* has been the dominant technique in practice. What is the relationship between these two paradigms? And what are some practical difficulties in using information flow control over access control?

Access control is, in some sense, the traditional way to handle the problem of providing fine-grained information confidentiality. Indeed, many (if not most) contemporary systems rely on access control policies and mechanisms to specify and enforce confidentiality guarantees.

Access control limits *which* entities may access *what* pieces of information, but only at the time of access. This is simple, but leads to a major weakness: access control is *all-or-nothing* in the sense that once a subject has access to an object, it can do anything it wants with the information represented by that object. As a result, there is no real control over the way that information can *flow* through a system once some subject gains access to it—the subject is free to improperly leak information in whatever way they wish.

Access control can thus be seen as a coarser and less complete version of information flow control. This coarseness also leads to some theoretical limitations: in 1976, Harrison, Ruzzo, and Ullman proved that, when using the access control matrix as a model of access control, it is *undecidable* whether a right to an object will leak to an unauthorized process [8]. In other words, in the general case, ensuring fine-grained confidentiality using access control is not computationally tractable.

Information flow control (IFC) can be viewed as a response to this weakness. IFC seeks to provide a finer-grained model of how information actually *propagates* through a system and to provide formal guarantees about information security and privacy—accounting for the flow of information after initial access—in a decidable way.

This, of course, comes with a host of difficulties (as is often the case whenever finer-grained control is desired). Information flow control requires that we track how security levels associated with subjects and objects, often known as *security labels*, are propagated through a system. It also requires that we ensure all accesses at *each* point in *any* sequence of commands do not violate the current security labels at that point.

This sort of fine-grained tracking and checking of information throughout a program execution requires much deeper and more invasive integration with (or modifications of) existing systems. In some ways, it may require a ground-up re-implementation of the system, which is not feasible in many cases.

Broadly, there are two sorts of foundational systems through which we *can* implement the fine-grained tracking and checking required by information flow control: **programming languages** and **operating systems**. Work on information flow control can be roughly divided into these two categories, each with its own advantages and areas of focus.

2.2 Language-Based vs. OS-Based IFC

Language-based IFC tracks information propagation within a program written in some IFC-aware programming language. This allows fine-grained tracking of information through program structures, but trusts that the underlying operating system abstraction is secure, i.e. information flow security is preserved when accessing system resources like files and sockets.

OS-based IFC monitors information flow through underlying system resources using operating systems abstractions. This provides security and privacy guarantees about information flow through those low-level system resources, but has no visibility into (or very limited and often inefficient visiblity into) the flow of information through structures within programs themselves.

The concerns of these two categories of IFC work are, in many ways, orthogonal and mutually beneficial. In an ideal world, we would have both programming languages *and* operating systems that allow for such fine-grained information flow control and that cooperate to provide efficient end-to-end information flow security guarantees.

There are some important practical distinctions between these two categories that are worth pointing out:

1. OS-based IFC is more *general* in the sense that *any program* running on an OS equipped with IFC enjoys the information flow security benefits provided by the OS.

2. OS-based IFC can also be viewed as more *heavy-handed*, since it requires modification to the operating system [12] or an entirely new operating system altogether [20]. In many production settings, choice of operating system is under the control of some centralized (and often conservative) administrator and affects *all* other users of the machines on which it is installed.

3. Language-based IFC is more *specific* in the sense that a program must be written the IFC-equipped language in order for it enjoy the desired information flow properties. However, it offers fine-grained and often very efficient analysis of information flow within programs themselves, since the analysis is program-specific and can often be done statically. Comparatively, OS-based IFC often must be *dynamic*, since the OS usually has no information about program behavior prior to program execution. This comes with a performance cost.

4. Language-based IFC can also be viewed as more *flexible*, particularly in production settings, since the programmer often has more control over what programming language they use (as long as the language runs on commodity operating systems and hardware).

This survey focuses on **language-based IFC**, although the boundary between the programming languages and operating systems work on IFC is often blurry and closely intertwined [12].

2.3 The Lattice Model of Information Flow Security

The **lattice model** of information flow security, as introduced[2] in Dorothy Denning's seminal 1976 paper [4], forms the foundation of nearly all later work on information flow security. That said, it can also be viewed as the genesis of *language-based* information flow security more specifically, since Denning's line of work (which was expanded upon in a 1977 paper by Dorothy and Peter Denning [5]) was focused on providing efficient *compile-time* guarantees of information flow security.

