# A Type System for Optimizing Dynamic IFC

DANIEL GALÁN PASCUAL, ETH Zurich, Switzerland
FRANÇOIS HUBLET, ETH Zurich, Switzerland
SRĐAN KRSTIĆ, ETH Zurich, Switzerland
ROMAN FISCHER, ETH Zurich, Switzerland
COLIN PFINGSTL, ETH Zurich, Switzerland
DAVID BASIN, ETH Zurich, Switzerland

Dynamic information-flow control (IFC) enforces confidentiality policies at runtime by tagging values with security labels and blocking policy-violating outputs by terminating the running system. Pervasive label tracking and enforcement checks incur high runtime costs, which limits practical IFC deployment to performance-insensitive workloads. We present a novel alternative called MinIF, a type-directed program transformation that statically eliminates the overhead of dynamic IFC for existing systems.

The central contribution of MinIF is a flow-sensitive type system that tracks which sensitive inputs influence a value and whether the enforcement mechanism would accept operations on it, even though the enforced policy is unknown to the type system. Using the type system, MinIF statically predicts enforcement outcomes and removes redundant checks along with the label-tracking code that served them, and we prove that the optimized program preserves both the behavior and the enforcement decisions of the original. For IFC systems with introspection, the optimization is fully automatic, as the introspection queries already present in the program supply all the permission information MinIF needs, with no programmer annotations. Unresolved checks surface as warnings, and the absence of warnings gives developers a static guarantee against enforcement-induced system termination.

We evaluate MinIF on Python programs running on the WebTTC dynamic IFC platform. On benchmarks, MinIF eliminates between 13% and 99% of the enforcement overhead, and compute-intensive workloads that time out under enforcement now complete in milliseconds.

CCS Concepts: • **Security and privacy** → **Information flow control**; *Formal security models*; • **Software and its engineering** → *Compilers*; • **Theory of computation** → *Program analysis*.

Additional Key Words and Phrases: information flow control, flow-sensitive types, program transformation, taint tracking optimization, security enforcement

**ACM Reference Format:**
Daniel Galán Pascual, François Hublet, Srđan Krstić, Roman Fischer, Colin Pfingstl, and David Basin. 2026. A Type System for Optimizing Dynamic IFC. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 324 (October 2026), 28 pages. https://doi.org/10.1145/3839456

## 1 Introduction

Information-flow control (IFC) [1] is a fundamental paradigm for protecting the security of data that programs manipulate. Given an information-flow policy, an IFC mechanism tracks how a program uses and constrains how the data is released to the outside world. An information-flow

Authors' Contact Information: Daniel Galán Pascual, ETH Zurich, Zurich, Switzerland, daniel.galan@inf.ethz.ch; François Hublet, ETH Zurich, Zurich, Switzerland, francois.hublet@inf.ethz.ch; Srđan Krstić, ETH Zurich, Zurich, Switzerland, srdan.krstic@inf.ethz.ch; Roman Fischer, ETH Zurich, Zurich, Switzerland, fischrom@student.ethz.ch; Colin Pfingstl, ETH Zurich, Zurich, Switzerland, cpfingstl@student.ethz.ch; David Basin, ETH Zurich, Zurich, Switzerland, basin@inf.ethz.ch.


ACM 2475-1421/2026/10-ART324
https://doi.org/10.1145/3839456

policy expresses security requirements by associating data values with security labels, such as *secret* and *public*, and specifying the flows permitted between these labels.

IFC mechanisms can be static or dynamic. Static IFC performs a worst-case analysis of information flows in a program's source code and rejects programs that cannot guarantee policy compliance. Due to this conservative, worst-case analysis, static IFC is imprecise and rejects some policy-compliant programs. Dynamic IFC is an enforcement mechanism [2] that tracks data at runtime; it stores labels in memory, checks every program action, and prevents policy-violating ones. While dynamic IFC is generally more precise than static IFC, it incurs significant performance overhead [3].

To offset these drawbacks, *hybrid* IFC approaches [4, 5, 6, 7] combine static source code analysis with dynamic execution monitoring. Most hybrid approaches require new IFC mechanisms that tightly couple static and dynamic checking; e.g., invoking static analysis from within a dynamic monitor. We take a different approach: rather than proposing new dynamic IFC mechanisms, we optimize existing ones automatically through a fully separate compile-time program transformation.

Specifically, we present the MinIF system, whose main component is a transpiler that statically analyzes programs to reduce the dynamic IFC overhead while preserving security and precision. For IFC systems with *introspection* [7, 8, 9, 10], i.e., the ability to query whether an action would violate the policy before its execution, our approach is fully automatic and annotation-free: the introspection queries already present in the program provide the information needed to predict enforcement outcomes and eliminate redundant checks, with no additional programmer effort. More broadly, optional annotations can encode optimization-relevant facts for any IFC-related APIs (e.g., declassifiers, sanitizers, label-raising operations), extending MinIF's reach to non-introspective systems.

We illustrate our approach on a concrete program that uses introspection.

*Example 1.1.* Consider the following Python program run by a dynamic IFC mechanism:

```
1  posts = db.read_posts()   # equivalent to posts = ["Post 1", "Post 2", ...]
2  selected_posts = []
3  i = 0
4  while i < len(posts) and i < 30:
5      post = posts[i]
6      if check(post, action=display, user=current_user):
7          selected_posts += [post]
8      i += 1
9  display(selected_posts, user=current_user)
```

The code is part of a Twitter-like social media application that shows the latest posts to the logged-in user. First, the program reads all available posts from a database, then selects those that can be shown to the user. To implement pagination, the loop processes a maximum of 30 posts before displaying them in the user interface. This will also be our running example throughout the paper.

Since calling `display` outputs information to an external user, a dynamic IFC mechanism may prevent this action to preserve policy compliance. Therefore, the code does not pass the first 30 posts (`posts[:30]`) directly from the database to this routine. Instead, the `check` call at line 6 performs introspection, probing whether a call to `display` with the given post would be policy-compliant. In the context of this loop, this ensures that all elements in `selected_posts` can be shown to the current user. As a result, we can predict that the call to `display` will succeed.

We perform this prediction automatically and without programmer annotations. We leverage this prediction to remove dynamic IFC checks triggered by the call to `display`, since we know the check will succeed here. Furthermore, code that tracks labels can be selectively eliminated when its sole purpose is to enable the removed checks. Statically determining where such checks and label tracking can be safely eliminated is a key novelty of our approach.

Statically predicting the outcome of a dynamic check is difficult because the policy resides only in the enforcement mechanism, which consults it at runtime. Static IFC does not face this problem, as the policy is an input to the analysis. Our setting inverts this. A static analysis must instead infer a check's outcome from evidence found in the program itself.

Our main technical contribution is a flow-sensitive type system [11] that captures this evidence as two complementary facts about each value, computed by full type inference (Section 4): its *influences*, i.e., the sensitive inputs that may affect it; and its *permissions*, i.e., the checks it is already known to pass or fail. The program's introspection queries supply them automatically, and signatures of IFC-related operations can encode them otherwise. The inferred types guide a program transformation (Section 5) that removes the checks predicted to succeed and the label tracking that only served them, and we prove that the transformation preserves both program behavior and enforcement decisions.

Our implementation supports a large class of languages and enforcement mechanisms. Adding a new language-mechanism pair only requires a lightweight adapter, while the core analysis and optimization pipeline is generic. We optimize Python programs for a dynamic IFC mechanism called WebTTC [7, 12] and evaluate the performance gains on practical applications (Section 6).

*Contributions.* With this work, we make both theoretical and practical contributions.

- We present IFIR, an intermediate representation that decouples IFC analysis from specific languages and enforcement mechanisms (Section 4).
- We formalize a flow-sensitive type system for IFIR that, unlike prior IFC type systems, jointly tracks information-flow influences and enforcement permissions, predicting the outcome of each check without access to the enforced policy. This enables the static elimination of enforcement checks even for labeled data and provides a formal guarantee against enforcement-induced terminations (Section 4). For IFC systems with introspection, type inference is fully automatic and requires no annotations.
- We design a program transformation that is the first hybrid IFC optimization with formal proofs of both behavior and enforcement equivalence (Section 5).
- We realize our approach in MinIF (Section 3), a tool pipeline that automatically optimizes programs across diverse language-enforcer pairs using lightweight adapters.
- We evaluate MinIF on Python web applications for WebTTC, showing overhead reductions of up to 99% and compute-intensive workloads that timed out under full enforcement completing in milliseconds (Section 6).

Overall, this is the first approach that uses introspection queries to predict IFC check outcomes and enable the automatic elimination of enforcement checks through type-directed program transformation (Section 7). These performance improvements make IFC mechanisms with strong security guarantees practically deployable.

## 2 Preliminaries

Our approach applies to a broad class of dynamic IFC systems and enforcement mechanisms. To capture this generality, we present the necessary background in an abstract way that encompasses various concrete instantiations found in the literature. Where we introduce such abstractions, we indicate how they relate to specific systems to highlight their practical applicability.