[2]Note that the lattice model of security itself is neither unique to information flow security nor first introduced in the context of information flow security. In fact, lattices are also historically used to model access control models like Bell-LaPadula [17].

The key idea in Denning's 1976 paper is that it is possible (and justified) to model information flow security as a *universally bounded lattice* of security classes. Here, a lattice (sometimes more explicitly called a *complete* lattice) is the algebraic structure that comprises a **partially ordered set** in which each pair of elements in the set has a least upper bound *and* a greatest lower bound. A *universally bounded* lattice is a lattice that also has a *top* element (an element greater than all other elements in the set) and a *bottom* element (an element less than all other elements in the set).

A partially ordered set is a set of elements and an associated relation function between those elements that is: reflexive, antisymmetric, and transitive. Specifically, this means the relation is:

1. reflexive: for any element a, $a \sqsubseteq a$ (where $\sqsubseteq$ is the ordering relation)
2. antisymmetric: for any pair of elements a and b, if $a \sqsubseteq b$ and $b \sqsubseteq a$, then $a = b$
3. transitive: if $a \sqsubseteq b$ and $b \sqsubseteq c$, then $a \sqsubseteq c$

In the lattice model of information flow security, elements of the lattice are *security classes* and relations in the lattice are *flow relations* from security class a to security class b, where information in class a is permitted to flow into class b. Security classes are bound to entities in the system, either through *static binding* (where the security class of an entity is constant) or *dynamic binding* (where the security class of an entity is permitted to vary with its contents).

Denning observes that certain assumptions are necessary about relations between security classes if the universally bounded lattice model is to hold, but that these assumptions are justified and follow *directly* from the natural semantics of information flow. Specifically, the following assumptions—which define a universally bounded lattice—are (a) required for the consistency of an information flow security model (all flows implied by a permissible flow should also be allowed by the flow relation) and (b) lose no generality (all information flow traces that can be captured by our security classes and flows-to relation can be modelled with these assumptions):

1. $\langle SC, \rightarrow \rangle$ is a partially ordered set (where SC is the set of security classes and $\rightarrow$ is the flow relation)
2. SC is finite
3. SC has a lower bound L such that $L \rightarrow a$ for all $a \in SC$
4. $\oplus$ is a least upper bound operator on SC
5. The above assumptions additionally imply the existence of $\otimes$, a greatest lower bound operator on SC, and unique upper bound H

The interested reader can find the full argument justifying these assumptions in Section 2.2 of [4].

A core advantage of the lattice model is that, by modelling information flow in this way—which is *sound*, but not *precise*—we sidestep Harrison, Ruzzo, and Ullman's earlier undecidability result (at the cost, of course, of a loss of precision) [5]. In this context "sound, but not precise" means: some information flow secure programs will be rejected by this model (not precise), but all programs certified by this model will be information flow secure (sound).

2.3.1 An Early Security Type System

Denning and Denning use this lattice model to construct a language in which information flow security can be certified *at compile time* [5] using, essentially, a bare-bones effect-tracking type system. In many ways, all future work on security type systems follows from this idea[3].

The core novelty of Denning and Denning's typing mechanism is the way it exploits the structure of lattices to provide an *efficient* way to check fine-grained information flow. At a high-level, the lattice structure enables the following optimizations:

- The *transitivity* property of flow relation allows us to *compose* information flow certification checks, since small, certified, local flows can be automatically combined into larger, certified, global flows. In Denning and Denning's toy language, the semantics for flow relations need only be defined for a small number of core syntactic constructs. The semantics for flow relations for any program in their toy language can be constructed from these core semantics.

- The existence of a *least upper* and *greatest lower bound* for all subsets of the set of security classes *SC reduces the amount of information* needed to track information flow. Specifically, the compiler is able to *group* the security classes of sources and flows together using these bounds and check flows between the bounds, instead of between all the individual security classes.

2.3.2 Strengths and Limitations

Two strengths of the Denning-style type system approach are that (a) it can be done statically at compile-time, meaning that there is simultaneously no runtime overhead and that certified programs are *correct-by-construction* and (b) it is able to capture and check both *explicit* **and** *implicit* flows.

Explicit flows occur when information is passed **explicitly** from the right-hand side to the left-hand side of an assignment statement, such as `public := secret` [14]. In this statement, information represented by `secret` is explicitly flowing into the entity represented by `public`.

Implicit flows occur when information is passed **implicitly** via a *control structure* (like an if-then-else statement) [14]. For example, the statement `if secret then public := somevalue` contains an implicit flow: the value of `public` depends on the value of `secret`, so information is flowing "implicitly" from `secret` to `public`.

However, there are some major limitations of Denning and Denning's language and type system, which work afterwards sought to address.

- The toy language introduced in Denning and Denning's paper is extremely limited. Adding Denning-style IFC type-checking to more expressive and realistic languages is a long-standing research question (and one of the major contributions of the Jif/JFlow project) [10, 15].

- The type-checking system in Denning and Denning's paper is *not ergonomic* (by certain standards) in the sense that it requires many manual annotations.

- The type-checking system is *not realistic* in the sense that it fails to model important real-world patterns like information declassification (and the dual operation: endorsement).

- The purely static type-checking system is relatively *restrictive* and has difficulty modelling dynamic binding of security labels to entities or dynamically changing security policies.

[3] Indeed, the term "Denning-style type systems" is often used to describe this line of work.

Indeed, these tensions between expressiveness, flexibility, efficiency, and usability have spurred on a rich line of research.

3 A Core Tension: Dynamic vs. Static

One such core tension in language-based IFC is the tradeoff between dynamic and static enforcement. Traditional static analysis, e.g., Denning-style type systems, tracks variable security labels and checks the code for security and privacy violations before running it. This does not result in any runtime overhead. However, being conservative, there is a risk of rejecting secure programs. Dynamic analysis, such as a monitor at runtime that checks the execution in real time, is more flexible but runs the risk of not being able to block insecure programs.

3.1 Theoretical Boundaries

An important distinction between types of information flow control enforcement is the difference between *flow-insensitive* and *flow-sensitive* enforcement. The distinction also boils down dynamicism vs. static checking:

- Flow-insensitive means fixed labels for variables throughout execution.
- Flow-sensitive variable labels can dynamically change over time.

Flow-sensitive static information flow analysis is a generalization of flow-insensitive static analysis, and accepts more information flow secure programs. It has also been shown that, in the flow insensitive case, purely dynamic-based information flow enforcement can not only enforce the same degree of information flow security as Denning-style static analysis—namely termination-insensitive non-interference—but is also more permissive (in that it accepts more secure programs) [16].

That said, Russo and Sabelfeld give an interesting *impossibility result* in the flow-sensitive case that sheds light on an important theoretical boundary: it is not possible to construct a purely dynamic monitor that is *sound* **and also** *as permissive* as a static type-system [14].

Russo and Sabelfeld go on to show that it is possible recover soundness in the flow-sensitive case by taking a hybrid approach, combining both a static type system and a dynamic monitor. They propose a general framework for such a hybrid mechanism, built on a simple imperative language, that is parametric over the enforcement actions of the monitor (i.e. block output, output default values, or suppress event) [14, 9].

3.2 Dynamic Labels, Statically

It is also interesting to consider how much dynamicism, i.e. with dynamic labels and policies, can be moved to static compile-time checks. Indeed, Zheng and Myers consider the use of $\lambda DSec$, a typed lambda algorithm language that combines dynamic labels with static constraints using *dependent types* (types that depend on program values) [21]. Generally, dependent types can move certain checks that can conventionally only be done at runtime into a static, pre-execution typing check.

That said, in order to completely ensure secure control of information flow in situations where access rights can be dynamically changed and determined, it is necessary to also use dynamic labels that can be manipulated and checked at runtime, with some sort of dynamic monitor [21].

4 Practical Integration into Existing Languages

A major challenge for existing IFC techniques is integration with existing languages. In many ways, this is another dimension of the core tensions mentioned earlier: we gain a degree of usability and practicality by integrating with existing languages, at the possible cost of expressiveness, flexibility, and efficiency.

Ideally, we would like to (a) use an existing (and hopefully "mainstream") language (b) with minimal modifications (c) to enforce deep and precise information flow security properties of programs (d) with minimal runtime overhead and (e) minimal cognitive overhead for the programmer. How can we satisfy all these ideals?