### 2.1 Notation

We write $A^*$ for sequences of elements from set $A$. We write $[]$ for the empty sequence and $[a, b, c]$ for non-empty sequences. We use $\pi[i]$ to obtain the $i$-th element of a sequence. Sequence concatenation is written $\pi_1 \cdot \pi_2$. We write $\pi' \subseteq \pi$ when $\pi'$ is a (possibly non-contiguous) subsequence of $\pi$, meaning that $\pi'$ can be obtained by removing zero or more elements from $\pi$.

We annotate each node of a program's abstract syntax tree with a unique *program location* $\ell$. We write $\xi_\ell$ for a (sub)program $\xi$ annotated with location $\ell$, using the subscript to single out one occurrence of $\xi$ from other occurrences of the same form. Locations are assigned from compiler metadata and are not written by the programmer.

### 2.2 Runtime Enforcement

Runtime enforcement [2, 13] is a security approach where a trusted enforcement mechanism, called an *enforcer*, monitors the executions of an untrusted program, observing the events they generate and taking corrective actions when an imminent violation of the security policy is detected.

We model this using labeled small-step operational semantics [14, 15]. Following Plotkin's structural approach [14], we define a *labeled small-step semantics* as a quintuple $\langle C, E, \overset{\cdot}{\rightsquigarrow}, F, \lightning \rangle$ where $C$ is the set of *configurations*, $E$ is a set of *events* representing side effects and outputs, $\overset{\cdot}{\rightsquigarrow} \subseteq C \times E^* \times C$ is the small-step transition relation producing event sequences, $F \subseteq C$ is the set of *final* configurations, and $\lightning \in C$ is a distinguished *error configuration* with $\lightning \notin F$.

An *execution* is a (possibly infinite) chain $c_0 \overset{\pi_1}{\rightsquigarrow} c_1 \overset{\pi_2}{\rightsquigarrow} \dots$ where each step produces events $\pi_i$. The *trace* $\pi = \pi_0 \cdot \pi_1 \cdot \dots$ collects all events from the execution. A finite execution $c_0 \overset{\pi_0}{\rightsquigarrow} \dots \overset{\pi_{n-1}}{\rightsquigarrow} c_n$ has *normal termination* if $c_n \in F$, *explicit failure* if $c_n = \lightning$, and is *stuck* if $c_n \notin F$, $c_n \neq \lightning$ and there exists no $c'$, $\pi$ with $c_n \overset{\pi}{\rightsquigarrow} c'$. We model undefined behavior as stuck configurations, which occur only in ill-formed programs, while explicit failure (e.g., uncaught exceptions or panics) may occur in well-formed programs.

We model enforcement mechanisms following the standard approach of instrumenting programs to log events whenever the program produces side effects. An *enforcer* is a function $\mu : E \to \{\text{allow}, \text{deny}\}$ that checks events against a security policy [2] and determines whether they are allowed or denied. We focus on stateless enforcers that check events independently, following the basic enforcement model from prior work [2, 13].

Building on this enforcement model, we introduce an *execution-cutting* semantics that terminates the execution as soon as the enforcer detects a security violation. Given a semantics $\mathcal{S} = \langle C, E, \overset{\cdot}{\rightsquigarrow}, F, \lightning \rangle$, the *enforced semantics* for an enforcer $\mu$ is a labeled small-step semantics denoted as $\mathcal{S}|_\mu := \langle C, E \cup \{\lightning\}, \overset{\cdot}{\underset{\mu}{\rightsquigarrow}}, F, \lightning \rangle$ where $\lightning$ is a sentinel event signaling enforcement rejection and $\overset{\cdot}{\underset{\mu}{\rightsquigarrow}}$ is the smallest relation satisfying:

$$\frac{\textsc{Accept} \quad c \overset{\pi}{\rightsquigarrow} c' \qquad \text{filter}_\mu(\pi) = \pi}{c \overset{\pi}{\underset{\mu}{\rightsquigarrow}} c'} \qquad \frac{\textsc{Reject} \quad c \overset{\pi}{\rightsquigarrow} c' \qquad \text{filter}_\mu(\pi) \neq \pi}{c \overset{\text{filter}_\mu(\pi)}{\underset{\mu}{\rightsquigarrow}} \lightning},$$

where $\text{filter}_\mu([e] \cdot \pi) := [e] \cdot \text{filter}_\mu(\pi)$ if $\mu(e) = \text{allow}$ and $\text{filter}_\mu([e] \cdot \pi) := \lightning$ if $\mu(e) = \text{deny}$. This definition embodies a principle, admitting elements in sequence and stopping at the first rejection, defined in Section 4.3.

### 2.3 Dynamic IFC

Non-interference [16] formalizes confidentiality by requiring that high-sensitivity inputs cannot influence low-sensitivity outputs. Since non-interference is a hyperproperty (it quantifies over pairs of executions) [17], direct runtime enforcement is costly [18]. Instead, dynamic IFC soundly approximates non-interference by enforcing a tractable safety property: programs are instrumented to tag values with labels that capture information flows, and the enforcer checks that outputs comply with the allowed information flows, as specified by a policy, by examining the associated labels.

For conciseness, we only focus on *confidentiality*: preventing sensitive inputs from influencing outputs in unauthorized ways. Although we do not consider integrity (preventing untrusted inputs from influencing trusted outputs), we could handle it with modest additional effort [3].

Following standard label-tracking approaches [19, 20], we model labels algebraically. Labels $w \in \mathbb{W}$ form a join semilattice ordered by a *can flow to* relation $\sqsubseteq$, with join $\cup : \mathbb{W} \times \mathbb{W} \to \mathbb{W}$ for combining labels and bottom element $\emptyset \in \mathbb{W}$ (the empty label, indicating no sensitive influence). Computation propagates labels by joining their input labels. A value labeled with $\emptyset$ is not considered sensitive. Concretely, we instantiate $\mathbb{W}$ as the powerset of input sources (e.g., database fields or user inputs), ordered by subset inclusion. This is without loss of generality: Hunt and Sands [11] show that the powerset-of-sources model is *principal*, i.e., any security lattice arises as a quotient of the powerset lattice. Our abstract treatment therefore accommodates any concrete label lattice.

Branching on a sensitive value can leak information through *implicit flows* [20]. Dynamic IFC handles this via a *program context pc* $\in \mathbb{W}^*$, which accumulates the labels of conditions governing the current execution path. Any value produced inside a branch inherits *pc* by joining it into the result label. Concretely, *pc* is maintained as a stack of labels, pushed and popped at branch entry and exit.

To support policy enforcement on labeled data, we model enforcement events as pairs $\langle s, w \rangle$ where $s \in \mathbb{S}$ is a *security scope* (an enforcer-dependent representation of the event's kind, such as action, subjects, and context) and $w$ is a label aggregating the labels of inputs influencing the event. This formulation generalizes various policy models: for example, Amazon's Cedar policy language [21] follows the PARC model (Principal, Action, Resource, Context), while WebTTC [7] aligns with the GDPR and encodes a user (data subject) and data processing purpose. By abstracting the security scope $s$, our model accommodates these diverse policy frameworks.

*Introspection in Dynamic IFC.* Introspection, found in systems such as WebTTC [7] and HLIO [22], allows a program to query the enforcer $\mu$ about whether an event would be accepted and branch on the result without triggering enforcement. We adopt their common structure as a minimal interface (see Section 7 for how each system instantiates it): the program tests $\mu(\langle s, w \rangle)$ and chooses between alternative code paths accordingly, giving the operational semantics direct access to $\mu$.

Introspection introduces an implicit information flow: a deny verdict reveals that the queried value carries restricted labels, so the verdict must carry those labels and be subjected to enforcement. This raises an immediate concern: if the verdict were itself subject to rejection, querying would trigger the very violation the program was trying to avoid. Introspection is therefore only well-formed when the verdict is always *permitted* for the queried scope.

*Example 2.1.* A program queries whether `print(x)` is permitted, then either prints `x` or a placeholder. The verdict is always printable; even a deny does not prevent printing the placeholder.

## 3 The MinIF System

MinIF is an approach and system that reduces dynamic IFC overhead by statically identifying and removing provably unnecessary instrumentation. Dynamic IFC systems introduce overhead from two sources: (1) *label tracking*, propagating label metadata through every operation; and (2) *enforcement checks*, querying the enforcer at sensitive operations. Even when introspection confirms an operation is safe, naive instrumentation still tracks and checks it: tracking in case it enables future enforcement points, and checking despite the predictable outcome. MinIF eliminates both provably redundant checks and propagation that only serves eliminated checks.