Here, we survey some existing approaches to easing these tensions, all of which integrate with existing programming languages. We outline each approach, find commonalities and differences between the techniques they use, consider the portability of some of those techniques, and then generalize these ideas to broader guidance for both future IFC integrations with existing programming languages as well as the design of programming languages themselves.

4.1 JFlow

JFlow is a language extension based on Java. It adds information flow security labels and related keywords, but is still fully compatible with Java syntax and finally generates ordinary JVM bytecode. Unlike Jeeves and Carapace, the other two examples we analyze, JFlow is not available as a library. JFlow is primarily a static, type-system–based approach to IFC that compiles decentralized labels to ensure data flows only to authorized principals [10].

4.1.1 An overview of JFlow

JFlow is an extension of the Java language that uses static checking of flow annotations as an extended form of type checking. Programs written in JFlow can be statically checked by the JFlow compiler to prevent information leakage through storage channels. Static information flow analysis has been proposed for a long time, but it has not been widely adopted by mainstream security practices. The key reason is that previous models are too limited or too strict to be implemented. The goal of JFlow is to introduce stronger expressiveness while maintaining the advantages of static checking, allowing developers to write practical programs in a natural way [10].

4.1.2 Lessons on language integration

Information-flow control can be integrated into a mainstream object-oriented (OO) language largely through static typing, without sacrificing usability. The central design choice in JFlow is to keep the security logic orthogonal to the core program logic, while enforcing it early—at compile time—rather than at run time [10]. To make this practical, JFlow adds a small amount of syntax to Java, pushes most annotation work onto the compiler, and resorts to targeted runtime checks or declassification only when static reasoning would be overly restrictive.

1. In JFlow's static information flow control, each variable is bound to a static security label. Developers explicitly annotate the owner of the data and its authorized readers using the syntax {`owner:[readers]`} in Java source code. The type-label checker at compile time verifies confidentiality levels for each assignment: an assignment like `z = x` is allowed only if the label of `x` is not weaker than that of `z`. Once approved, the target variable `z` receives a new static label binding, and future operations cannot bypass this constraint, thereby blocking potential data leakage at the semantic level.

2. It is impractical to require programmers to manually annotate labels for every variable. To address this, JFlow reduces the annotation burden through default label inference and implicit label polymorphism. If a variable, parameter, or method declaration omits the {`owner:[readers]`} label, the compiler infers it based on context. Local variables are inferred from usage, instance fields default to the public label {}, and method parameters are treated as "implicitly labeled". Thus, general methods like `add(x, y)` do not need to be duplicated for different security levels. Method return values and exceptions receive the join of all parameter labels to ensure the safe closure of information flow.

3. JFlow primarily relies on static type-label checking, but includes limited runtime checks for situations that static analysis cannot resolve. The main mechanism is the `switch-on-label` construct: at runtime, the label of an expression is evaluated, and the system executes the first branch with a matching or stricter label. If no match is found, a controlled exception is thrown to prevent leakage [9]. Additionally, JFlow treats `principal` as a first-class value, which can be used in policy declarations and `actsFor` reasoning, allowing changes in subject hierarchy to be safely verified at compile time.

4. To support practical use, JFlow enables information declassification. If a process has the authority to act for a subject p, it may remove the security policy owned by p (including its `actsFor` chain), thereby lowering the data's label and releasing the information safely. This feature is essential, as strict enforcement without declassification would often be too restrictive for real-world applications[10].

5. Earlier IFC systems lacked support for class-level label parameterization and often simulated "polymorphism" through runtime coercion, introducing new covert channels [15]. JFlow addresses this limitation by supporting parameterized labels and subjects. Classes and interfaces can declare parameters like `<L>` or `<P>` for static type safety across labels or principals. For example, `Vector<L>` becomes a lightweight dependent type. To ensure well-formedness, only immutable values may be used as parameters, and covariance is restricted (e.g., such parameters may not appear in mutable fields or method arguments). Additionally, declassification authority may be bound to class parameters via the `authority` clause, which requires the instantiating context to possess the appropriate agency rights[10].

4.1.3 Main contributions

Compared with the earlier purely static security level model, JFlow introduces two mechanisms, label polymorphism and controlled degradation, while maintaining the information flow check at compile time. Label polymorphism allows the same piece of code to serve multiple security labels, greatly improving reusability. Controlled degradation provides an auditable secret release channel, which not only avoids "misblocking that should not be blocked" but also maintains a strict confidentiality boundary. Flow achieves a better balance between expressiveness and false positive rate [10].

In addition, although JFlow needs to add dedicated extensions to the Java compiler, it moves all security checks forward to the compilation stage, so there is almost zero overhead during execution. This model is particularly suitable for systems that are performance-sensitive or must pass formal verification, such as high-frequency trading and embedded firmware. Although runtime solutions such as Jeeves are more flexible, they may be limited by solution overhead in strict real-time or resource-constrained scenarios. Carapace, by contrast, favors transparency and dynamism, whereas JFlow trades that flexibility for earlier, cost-free enforcement in performance-critical domains.

4.2 Jeeves

Jeeves [19] represents another approach to integrating information flow control into existing languages for use in practical projects. Jeeves is still a novel *domain-specific language*, but it is embedded in an existing language—Scala (and later, Python)—and can be used as a *library*. Jeeves is also primarily a *dynamic* approach to IFC, using a runtime system based on SMT-solving to dynamically check that appropriately values are emitted from output channels like printing.

4.2.1 An overview of Jeeves

Jeeves a *functional constraint language* implemented as an embedded domain-specific language in Scala. Jeeves is primarily a *dynamic* IFC system and operates on *output channels* (visible program side-effects like printing to the screen, writing to a file, or even sending an email) at runtime. In particular, it checks the state of security labels on subjects (usually users) and objects (some information) at those output channels and dynamically determines what output to emit.

Jeeves is based on three main concepts: *sensitive values*, *policies*, and *contexts*. These concepts are essentially just iterations on the core idea of security label tracking, realized in a particular practical form.

Sensitive values in Jeeves are pairs of values $\langle v_\bot, v_\top \rangle_l$, where $v_\bot$ is the low-confidentiality value (which should be revealed in low-confidentiality settings), $v_\top$ is the high-confidentiality value (which should be revealed in high-confidentiality settings), and l is a variable representing the security level (either $\top$ or $\bot$). Policies are constraints on the values of level variables (which are just security labels, in the classical parlance) which declaratively specify how and when to set level variables to $\top$ or $\bot$ and, by implication, which value ($v_\bot$ or $v_\top$) can be revealed at what output channels. Policies may refer to a context, which characterizes an output channel and is used to associate policies with output channels (i.e. "apply this policy on output channels with this context").

The *constraint* part of Jeeves is perhaps the most important part of the system: it refers to the fact that Jeeves is able to *automatically enforce* the high-level policies written by the programmer (in a decidable logic) using a constraint solver. This approach directly addresses the programmer usability problem: as long as the programmer can properly specify the appropriate confidentiality policies and annotate the proper security labels for some set of entities, the system can enforce a degree of information flow security without further manual effort.

4.2.2 Lessons on language integration

A major contribution of Jeeves is not a language-specific integration technique, but rather a general approach to making IFC more practical.

On the programmer usability side, it proposes a model in which the core program logic and the enforcement of security and privacy policies are *separate components*. Additionally, it tries to further ease developer burden by making policy enforcement *declarative*: the developer simply specifies what a policy should be (in the Jeeves domain-specific language) and the system *automatically* ensures that values sent on output channels adhere to that policy.

On the language integration side, its focus on dynamic runtime checks rather than static compile-time checks, allow Jeeves to be more easily integrated as a *library* into existing languages without specialized language extensions. In particular, Jeeves provides the following guidance on language integration:

1. Finding ways to implement IFC as a *library* can be helpful in easing language integration. Such a library generally needs to enforce checks on side-effecting operations like I/O.

2. Dynamic IFC is more immediately practical as a language library, since its checks are performed at runtime and thus do not require changes to the host language's compiler. The deeper changes needed for fully static IFC often require the use of compiler extensions or certain advanced type system features not currently present in most mainstream languages.

3. Information flow control requires certain semantic language properties to hold: properties that mainstream languages do not have. Implementing an *embedded domain-specific language* in another, more mainstream host language is one way to obtain the benefits of a custom language with the desired semantics, while still having access to a broader language ecosystem. It allows pieces of an application to be written in the embedded DSL and integrated into a larger program written in the host language. This points to a need for languages that have good support for implementing embedded domain-specific languages, i.e. that have features like operator overloading and a fleshed-out metaprogramming/macro system.