### 3.1 Key Idea: Weakening Instrumentation via Static Predictions

Most runtime enforcement approaches synchronize program execution and enforcement: every attempted security-relevant event is logged to the enforcer. While this ensures sound enforcement,

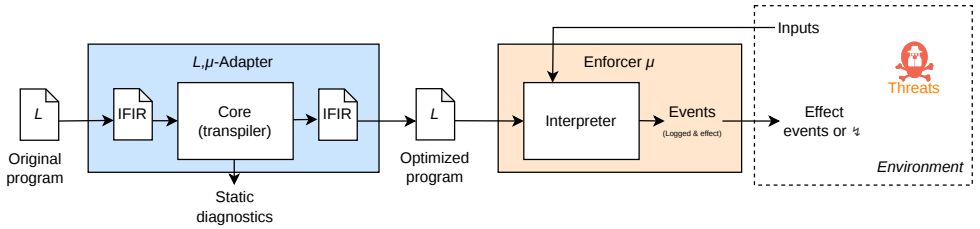


Fig. 1. Component View of the MinIF System

it introduces significant overhead. This tight coupling is sometimes unnecessarily restrictive: when we can *statically predict* that certain events will be accepted by the enforcer, then we can safely omit logging them, thus reducing runtime overhead while preserving security.

*Definition 3.1 (Marked Events).* Given an event set $E$, we define the *marked event set* $E^{\updownarrow} := \{\uparrow e, \downarrow e \mid e \in E\}$, where $\uparrow e$ (read "*log e*") denotes logging event $e$ to the enforcer for validation, and $\downarrow e$ ("*effect e*") denotes executing the event $e$, causing its actual system effects.

An execution trace over marked events (a *marked trace*) is a sequence $\pi \in (E^{\updownarrow})^*$ that records both logging and executing actions. From a marked trace, we extract two views: the *actual trace* $\pi|_{\downarrow} = [e \mid \downarrow e \in \pi]$ contains the events that actually executed (representing observable system behavior), and the *observed trace* $\pi|_{\uparrow} = [e \mid \uparrow e \in \pi]$ contains the events that the enforcer validated. Both $\pi|_{\downarrow}$ and $\pi|_{\uparrow}$ are traces over the unmarked event set $E$, representing different perspectives on the same execution.

In traditional instrumentation approaches, each sensitive operation is preceded by an enforcement check, i.e., produces a paired sequence $\uparrow e \downarrow e$ in the trace, ensuring synchronization where $\pi|_{\downarrow} = \pi|_{\uparrow}$. A trace is *under-logged* if $\pi|_{\downarrow} \not\subseteq \pi|_{\uparrow}$, i.e., some operations bypass enforcement. Under-logging is generally unsafe, but becomes safe when the hidden events would always be accepted: if we can statically guarantee that removing $\uparrow e$ from the trace does not change the enforcer's subsequent decisions, then eliminating this logging preserves the enforcement semantics while reducing overhead.

*Definition 3.2 (Well-Formed Traces).* A marked trace $\pi$ is *well-formed* if each logging $\uparrow e$ is immediately followed by the corresponding execution $\downarrow e$ or by an enforcer rejection:

$$\forall i.\, \pi[i] = \uparrow e \implies \pi[i+1] = \downarrow e \lor \pi[i+1] = \lightning$$

Note that this definition of well-formedness includes a judiciously chosen subset of under-logged traces, widening the scope of reasonable traces beyond the output of traditional instrumentation approaches. However, spurious or out-of-order enforcement checks are still not allowed.

Our approach leverages a type system to predict enforcer acceptance. It creates safe under-logged well-formed traces by selectively removing redundant enforcement checks. The type system tracks two concepts for each value: *influences* (i.e., sensitive inputs affecting the value) and *permissions* (i.e., predictions whether the enforcer would accept or reject operations on the value). From these, we statically predict which enforcement checks are redundant and can be safely eliminated.

*Example 3.3.* Consider `display(sanitize(db.read()))`, where `sanitize` is a label-insensitive trusted declassifier that escapes, masks, or aggregates its input. Although `display` would normally trigger an enforcement check, the output of `sanitize` is statically known to contain no sensitive data. Hence the check is redundant.

### 3.2 System Architecture

MinIF operates on a custom intermediate representation called IFIR (Information Flow Intermediate Representation), which explicitly represents label tracking and enforcement operations (Section 4). Namely, a construct called a *guard* (defined shortly) corresponds to an enforcement check. The optimization pipeline proceeds in three phases.

First, **type inference** (Section 4) simulates dynamic IFC statically to compute influences and permissions for each program point. From these types, we derive sound predictions of enforcement outcomes for each guard: accepted by the enforcer (allow), rejected (deny), or uncertain (maybe). Programs with a deny prediction are statically rejected, as they would violate the policy at runtime. Second, **unguarding** (Section 5.1) removes enforcement checks from operations predicted as allow. These checks are provably redundant, so eliminating them reduces overhead without affecting security. Third, **untracking** (Section 5.2) performs backward slicing from remaining guards to identify label tracking that has become dead code. If a label-introducing construct flows to eliminated guards and to no other security-sensitive operations, both the construct and all propagation along that path can be safely removed, eliminating overhead for values that never reach active guards.

The optimized program is semantically equivalent to the original, producing identical outputs and enforcement decisions. We formalize and prove this in Section 5. Fig. 1 shows how MinIF integrates into an IFC system where the core transpiler implements the optimization pipeline and returns either compile-time errors (program rejected) or an optimized program, the interpreter executes the optimized program with reduced instrumentation, and the enforcer processes the reduced event trace. As usual in the IFC setting, the attacker is part of the environment: they can provide inputs to the program and access the program's source code and (allowed) outputs.

MinIF's design is language- and enforcer-agnostic, isolating the core transpiler from lightweight $L$,$\mu$-adapters that translate between source languages $L$ and IFIR. Supporting new language-enforcer pairs requires only implementing new adapters, while the core remains unchanged. As a case study, we optimize IFC-enabled Python programs for the WebTTC [7] platform, demonstrating the approach's practical applicability. The next sections formalize the minimal core of IFIR. To support realistic applications, an implementation extends this core with external function declarations and interprocedural analysis, discussed in Section 6.

## 4 IFIR and Its Type System

We introduce a new language called Information Flow Intermediate Representation (IFIR). We first present IFIR as a typed lambda calculus extended with label tracking constructs. Then, we extend it with side effects and imperative statements. The result is an expressive language that supports programs in a range of paradigms, including functional and procedural styles. IFIR serves as a vehicle to formalize our second contribution: a flow-sensitive type system. We briefly show IFIR's operational semantics as a foundation for the type system, and then focus on static analysis.

*Selective Label Tracking.* IFIR performs *selective* label tracking, meaning that only some parts of a program and values in memory participate in label propagation. To this end, we partition runtime values into two disjoint sets: *tracked* values $\langle v, w\rangle \in \mathbb{V}\times\mathbb{W}$, which have associated label information, and *untracked* values $v \in \mathbb{V}$, which do not. We denote their union as *extended values* $\widehat{\mathbb{V}} := (\mathbb{V}\times\mathbb{W}) \uplus \mathbb{V}$. Tracked values are subdivided into *sensitive* values, which are labeled by a non-empty label $w \neq \emptyset$, and *public* values, which are labeled by the empty label $w = \emptyset$. Note that *public* values are distinct from *untracked* values: public values carry label information stating that they depend on no sensitive input, while untracked values may depend on sensitive inputs without recording them.

Similarly, many program constructs come in tracked and untracked versions. Tracked constructs operate only on tracked values and propagate labels between them as usual in dynamic IFC, while

| | | | |
|---|---|---|---|
| **Monotype** | $\tau, \rho$ | ::= | $\ell \mid \tau^S \mid \mathbb{1} \mid \mathbb{0} \mid \tau \otimes \rho \mid \tau \oplus \rho \mid \mathbb{U} \mid \alpha \mid \tau \to \rho$ |
| **Polytype** | $\sigma$ | ::= | $\tau \mid \forall \alpha.\, \tau$ |
| **Context** | $\Gamma, \Delta$ | ::= | $(x_1 : \sigma_1),\ \ldots,\ (x_n : \sigma_n),\ \alpha_1,\ \ldots,\ \alpha_m$ |
| **Term** | $t, u$ | ::= | $x \mid \lambda(x : \tau).\, t \mid t(u) \mid \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u \mid k \mid \diamond \mid \bowtie_s t \mid \mathbb{u}(t)$ |

Fig. 2. Syntax of IFIR expressions and types

untracked constructs, syntactically marked with !, operate on untracked values, perform no label propagation, and have better runtime performance. Since enforcement requires label information, however, untracked constructs cannot be safely used in security-critical code.

### 4.1 Syntax and Behavior of IFIR Expressions

IFIR expressions are based on the let-polymorphic lambda calculus extended with label tracking and enforcement-specific constructs. Their syntax is defined in Fig. 2, with new constructs highlighted.

*Pure Computation.* Function application (henceforth function *calls*) and function abstraction have a standard semantics with added label propagation: if the return value $\langle v', w' \rangle$ of a function has been influenced by the argument $\langle v, w \rangle$, then $w' \sqsupseteq w$. Expressions $\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u$ follow the standard call-by-value semantics, with $t$ eagerly evaluated and the resulting value (including its labels) substituted in $u$.

Literals $k$ (Fig. 2) are compile-time constants of a built-in data domain comprising integers, booleans, and text strings. Given that source code constants are not inputs, they produce values that are tracked and public. Tracked values may be explicitly coerced into their untracked analogues using the *untracking* construct $\mathbb{u}(t)$. Once removed, label information cannot be recovered again.

*External Functions and Side Effects.* To enable observable side effects, IFIR provides *external functions*: impure operations supplied by the environment (e.g., I/O, database access) that may violate referential transparency or be partial. Every call $f(\langle v, w \rangle)$ to an external function produces an actual event $\downarrow \langle f(v), w \rangle$ in the trace. The event carries $f(v)$, a particular instantiation of the security scope, and the aggregate label $w$ of all values influencing the call. Enforcers can accept events irrelevant for security without loss of generality.

*Label Tracking and Enforcement.* The *source* construct $\diamond$ takes an input from the environment and tracks it with a fresh unique label $w$. This is the only way to introduce new labels in IFIR. Consequently, if a value is labeled with some $w' \sqsupseteq w$, it has been influenced by this input. The $\diamond$ construct has no untracked variant. In typing rules and examples, we write $\diamond_\ell$ to make the source's location $\ell$ explicit. Since $\ell$ is compiler metadata rather than something written by a programmer (Section 2.1), the grammar in Fig. 2 omits it.

Enforcement checks can be explicitly written using the *guard* construct $\bowtie_s t$, where $s \in \mathbb{S}$ is a security scope literal (e.g., $\langle \texttt{display}, \texttt{current\_user} \rangle$ below). Guards first evaluate $t$ to obtain $\langle v, w \rangle$, then log an event $\uparrow \langle s(v), w \rangle$ in the trace and await the enforcer's verdict. If the enforcer rejects the event, then execution terminates with $\lightning$. Otherwise, the guard yields $\langle v, w \rangle$ and execution continues. Guards have no untracked variant as enforcement requires label information.

*Tracking Discipline.* Label tracking revolves around three roles: *sources* inject fresh labels, *sinks* consume them, and expressions in between *propagate* labels through computation. IFIR provides one construct for each role: $\diamond$ instantiates a source by capturing an external input, and $\bowtie_s$ instantiates a sink by consuming labels for enforcement. Sound tracking requires a critical invariant: *all paths*

*from sources to guards must preserve label tracking.* If labels are lost along a path (via $\mathbb{U}$), a guard that reads labels from an untracked value will crash — a condition we call *tracking mismatch*.

*Example 4.1.* Consider the Python program from Example 1.1 written in IFIR. Even though the IFIR statements used here (assignments, `while`, and `if check`) are only formally introduced in Section 4.3, they are in direct correspondence with the original program.

```
posts := db.read_posts() //equivalent to posts := [◇ "Post 1", ◇ "Post 2", ...]
selected_posts := []
i := 0
while i < len(posts) and i < 30 {
    post := posts[i]
    if check⟨display,current_user⟩ post {
        selected_posts := selected_posts + [post]
    }
    i := i + 1
}
display(⋈⟨display,current_user⟩ selected_posts)
```

In this program, operators and arrays are syntactic sugar for function calls, for example `posts[i]` desugars to `__getitem__(posts, i)`. The **`if check`** construct performs introspection, while the guard on the final `display` call marks the (redundant) enforcement check. The enforcer uses the scope ⟨`display`, `current_user`⟩ to represent the event "display to the current user".

*Operational Semantics.* IFIR's execution constitutes a labeled transition system over configurations $\langle \xi, m, pc \rangle$, where $\xi$ is a statement, $m$ is a memory mapping variables to extended runtime values, and $pc \in \mathbb{W}^*$ (a stack of labels) is the dynamic program context. The small-step relation $\langle \xi, m, pc \rangle \overset{\pi}{\rightsquigarrow} \langle \xi', m', pc' \rangle$ produces marked traces $\pi \in (E^{\updownarrow})^*$. We write $\langle t, m, pc \rangle \Downarrow v$ to mean $\exists pc', m'.\ \langle t, m, pc \rangle \rightsquigarrow^* \langle v, m', pc' \rangle$. Operations propagate labels via join $\cup$, guards log events $\uparrow \langle s(v), w \cup \bigcup pc \rangle$, and external functions effect events $\downarrow \langle s(v), w \cup \bigcup pc \rangle$ if they are allowed by the enforcer. Full definitions and rules are given in the extended version [23].

IFIR programs have three possible outcomes: normal termination (reaching `skip`), explicit failure (reaching ↯), or being stuck. Explicit failure is an enforcement-induced termination and occurs only when guards reject events. The stuck configurations are crashes and they arise from tracking mismatches when untracked values reach label sinks. Well-typed programs never get stuck, although they may terminate when enforcers reject operations.

*Restriction on Aliasing.* IFIR is a value-oriented language with no references or aliasing. When translating from a language with aliasing and mutable references (e.g., Python), our approach readily applies to source programs that do not rely on aliased mutation[1]. Our Python adapter (Section 6) relies on this assumption. Otherwise, adapters must enforce it via defensive copying, runtime alias checking, or an ownership discipline before translation into IFIR.

### 4.2 A Flow-sensitive Type System for IFIR Expressions

We introduce a type system for IFIR expressions that captures two fundamental aspects: the propagation of influences between values (*influences*) and the outcomes of future security checks (*permissions*). We explain the intuition relating types to IFIR's IFC-enabled operational semantics and formalize this notion using abstract interpretation [25].

[1]Programs that avoid aliased mutation can be translated into value-copy equivalents, as no alias observes the mutation [24].

We extend the Hindley-Milner type system [26, 27], addressing the new constructs introduced at the term level. The syntax of types is given in Fig. 2, where new constructs are highlighted. Following Hindley-Milner, we distinguish between monotypes and polytypes, described below.

*Influences and Dynamic Label Tracking.* Recall that the main task of IFIR's dynamic label tracking is to determine which input values influence certain values of interest. Following the powerset-of-sources model from Section 2.3, each input received at program location $\ell$ is tagged with a unique label. Consider an execution where the program has received $n$ input values $v_1, \ldots, v_n$, where each $v_i$ is tagged with a distinct label $w_i$. At any future point in the execution, a value labeled with $w_i$ may be influenced by $v_i$. Moreover, the tracking is sound: a value *not* labeled with $w_i$ is definitely not influenced by input $v_i$.

Our type system statically over-approximates these input-based *influences*. The type of a term $t$ describes which input *locations* may influence the values it produces. Specifically, each label $w \in \mathbb{W}$ carries an associated set of source locations $\mathrm{loc}(w)$ such that $\mathrm{loc}(w \cup w') = \mathrm{loc}(w) \cup \mathrm{loc}(w')$ and $w = \emptyset \iff \mathrm{loc}(w) = \emptyset$. When a term $t$ is associated with a location $\ell$, this indicates that all executions of $t$ produce a value labeled with some $w_i$, where $\ell \in \mathrm{loc}(w_i)$. The type system is *sound*: if location $\ell$ is not in $t$'s type, then no execution of $t$ can produce a value labeled from $\ell$. Note that this over-approximation collapses all distinct inputs from the same program location into a single abstraction.

*Permissions and Security Checks.* Beyond tracking influences, we use our types also to reason about the success or failure of future IFC security checks. We model this through *permissions*, which predict the outcomes of the enforcer's verdicts at guards $\bowtie_s t$. Associating a *positive permission* $+s$ to a term $t$ indicates that all matching guards $\bowtie_s t$ will succeed. Conversely, a *negative permission* $-s$ indicates that they will fail. We lift this notion to permission sets: when a term $t$ is associated with a set of permissions $S$, then for every scope $s$, a guard $\bowtie_s t$ will have a positive verdict if $+s \in S$ and $-s \notin S$, or a negative verdict if $+s \notin S$ and $-s \in S$. Otherwise, no prediction can be made.

*Definition 4.2 (Monotypes).* Monotypes $\tau$ include:

- program locations $\ell$, which represent influences from inputs at that location, and permission-annotated types $\tau^S$, which attach permission sets to types;
- the *public* type $\mathbb{1}$, denoting the absence of influences, and the *never* type $\mathbb{0}$, denoting the uninhabited type;
- type intersections $\tau \otimes \rho$, indicating that values satisfy *both* $\tau$ and $\rho$, and type unions $\tau \oplus \rho$, indicating *either* $\tau$ or $\rho$ depending on the execution branch taken;
- type variables $\alpha$, which are placeholders instantiated during type checking, and function types $\tau \to \rho$; and
- the *untracked* type $\mathbb{U}$, indicating unavailable label information.

A monotype is *ground* when it contains no type variables, and is *non-function* if it is not of the form $\tau \to \rho$. Let Mono (resp., GMono) be the set of all (resp., ground) monotypes.