4. A *declarative* approach to policy specification and enforcement can ease developer burden by reducing the amount of annotations and custom enforcement logic. In particular, *SMT solvers* are an interesting tool for solving the constraints necessary in such an approach. This points to a need for better language integration with such solvers (in fact, the Jeeves authors had to implement a custom embedding of the Z3 SMT solver into Scala before implementing the higher-level Jeeves library).

The Jeeves approach to providing a form of dynamic IFC is fairly portable across languages, since it requires very few language-specific features. Any language that allows things like simple macros and operator overloading (which many mainstream languages like Python do allow) and that has good integration with an SMT solver (which often uses those same features, like macros and operator overloading), can provide Jeeves-style IFC. Indeed, the authors of the original Scala version of Jeeves later ported the idea to Python (perhaps hoping for broader adoption).

Of course, the downside is that this approach is still almost purely *dynamic*, which results in performance overhead that may be unacceptable for certain applications. It is also more difficult to get certain formal guarantees with a dynamic approach, such as implicit flow and covert channel detection (although not theoretically impossible) [11, 1].

4.3 Carapace

Carapace is a *static-dynamic* hybrid IFC method [2]. It is offered as a Rust *library* with integrity labels that function with unaltered Rust and its compiler. With the exception of two types of code that need to be trusted and audited: (1) code specifically marked as declassifying or endorsing data, and (2) explicitly unsafe code (code that uses Rust's unsafe keyword), application code that uses Carapace is untrusted (i.e., not included in the TCB) because Carapace guarantees noninterference.

4.3.1 An overview of Carapace

Carapace employs the classic non-interference IFC formal model, which uses secrecy and integrity labels to constrain data flow and supports trusted policy exceptions (i.e. decryption and

endorsement). Carapace supports the use of both static tags (handled by the Rust type system at compile time) and dynamic tags (managed at runtime), with separate type representations and join operations.

Carapace achieves strong noninterference guarantees without requiring modifications to the underlying language or compiler by leveraging Rust's existing features such as macros, type system, and encapsulation.

Its core mechanism involves labeling every value with both a secrecy label (indicating who can read it) and an integrity label (indicating who can influence it). Access to these labeled values is controlled through lexically scoped secure blocks, with type and runtime checks ensuring that label constraints are respected.

Controlled exceptions to these rules, such as declassification and endorsement, are permitted only within trusted blocks and are governed by the principal's capabilities. This design can be ported to other languages that support an equivalently rich type system, lexical scoping, and runtime policy enforcement, following the central principle of encapsulating sensitive data in security-aware types and rigorously restricting their flow.

4.3.2 Lessons on language integration

1. Library-based IFC is viable but constrained: Carapace proves that fine-grained hybrid IFC can be enforced as a library in Rust without modifying the language or compiler. As long as the host language has a sufficiently expressive type system, macro capabilities, and encapsulation features, it is possible to enforce powerful security properties through a library alone.

2. Lexical scoping enables secure reasoning: Lexically scoped secure blocks simplify both enforcement and reasoning about information flow, while languages with global or loosely scoped control flow (e.g., dynamic languages or callback-heavy systems) make secure flow tracking harder. Therefore, a language that supports clear scoping constructs (e.g., Rust blocks) is helpful.

3. Noninterference enforcement requires control over side effects: Carapace restricts side effects, including hidden ones (destructors, deref, operator overloads), using Rust's trait system. Without side-effect tracking or restrictions, it is difficult to enforce IFC soundly—especially for implicit flows.

4. Macros and other DSL facilities: Rust macros let Carapace transform syntax and enforce discipline, e.g., disallowing unsafe access outside secure blocks. A macro system or code transformation tool is invaluable for enforcing IFC without modifying the base language. Languages like Java (with annotation processors) can simulate this; languages without macros need AST rewriting or interpreters.

4.3.3 Exploring the Portability of Carapace

Based on the introduction of Carapace, we will explore the possibility of porting it to other languages such as Java and Python.