We take inspiration from union and intersection type systems to track complex label combinations: unions model branching (either one label or another), while intersections model merging (combining labels from both operands). The never type $\mathbb{0}$ represents terms that diverge without yielding a value. Since such terms produce no output, tracking their influences or permissions is meaningless: without a value, there can be no sensitive value to protect. The untracked type $\mathbb{U}$ marks values that *will* be computed at runtime but without label tracking. While $\mathbb{0}$ poses no IFC risk (no value can leak), $\mathbb{U}$ indicates that enforcement is bypassed, and thus the type system cannot verify security.

A typing judgment $\Gamma, \overline{pc} \vdash t : \tau$ asserts that term $t$ has monotype $\tau$ under environment $\Gamma$ and static program context $\overline{pc}$. Environments map variables to polytypes. The program context $\overline{pc}$ is

$$\frac{k \text{ is const.}}{\Gamma \vdash k : \mathbb{1}}\ \text{S-Const} \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \mathrm{inst}(\sigma)}\ \text{S-Var} \qquad \frac{\Gamma \vdash t : \tau \qquad \tau \neq \mathbb{U}}{\Gamma \vdash \bowtie_s t : \tau}\ \text{S-Guard}$$

$$\frac{}{\Gamma \vdash \diamond_\ell : \ell}\ \text{S-Source} \qquad \frac{}{\Gamma \vdash \mathbb{u}(t) : \mathbb{U}}\ \text{S-Untrack}$$

$$\frac{\Gamma, (x : \tau) \vdash t : \rho}{\Gamma \vdash \lambda(x : \tau).\, t : \tau \to \rho}\ \text{S-Abs} \qquad \frac{\Gamma \vdash t : \tau \to \rho \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t\ u : \rho}\ \text{S-App}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma, (x : \mathrm{gen}(\Gamma, \tau)) \vdash u : \rho}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : \rho}\ \text{S-Let}$$

Fig. 3. Type rules for IFIR expressions

relevant when introducing statements; for now, we write $\Gamma \vdash t : \tau$ when $\overline{pc}$ is not needed. Following Hindley-Milner, $\mathrm{inst}(\sigma)$ denotes a fresh monotype instantiation of polytype $\sigma$, $\mathrm{FV}(\tau)$ denotes the free type variables of $\tau$, and $\mathrm{gen}(\Gamma, \tau)$ generalizes $\tau$ to the polytype $\forall \overline{\alpha}.\, \tau$ where $\overline{\alpha} = \mathrm{FV}(\tau) \setminus \mathrm{FV}(\Gamma)$. The typing rules are given in Fig. 3, where rules with highlighted names are the IFIR-specific extensions and the remaining rules are standard Hindley-Milner.

The type rules in Fig. 3 have an implicit check ensuring that tracked and untracked constructs and values match. That is, tracked constructs (e.g., a function call $f(\cdots)$) expect all types appearing in the rule not to be $\mathbb{U}$, while untracked constructs (e.g., $f!(\cdots)$) expect all types to be $\mathbb{U}$. This typing discipline ensures that well-typed programs are free from tracking mismatches and hence from runtime crashes.

*Example 4.3.* The expression $f!(\diamond)$ is untypable as $\diamond$ always has a type distinct from $\mathbb{U}$, but the untracked function call expects the argument to be untracked. However, $f(\diamond_\ell)$ and $f!(\mathbb{u}(\diamond))$ are typable with, respectively, types $\ell$ and $\mathbb{U}$.

*Abstract Interpretation.* Formally, our type system is an abstract interpretation of the dynamic label tracking semantics. The *concrete domain* is the set $\widehat{\mathbb{V}} = (\mathbb{V} \times \mathbb{W}) \uplus \mathbb{V}$ of extended runtime values defined at the start of this section. The *abstract domain* consists of *ground* monotypes $\tau$, which over-approximate sets of possible concrete values.

*Definition 4.4 (Abstraction and Concretization Functions).* We define a *concretization function* $\gamma : \mathrm{GMono} \to \mathcal{P}(\widehat{\mathbb{V}})$ mapping each type to the set of concrete values it represents:

$$\gamma(\mathbb{1}) := \{\langle v, \emptyset\rangle \mid v \in \mathbb{V}\} \quad \gamma(\mathbb{0}) := \emptyset \quad \gamma(\mathbb{U}) := \mathbb{V} \quad \gamma(\tau \oplus \rho) := \gamma(\tau \otimes \rho) \cup \gamma(\tau) \cup \gamma(\rho)$$

$$\gamma(\tau^S) := \{\langle v, w\rangle \in \gamma(\tau) \mid \text{permissions hold for } \langle v, w\rangle \text{ and } S\}$$

$$\gamma(\ell) := \{\langle v, w\rangle \in \mathbb{V} \times \mathbb{W} \mid \mathrm{loc}(w) = \{\ell\}\}$$

$$\gamma(\tau \otimes \rho) := \{\langle v, w_1 \cup w_2\rangle \in \mathbb{V} \times \mathbb{W} \mid \langle v, w_1\rangle \in \gamma(\tau), \langle v, w_2\rangle \in \gamma(\rho)\}$$

$$\gamma(\tau \to \rho) := \{\langle f, w\rangle \in \text{Closures} \times \mathbb{W} \mid \forall \langle v, w'\rangle \in \gamma(\tau).\, f(\langle v, w'\rangle) \in \gamma(\rho)\}$$

The permission constraint in the definition of $\gamma(\tau^S)$ states that, for every scope $s$, if $+s \in S$ then evaluating $\bowtie_s \langle v, w\rangle$ never causes the enforcer to interrupt execution, and if $-s \in S$ then it always does. See the full definition in the extended version [23].

In the definition of $\gamma(\tau \to \rho)$, closures are base values, semantic partial functions $\widehat{\mathbb{V}} \rightharpoonup \widehat{\mathbb{V}}$ that encapsulate the memory, program counter, and environment of the function body. Thus, $f(\langle v, w'\rangle)$ denotes their application and is undefined when the computation diverges. Closures can be tracked like other values and their labels influence their results. The rules of Fig. 4 and the definitions in the extended version [23] make this precise.

$$\frac{\rho \equiv \tau}{\tau \equiv \rho} \quad \frac{\tau \equiv \rho \qquad \rho \equiv o}{\tau \equiv o} \quad \frac{\tau_1 \equiv \rho_1 \qquad \tau_2 \equiv \rho_2}{\tau_1 \to \tau_2 \equiv \rho_1 \to \rho_2} \quad \frac{\tau_1 \equiv \rho_1 \qquad \tau_2 \equiv \rho_2}{\tau_1 \otimes \tau_2 \equiv \rho_1 \otimes \rho_2} \quad \frac{\tau_1 \equiv \rho_1 \qquad \tau_2 \equiv \rho_2}{\tau_1 \oplus \tau_2 \equiv \rho_1 \oplus \rho_2}$$

$$\frac{S^- = \emptyset}{\mathbb{1}^S \overset{\blacktriangleright}{\equiv} \mathbb{1}} \quad \frac{\{+s, -s\} \subseteq S}{\ell^S \overset{\blacktriangleright}{\equiv} \mathbb{0}} \quad \ell^S \otimes \ell^{S'} \blacktriangleright \ell^{(S^+ \cap S'^+) \cup (S^- \cup S'^-)} \quad \frac{\tau \text{ non-function}}{\tau \otimes (\rho \to o) \blacktriangleright \rho \to (\tau \otimes o)}$$

$$\tau \equiv \tau \quad \tau \bullet \rho \equiv \rho \bullet \tau \quad (\tau \bullet \rho) \bullet o \equiv \tau \bullet (\rho \bullet o) \quad \tau \bullet \tau \overset{\blacktriangleright}{\equiv} \tau \quad \text{for } \bullet \in \{\oplus, \otimes\}$$

$$\mathbb{0} \oplus \tau \overset{\blacktriangleright}{\equiv} \tau \quad \mathbb{1} \otimes \tau \overset{\blacktriangleright}{\equiv} \tau \quad \mathbb{0} \otimes \tau \overset{\blacktriangleright}{\equiv} \mathbb{0} \quad \mathbb{U} \otimes \tau \overset{\blacktriangleright}{\equiv} \mathbb{U} \quad \tau \otimes (\rho \oplus o) \overset{\blacktriangleright}{\equiv} (\tau \otimes \rho) \oplus (\tau \otimes o)$$

$$\ell^{\mathbb{0}} \overset{\blacktriangleright}{\equiv} \ell \quad (\tau \otimes \rho)^S \overset{\blacktriangleright}{\equiv} \tau^S \otimes \rho^S \quad (\tau \oplus \rho)^S \overset{\blacktriangleright}{\equiv} \tau^S \oplus \rho^S \quad \ell \oplus \ell^S \overset{\blacktriangleright}{\equiv} \ell$$

Fig. 4. Properties of type equivalence ( sound approximations and exact equivalences).

The abstraction function $\alpha$ maps each tracked value $\langle v, w \rangle$ with $\mathrm{loc}(w) = \{\ell_1, \ldots, \ell_k\}$ to the type $\alpha(\langle v, w \rangle) := (\ell_1 \otimes \cdots \otimes \ell_k)^{P(w)}$, where $P(w) := \{+s \mid \mu(s, w) = \text{allow}\} \cup \{-s \mid \mu(s, w) = \text{deny}\}$ encodes the enforcer's verdicts, and $\alpha(v) := \mathbb{U}$ for untracked values.

*Example 4.5.* Consider the function `ite(c, t, e)`, which returns `t` if `c` is true and `e` otherwise. The program `ite(cond, x + y, y)` returns either `x + y` or `y` depending on `cond`. Under dynamic label tracking, the result carries the labels of `cond` combined with those of the selected branch: either `x` and `y` together, or `y` alone. Statically, this is represented by the type signatures:

$$\texttt{ite} : \forall \alpha. \forall \beta. \forall \gamma.\ \alpha \to \beta \to \gamma \to \alpha \otimes (\beta \oplus \gamma)$$
$$\bullet \ \texttt{+} \ \bullet : \forall \alpha. \forall \beta.\ \alpha \to \beta \to \alpha \otimes \beta$$

Given the environment $\Gamma := \{\mathsf{cond} \mapsto \mathbb{1}, \mathsf{x} \mapsto \ell_1, \mathsf{y} \mapsto \ell_2\}$, the program has type $\mathbb{1} \otimes ((\ell_1 \otimes \ell_2) \oplus \ell_2)$.

*Definition 4.6 (Polytypes).* Polytypes $\sigma$ extend monotypes with universal quantification $\forall \alpha.\tau$, enabling polymorphic typing where type variables $\alpha$ can be instantiated with concrete types. Type instantiation preserves type structure, including permission annotations. For example, instantiating $\forall \alpha.\ \alpha^{+s} \to \alpha$ with $\ell$ yields $\ell^{+s} \to \ell$.

Following Hindley-Milner conventions, we write the never type as $\forall \alpha.\ \alpha$ in the polytype syntax. It is definitionally equal to the never monotype $\mathbb{0}$ and represents the same uninhabited type.

*Principal Types.* Many syntactically distinct ground types have identical semantic interpretations. We formalize this through type equivalence.

*Definition 4.7 (Type Equivalence).* Two ground monotypes are semantically *equivalent*, denoted $\tau \equiv \rho$, if and only if $\gamma(\tau) = \gamma(\rho)$.

Type equivalence is a congruence with respect to type constructors. Moreover, the atomic type operations form a commutative semiring under type equivalence, with multiplication $\otimes$, addition $\oplus$, multiplicative identity $\mathbb{1}$, and additive identity $\mathbb{0}$. Key semiring properties (Fig. 4; the significance of $\overset{\blacktriangleright}{\equiv}$ and $\blacktriangleright$ will become clear shortly) follow from the set-theoretic properties of $\gamma$ and provide familiar reasoning principles: commutativity, associativity, and distributivity. In the figure, we write $S^+$ and $S^-$ for the positive and negative parts of a permission set $S$.

These axioms induce a normalization procedure. We orient the equivalences marked with $\overset{\blacktriangleright}{\equiv}$ as left-to-right rewrite rules, working modulo AC (associativity and commutativity of $\otimes$ and $\oplus$). Applied exhaustively, these yield a *normal form* $\mathrm{nf}(\tau)$ in disjunctive normal form (DNF), i.e., a union of intersections of atomic types. Implementations additionally benefit from *sound approximation*

| | | | |
|---|---|---|---|
| **Term** | $t, u$ | $::=$ | $\lambda(x : \tau).\ (\xi\ ;\ t) \mid \ldots$ (Fig. 2) |
| **Statement** | $\xi$ | $::=$ | $t \mid \texttt{skip} \mid x\ \texttt{:=}\ t \mid \xi_1\ ;\ \xi_2 \mid \texttt{if}\ t\ \{\ \xi_1\ \}\ \texttt{else}\ \{\ \xi_2\ \}$ |
| | | $\mid$ | $\texttt{while}\ t\ \{\ \xi\ \}$ |
| | | $\mid$ | $\texttt{if check}_s\ t\ \{\ \xi_1\ \}\ \texttt{else}\ \{\ \xi_2\ \}$ |

Fig. 5. Syntax of IFIR statements

*rules* $l \blacktriangleright r$, which keep normal forms small and amenable. Such rules only enlarge the concretization, i.e., $\gamma(l) \subseteq \gamma(r)$ rather than $\gamma(l) = \gamma(r)$, so every value of the original type inhabits the rewritten type. The *approximate normal form* $\mathrm{anf}(\tau)$ extends nf with these rules. The highlighted rules in Fig. 4 are of this kind, and anf can be used instead of nf to keep type inference efficient.

Lemma 4.8. *The procedure* nf *terminates and is sound for type equivalence:* $\tau \equiv \mathrm{nf}(\tau)$*, hence* $\mathrm{nf}(\tau) = \mathrm{nf}(\rho)$ *implies* $\tau \equiv \rho$*. Furthermore,* anf *terminates and is sound as an approximation:* $\gamma(\tau) \subseteq \gamma(\mathrm{anf}(\tau))$*, with equality whenever no approximation rule fires.*

Proof. All $\overset{\blacktriangleright}{\equiv}$ rules are $\gamma$-equivalences and the approximation rules only enlarge the concretization. Both procedures terminate by the same lexicographic measure. The full proof is in the extended version [23]. □

Notably, the semiring lacks absorption laws ($\tau \oplus (\tau \otimes \rho) \not\equiv \tau$), distinguishing it from a lattice. Type unions preserve *which combinations* of influences can take place depending on the execution branch taken, not just their presence – essential for permission reasoning: $\ell_1$ and $\ell_1 \otimes \ell_2$ may have different enforcement outcomes, so $\ell_1 \oplus (\ell_1 \otimes \ell_2)$ cannot be simplified without losing precision.

Normal forms serve as *principal types*: inference compares them syntactically and treats types with equal normal forms identically. On the Hindley-Milner fragment of arrows and type variables, the inferred polytypes are principal in the standard sense [26]. Security types enter every comparison through their normal forms, so inferred environments are unique up to normal-form equality. Whether principality extends to full type equivalence remains open, since normal forms are sound but not complete for $\equiv$ (Lemma 4.8).

*Example 4.9.* In Example 4.5 the program `if cond then x + y else y` initially has type $\mathbb{1} \otimes ((\ell_1 \otimes \ell_2) \oplus \ell_2)$, which normalizes to the principal type $(\ell_1 \otimes \ell_2) \oplus \ell_2$ by distributivity and $\otimes$'s identity.

*Inference.* Type inference for IFIR follows the Hindley-Milner discipline: the typing rules are syntax-directed and generate unification constraints, which are solved by standard unification. The only non-standard aspect is that unification operates modulo type normalization: permission expressions and label types are reduced to a normal form before being compared, accounting for the algebraic structure of the type language. We implement this algorithm as part of our prototype (Section 6). For simplicity, the formal presentation omits user-provided type annotations, but they are straightforward to incorporate as unification constraints. The one exception is lambda parameter annotations ($\lambda(x : \tau).\ t$), which the formal system requires but which the implementation infers, making all annotations optional in practice.

### 4.3 IFIR Statements

We now extend IFIR expressions with imperative statements. While the expression fragment established the core algebraic principles, statements add control flow and mutable state capabilities that are central to modeling real-world applications with existing dynamic IFC systems, although they require more nuanced typing rules.

The statement language, defined in Fig. 5, combines standard imperative constructs with IFC-specific introspection primitives. Sequencing, assignments $x$ `:=` $t$, conditionals, and `while` loops

$$\textsc{S-Skip}\ \frac{}{\Gamma;\overline{pc} \vdash \texttt{skip} \dashv \Gamma} \qquad \textsc{S-Seq}\ \frac{\Gamma;\overline{pc} \vdash \xi_1 \dashv \Gamma' \qquad \Gamma';\overline{pc} \vdash \xi_2 \dashv \Gamma''}{\Gamma;\overline{pc} \vdash \xi_1\ ;\ \xi_2 \dashv \Gamma''} \qquad \textsc{S-Expr}\ \frac{\Gamma;\overline{pc} \vdash t : \tau}{\Gamma;\overline{pc} \vdash t \dashv \Gamma}$$

$$\textsc{S-Assign}\ \frac{\Gamma;\overline{pc} \vdash t : \tau \qquad \tau \text{ non-function}}{\Gamma;\overline{pc} \vdash x := t \dashv \Gamma\left[x \mapsto \bigotimes \overline{pc} \otimes \tau\right]} \qquad \textsc{S-If}\ \frac{\Gamma;\overline{pc} \vdash t : \tau \qquad \Gamma;\tau \cdot \overline{pc} \vdash \xi_1 \dashv \Gamma_1 \qquad \Gamma;\tau \cdot \overline{pc} \vdash \xi_2 \dashv \Gamma_2}{\Gamma;\overline{pc} \vdash \texttt{if}\ t\ \{\ \xi_1\ \}\ \texttt{else}\ \{\ \xi_2\ \} \dashv \Gamma_1 \oplus \Gamma_2}$$