Carapace in Java To port Carapace to Java, static analysis tools, runtime tagging libraries, code staking, and rewriting capabilities are needed. Generally, Java's type system is not quite expressive enough, lacks a macro mechanism, and is not able to statically track side effects. That said, there are ways to simulate the language features required to port Carapace to Java:

- Strongly typed systems with generalization: Java supports generics and interfaces (e.g., `SecureValue<T>`), but its expressive power is weak and it does not support Rust's trait bounds or higher-order types, which need to be checked at compile time by using Java annotations (e.g., `@SecLabel`, `@IntLabel`) in combination with a static analysis tool (e.g., Checker Framework) to solve this problem.

- Code conversion or macro functions: Carapace uses process macros in Rust to extend `secure_block!` and insert checking code while Java does not have a macro mechanism, but similar functionality can be achieved through: compile-time annotation processor; aspect-oriented programming (e.g., AspectJ); source code conversion tools (e.g., Spoon or the Javac plug-in).

- Encapsulation and Access Control: Carapace utilizes Rust's modular encapsulation to restrict access to `SecureValue` fields. While Java supports private and package access, it cannot restrict access based on context (e.g., whether or not it is in a secure block), and needs to maintain access constraints through code staking and runtime checking.

- Runtime label checking: Carapace uses fast and low overhead tag representations (e.g. 64-bit tags). Java, on the other hand, can represent tags as objects (e.g., `Set<Tag>`), but this can lead to possible performance degradation at runtime due to the high cost of object creation and the lack of underlying memory control. Performance can be optimized by using pooled immutable sets of tags and comparing them to references.

- Ability to control side effects: Carapace requires functions in secure blocks to be side-effect free, and Rust enforces this restriction through a trait system and negative impl. Rust enforces restrictions through a trait system and negative impl. Java does not have a type-level checking mechanism for side effects. You need to mark pure functions with the `@Pure` annotation or utilize static tools for side-effect checking or disable specific API calls via bytecode stubbing.

Carapace in Python To port Carapace to Python, security wrapper classes, a context manager, a dynamic labeling system, and a permissions context are needed. Python has no type system or side-effect constraints; everything is controlled by the runtime and the development specification. That said, there are also limited ways to simulate the required features in Python:

- Tag encapsulation mechanism: Rust uses `SecureValue<T, S, I>` to encapsulate labeled values. Python can encapsulate arbitrary objects using classes or decorators. Python supports dynamic encapsulation, but there is no way to statically restrict access, so it's up to the developer or a tool to check for it.

- Runtime label checking: Python has no type checking and no compile-time constraints; all IFC checking must be done at runtime. While Python is flexible and can dynamically insert checks, all safety can only be handled at runtime, which lacks performance and robustness.

- Emulating the secure block mechanism: Python has no macro mechanism, but can emulate `secure_block!` semantics with a context manager (the with statement).

- Capabilities and trusted blocks: The set of permissions for the current principal can be maintained through a global context.

- Side effect control is difficult to achieve: Python functions do not have side-effect markers, and there is no way to disable I/O, network calls, or printing. So you can't use traits or a type system to restrict behavior like Rust does. Therefore, manual auditing or static analysis tools (such as AST checking) are required when side effects may be implicit in a secure block.

4.4 Comparison of Integration Techniques

These instances of IFC integration with existing languages have many commonalities. All use the idea of security labels and track those labels through program executions (either statically, dynamically, or both), try to automate parts of annotation burden (using type inference or constraint solving), and aim to separate code for core functionality from code for security enforcement in an effort to untangle cross-cutting concerns. These are all excellent ideas to improve the efficiency and usability of language-based IFC.

Interestingly, they also all focus on providing *termination-insensitive non-interference* (a full taxonomy of non-interference can be found in [9]), which is a relaxed form of non-interference in which high-secrecy values *can* affect the termination of a program (so information leaks could happen through program termination).

This is a purely practical decision and a very reasonable one: after all termination is incredibly difficult to track in practice and very few languages require all programs to terminate, given how restrictive that would be. Even languages that do require termination—often dependently-typed languages for formal proof, such as Agda and Rocq—cannot prevent programs from terminating abnormally, i.e. through a stack overflow [14, 2].

It is also interesting that these three integration methods represent three broad categories of approaches to integration:

1. JFlow represents the "write a new language with a novel type-checker" approach. It achieves a degree of language integration by compiling to a mainstream language—Java. This approach is the most flexible and expressive, since it gives the language developer full control over what language features are present and what forms of IFC to support. However, one could argue that it has the highest barrier to entry for programmers.