$$\textsc{S-While}\ \frac{\forall i. \qquad \Gamma_i;\overline{pc} \vdash t : \tau_i \qquad \Gamma_i;\tau_i \cdot \overline{pc} \vdash \xi \dashv \Gamma_i'}{\Gamma_0;\overline{pc} \vdash \texttt{while}\ t\ \{\ \xi\ \} \dashv \Gamma_n} \quad \text{by } \Gamma_{i+1} = \Gamma_i' \oplus \Gamma_0 \text{ until } \Gamma_{n+1} = \Gamma_n \text{ where } 0 \le i \le n$$

$$\textsc{S-IfCheck}\ \frac{\Gamma;\overline{pc} \vdash t : \tau \qquad \Gamma_1;\overline{pc_1} \vdash \xi_1 \dashv \Gamma_1' \qquad \Gamma_2;\overline{pc_2} \vdash \xi_2 \dashv \Gamma_2'}{\Gamma;\overline{pc} \vdash \texttt{if}\ \texttt{check}_s\ t\ \{\ \xi_1\ \}\ \texttt{else}\ \{\ \xi_2\ \} \dashv \Gamma_1' \oplus \Gamma_2'} \quad \text{where} \quad \begin{aligned} \Gamma_1 &:= \text{refine}^{+s}(\Gamma, t, \tau) \\ \overline{pc_1} &:= \tau^{+s} \cdot \overline{pc}^{+s} \\ \Gamma_2 &:= \text{refine}^{-s}(\Gamma, t, \tau, \overline{pc}) \\ \overline{pc_2} &:= \text{sub}(\tau^{+s}) \cdot \overline{pc} \end{aligned}$$

$$\textsc{E-Abs-Stmt}\ \frac{\Gamma, (x : \tau);\overline{pc} \vdash \xi \dashv \Gamma' \qquad \Gamma';\overline{pc} \vdash t : \rho}{\Gamma;\overline{pc} \vdash \lambda(x : \tau).\ (\xi\ ;\ t) : \tau \to \rho}$$

Fig. 6. Type rules for IFIR statements

behave as usual with added label propagation. Assigned values carry their expression's labels, the condition's labels influence all values modified in either branch, and loops accumulate labels across iterations. We extend lambda abstractions to permit statement bodies ending in an expression, $\lambda(x : \tau).\ (\xi; t)$, enabling procedural-style function definitions.

Beyond these standard constructs, IFIR introduces the $\texttt{if}\ \texttt{check}_s\ t\ \{\ \xi_1\ \}\ \texttt{else}\ \{\ \xi_2\ \}$ statement for IFC introspection. Namely, it evaluates the expression $t$, then executes $\xi_1$ if the guard $\bowtie_s t$ would succeed (i.e., if the enforcer would allow the operation) and executes $\xi_2$ otherwise.

Our type system mirrors the dynamic program counter that tracks implicit flows (Section 2.2) with a *static program counter* $\overline{pc}$, a stack of types representing the nested control-flow contexts under which an expression or statement executes. We write this stack as $\overline{pc} = [\tau_1, \ldots, \tau_n]$ with $\tau_1$ at the top. The $\overline{pc}$'s *combined influence* is $\bigotimes \overline{pc} := \mathbb{1} \otimes \tau_1 \otimes \tau_2 \otimes \cdots \otimes \tau_n$.

We introduce auxiliary notation required to type statements. The *sub-label closure* $\text{sub}(\tau)$ over-approximates the downward closure of $\gamma(\tau)$ under sub-labels: $\gamma(\text{sub}(\tau)) \supseteq \{\langle v, w\rangle \mid w \subseteq w'' \text{ for some } \langle v, w''\rangle \in \gamma(\tau)\}$. For a normal form $\text{nf}(\tau) = \bigoplus_i \bigotimes_j \ell_{ij}^{S_{ij}}$, it is computed as $\bigoplus_i \bigotimes_j (\mathbb{1} \oplus \ell_{ij}^{S_{ij}})$, making every location optional. For instance, $\text{sub}(\ell^S) = \mathbb{1} \oplus \ell^S$. We write $t \in \text{Var}$ when the term $t$ is a variable, $\Gamma[t \mapsto \tau]$ for updating the binding of such a variable, and $\overline{pc}^S$ for adding the permissions $S$ to all elements of the program context, i.e., $\overline{pc}^S := [\tau_1^S, \ldots, \tau_n^S]$. Finally, the type environment refinement used by S-IfCheck updates a checked binding when the side conditions hold and leaves the environment unchanged otherwise:

$$\text{refine}^{+s}(\Gamma, t, \tau) := \Gamma[t \mapsto \tau^{+s}] \text{ if } t \in \text{Var}, \text{ else } \Gamma$$

$$\text{refine}^{-s}(\Gamma, t, \tau, \overline{pc}) := \Gamma[t \mapsto \tau^{-s}] \text{ if } t \in \text{Var}, \ \bigotimes \overline{pc} \equiv \mathbb{1} \text{ and } \tau \equiv \ell^S; \text{ else } \Gamma$$

The type system for statements is flow-sensitive, inspired by previous work from Hunt and Sands [11]. Judgments take the form $\Gamma \vdash s \dashv \Gamma'$, indicating that executing statement $\xi$ under environment $\Gamma$ produces an updated environment $\Gamma'$ reflecting any type changes to modified variables.

The typing rules, presented in Fig. 6, extend standard imperative typing with type tracking. The Seq rule threads environments through sequential composition, Assign updates the assigned variable's type, and If types both branches and combines their resulting environments pointwise with the type union $\oplus$. The While rule requires finding a fixed point environment $\Gamma_n$ that is consistent with the loop body, iterating from the initial environment until the types stabilize.

The most involved construct is `if check`, whose typing rule closely mirrors its runtime behavior; so we present the dynamic rule D-IfCheck first. Like a guard $\bowtie_s\ t$, the check evaluates $t$ to $\langle v, w\rangle$ and consults the enforcer. Here $\text{passed}_\mu(s, w \cup \bigcup pc)$, the label-level analogue of the trace filter $\text{filter}_\mu$ of Section 2.2, checks the label's sources in sequence and returns the sub-label $w'$ accepted before the first rejection. Hence $w' = w \cup \bigcup pc$ exactly when $\mu(s, w \cup \bigcup pc) = \text{allow}$. The check takes $\xi_{\text{true}}$ or $\xi_{\text{false}}$ accordingly, in both cases pushing the checked value's own contribution $w' \cap w$, as control flow now depends on the verdict.

D-IfCheck

$$\frac{w' := \text{passed}_\mu(s, w \cup \bigcup pc) \qquad c := (w' = w \cup \bigcup pc)}{\langle \texttt{if check}_s\ \langle v, w\rangle\ \{\ \xi_{\text{true}}\ \}\ \texttt{else}\ \{\ \xi_{\text{false}}\ \}, m, pc\rangle \overset{[]}{\rightsquigarrow} \langle \xi_c\ ;\ \texttt{leave}\{\xi_{\neg c}\}, m, [w' \cap w] \cdot pc\rangle}$$

The $\texttt{leave}\{\xi_{\neg c}\}$ marks the branch exit, where the pushed $pc$ entry is popped and joined, together with the remaining $pc$, into the variables the untaken branch could have assigned, capturing the implicit flow from the verdict. The remaining dynamic rules appear in the extended version [23].

The typing rule S-IfCheck encodes this introspection statically, abstracting the runtime verdict $w'$ by types and refining each branch accordingly. When the checked expression has multiple influences, the short-circuit behavior of $\text{passed}_\mu$ complicates the analysis. We consider the two possible cases:

- **Positive branch:** All labels passed. When $t$ is a variable, we refine its binding to $\tau^{+s}$. The checked value has been approved as a whole, so the decomposition direction of Assumption 1 extends acceptance to every atom of $\tau$. The construct also checks all influences in the $\overline{pc}$, so its elements receive $+s$ and finally the verdict influence $\tau^{+s}$ is pushed at the top.
- **Negative branch:** Short-circuiting means that the verdict reports only what passed before the first failure. If $\ell_1 \otimes \ell_2$ fails at $\ell_2$, the verdict is $\ell_1^{+s}$, and if it fails already at $\ell_1$, the verdict is $\mathbb{1}^{+s}$. At runtime, D-IfCheck thus pushes only the *sub-labels* of the checked value that passed. Statically, we do not know where the failure occurred, so the type $\text{sub}(\tau^{+s})$ over-approximates the pushed verdict. It admits every sub-label of $\tau$'s locations, each carrying $+s$, and its $\mathbb{1}$ disjunct covers the case where nothing passed. For environment refinement, we assign $-s$ only when $\bigotimes \overline{pc} \equiv \mathbb{1}$ and $\tau \equiv \ell^S$, i.e., the context carries no influences and the checked value is a single location. In every other case, we cannot attribute the failure to a specific label and therefore refine nothing.