2. Carapace represents the "hack the type system of an existing language to support effect tracking" approach. This integrates directly into the ecosystem of an existing language, but comes at the cost of relying on type-system features that may be bleeding-edge or otherwise unfamiliar. Also, not all type systems in existing languages are amenable to encoding the features required for state-of-the-art language-based IFC.

3. Jeeves represents the "delegate most checks to runtime using some constraint solver" approach. This is the most flexible approach, since it does not rely at all on the host language's type system and thus sidesteps many of the most problematic barriers to language integration. However, it also abandons many of the benefits of static checking, such as zero runtime overhead or soundness in the case of flow-sensitive information flow security.

4.5 Guidance for Future IFC Integration

Generally, a library-level approach is desirable for practical adoption. Requiring programmers to use a novel language, even if it is based on a familiar one like Java, may be too much to ask. For example, although the Jif/JFlow project supplies an incredibly rich, state-of-the-art language-based IFC system and compiles directly to Java through type erasure, it has not seen

widespread industry adoption[4]. As such, most recent attempts to make language-based IFC practical have focused on providing a small, library-level DSL in an existing language [19, 2, 13].

If purely dynamic IFC is acceptable, the library-level approach is already practical. Taking inspiration from Jeeves, any language that has a macro system (or other facilities for implementing embedded DSLs) can implement a library that dynamically tracks and checks that side-effects do not violate the specified security labels at runtime, using some kind of ad-hoc constraint solver or general-purpose SMT solver.

However, if any form of static checking is desired, the library-level approach becomes much more difficult. In essence, what is required amounts to implementing an *effect tracking type system*, which is the core functionality of static, language-based IFC. In many modern languages, this is simply impossible without having to resort to heavy-handed techniques like compiler or interpreter modification. However, it *is* much easier in pure, functional languages that already strictly control side-effects (like Haskell). Indeed, *monads*—a way to handle side-effects in a purely functional language—have resulted in a rich tradition of library-level IFC implementation in Haskell [13, 18].

Broadly speaking, without more expressive mainstream languages, particularly languages that have full-fledged effect systems (and perhaps even dependent types [7]), practical implementation of expressive, flexible, state-of-the-art static IFC will remain difficult, if not impossible. Even in Rust, a more recent language with a relatively rich type system (compared to other similarly popular languages), static IFC can only just barely be implemented in a research project [2] and only through building an ad-hoc effect tracking type system using bleeding-edge features of the Rust compiler. There may still be ways to go, in this respect.

5 Conclusion

We have presented a brief survey of language-based information flow control. In particular, we cover a bit of historical context, skim the basic theoretical foundations of this line of work, and analyze three different methods of integrating IFC into existing languages. We also extract a few lessons from these methods and distill some concrete guidance for future work in language integration of IFC.

References

[1] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 165–178, 2012.

[2] Vincent Beardsley, Chris Xiong, Ada Lamba, and Michael D Bond. Carapace: Static–dynamic information flow control in rust. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):364–392, 2025.

[3] Matthew A Bishop. The art and science of computer security, 2002.

[4] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

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

[4]Whether this is the fault of the research community or of industry priorities is up for debate.

[6] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

[7] Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov. A dependently typed library for static information-flow control in idris. In *Principles of Security and Trust: 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 8*, pages 51–75. Springer, 2019.

[8] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, August 1976.

[9] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software safety and security*, pages 319–347. IOS Press, 2012.

[10] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[11] Thomas FJ-M Pasquier, Jean Bacon, and Brian Shand. Flowr: aspect oriented programming for information flow control in ruby. In *Proceedings of the 13th international conference on Modularity*, pages 37–48, 2014.

[12] Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation*, pages 63–74, 2009.

[13] Alejandro Russo. Functional pearl: two can keep a secret, if one of them uses haskell. *ACM SIGPLAN Notices*, 50(9):280–288, 2015.

[14] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 186–199. IEEE, 2010.

[15] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[16] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 352–365. Springer, 2009.

[17] R.S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[18] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 95–106, 2011.

[19] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. *ACM SIGPLAN Notices*, 47(1):85–96, 2012.

[20] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. Making information flow explicit in histar. *Communications of the ACM*, 54(11):93–101, 2011.

[21] Lantian Zheng and Andrew C Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6:67–84, 2007.