*Example 4.10.* Consider $\texttt{if x \{ if check}_s\ \texttt{y \{ }\bowtie_s\ \texttt{y \} else \{ skip \} \} else \{ skip \}}$ where $x : \ell_x$ and $y : \ell_y$. The outer `if` branches under the influence of $\ell_x$, setting $\overline{pc}_1 := [\ell_x]$ for its body. In the `if check`, the positive branch executes when $y$ is safe for scope $s$, so we refine $y$'s type to $\ell_y^{+s}$. Since the branching decision is influenced by both $\overline{pc}_1$ and `y`, the $\bowtie$ is analyzed with $\overline{pc}_2 := [\ell_y^{+s}] \cdot \overline{pc}_1^{+s} = [\ell_y^{+s}, \ell_x^{+s}]$. Recall that safe introspection requires the verdict to have positive permission for scope $s$ in all cases.

In the negative branch, the check fails whenever *any* influence is disallowed. It is thus not statically known whether the failure is caused by influences in $\overline{pc}_1$, in `y`, or both, and thus refining to $\ell_y^{-s}$ is unsound. Formally, $\bigotimes \overline{pc}_1 = \ell_x \not\equiv \mathbb{1}$, so S-IfCheck adds no negative permission. Since it is also not known whether the verdict is influenced by $\ell_y$, inside the negative branch $x : \ell_x$ remains without permissions and $\overline{pc}_3 := [\text{sub}(\ell_y^{+s})] \cdot \overline{pc}_1 = [\ell_y^{+s} \oplus \mathbb{1}, \ell_x]$.

*Restrictions on Assignments.* As a simplifying assumption, we require that expressions appearing in assignments yield non-function values, i.e., they are not function abstractions at their outermost level. This restriction aligns with existing imperative IFC platforms, which do not allow function definitions to appear in locations influenced by control flow. We also require that assignment targets inside a lambda body are bound within that lambda, i.e., they are its parameters or local variables. Hence functions affect their callers only through their return value.

*Example 4.11.* The following types are inferred in the last statement in Example 4.1:

$$\texttt{db.read\_posts} : \mathbb{1} \to \ell_{\text{db}} \qquad \texttt{posts} : \ell_{\text{db}} \qquad \texttt{selected\_posts} : \ell_{\text{db}}^{\{+\langle \texttt{display},\texttt{current\_user}\rangle\}} \oplus \mathbb{1}$$

### 4.4 Properties

Following standard Hindley-Milner practice [26, 27], polytypes serve purely as an inference mechanism, occurring only in the environment $\Gamma$ and instantiated to fresh ground monotypes at each use site. The concretization $\gamma$ and all the properties below are therefore stated over ground monotypes.

We state the key properties of our type system below. The expression sublanguage satisfies analogous properties (detailed in the extended version [23]) that serve as lemmas in the statement-level proofs.

*Definition 4.12 (Well-Typed Memory).* A memory $m$ is *well-typed* under $\Gamma$, written $\models m : \Gamma$, if $m$ contains exactly the variables with ground monotypes in $\Gamma$ (i.e., $\mathrm{dom}(m) = \{x \in \mathrm{dom}(\Gamma) \mid \Gamma(x) \in \mathrm{GMono}\}$), and for all $x \in \mathrm{dom}(m)$ we have $m(x) \in \gamma(\Gamma(x))$.

*Definition 4.13 (PC Compatibility).* A dynamic program counter $pc = [w_1, \ldots, w_n]$ is *compatible* with static program counter $\overline{pc} = [\tau_1, \ldots, \tau_n]$, written $pc \sim \overline{pc}$, if they have equal length and each dynamic label satisfies its corresponding static type: $|pc| = |\overline{pc}| \wedge \forall i.\ \exists v.\ \langle v, pc[i]\rangle \in \gamma(\overline{pc}[i])$

Theorem 4.14 (Type Soundness). *If $\Gamma; \overline{pc} \vdash \xi \dashv \Gamma'$, $\models m : \Gamma$, and $pc \sim \overline{pc}$, then:*

- **(Progress)** *Either $\xi = \texttt{skip}$, or there exists $\xi'$, $m'$, $pc'$, $\pi$ such that $\langle \xi, m, pc\rangle \overset{\pi}{\rightsquigarrow} \langle \xi', m', pc'\rangle$, or $\langle \xi, m, pc\rangle \overset{\pi}{\rightsquigarrow} \lightning$. Notably, well-typed programs never get stuck.*
- **(Preservation)** *If $\langle \xi, m, pc\rangle \overset{\pi}{\rightsquigarrow} \langle \xi', m', pc'\rangle$, then there exist $\Gamma''$, $\overline{pc}'$ such that $\Gamma''; \overline{pc}' \vdash \xi' \dashv \Gamma'$, $\models m' : \Gamma''$, and $pc' \sim \overline{pc}'$.*

*Consequently, if $\langle \xi, m, pc\rangle \rightsquigarrow^* \langle \texttt{skip}, m', pc'\rangle$, then $\models m' : \Gamma'$.*

Proof (Summary). Progress follows by case analysis on $\xi$, and preservation by induction on the typing derivation. The key insight is that the flow-sensitive environment $\Gamma'$ correctly predicts the final memory state across all execution paths. For the full proof, see the extended version [23]. □

Theorem 4.15 (Decidability and Principal Types). *The typing rules in Figs. 3 and 6 are syntax-directed and thus directly yield a type inference algorithm. Given $\Gamma$, $\overline{pc}$, and $\xi$, this algorithm either computes the output environment $\Gamma'$, unique up to normal-form equality, with $\Gamma; \overline{pc} \vdash \xi \dashv \Gamma'$, or reports that $\xi$ is not typeable.*

Proof (Summary). The typing rules have no overlapping conclusions and require no search. Structural recursion on $\xi$ yields a terminating algorithm (termination of loop fixed points follows from finite normal forms) whose output is deterministic by construction. See the extended version [23]. □

*Security Guarantees.* We now derive security-specific guarantees that formalize how our type system predicts runtime label behavior and enforcement outcomes. They serve to preserve the underlying system's properties rather than to establish a new security property.

Corollary 4.16 (Tracking Safety). *If* $\Gamma; \overline{pc} \vdash \xi \dashv \Gamma'$, $\models m : \Gamma$, *and* $pc \sim \overline{pc}$, *then execution never gets stuck from tracking mismatches.*

Proof. Immediate from Progress (Theorem 4.14). The type system's tracked/untracked discipline prevents untracked values from reaching label sinks. □

Lemma 4.17 (Influence Soundness). *If* $\Gamma; \overline{pc} \vdash t : \tau$, $\models m : \Gamma$, $pc \sim \overline{pc}$, *and* $\langle t, m, pc\rangle \Downarrow \langle v, w\rangle$, *then* $\langle v, w\rangle \in \gamma(\tau)$. *In particular, if the location* $\ell$ *does not appear in* $\tau$, *then we have* $\ell \notin \mathrm{loc}(w)$.

Proof (Summary). Follows from Type Soundness (Theorem 4.14) and the definition of $\gamma$. The last clause follows from $\gamma(\tau)$ not containing values labeled from $\ell$ when $\ell \notin \tau$ (see the extended version [23]). □

The negative case (second clause) is particularly important for optimization, as it guarantees that omitted locations definitively do not influence the value.

Assumption 1 (Source-wise enforcement). *We consider enforcers that, for every scope* $s$, *accept a label exactly when they accept each of its sources individually, i.e.,* $\mu(s, w_1) = \mathsf{allow} \wedge \mu(s, w_2) = \mathsf{allow} \iff \mu(s, w_1 \cup w_2) = \mathsf{allow}$, *and* $\mu(s, \emptyset) = \mathsf{allow}$.

This holds structurally for WebTTC, whose enforcer checks each input source independently against the applicable policy. It also corresponds to the standard notion of transitive policies in multi-level security [28].

Corollary 4.18 (Permission Safety). *If* $\Gamma; \overline{pc} \vdash t : \tau$, $\bigotimes \overline{pc} \otimes \tau \equiv \rho^S$ *with* $+s \in S$ *and* $-s \notin S$, *and* $\langle t, m, pc\rangle \Downarrow \langle v, w\rangle$ *under well-typed conditions, then* $\bowtie_s \langle v, w \cup pc\rangle$ *succeeds (i.e.,* $\mu(s, w \cup pc) = \mathsf{allow}$*).*

Proof. Follows from Definition 4.4 and Lemma 4.17. □

This property is the foundation for safe guard elimination: a positive permission $+s$ statically guarantees that the runtime guard will succeed, which we leverage in Section 5 to construct behavior-preserving program transformations.

We remark, without formalizing it, that negative permissions dually predict failure: if $\Gamma; \overline{pc} \vdash t : \tau^S$ with $-s \in S$ and $+s \notin S$, every guard $\bowtie_s t$ fails. Our optimization does not rely on this direction.

*Relation to Non-interference.* MinIF preserves the guarantees of the underlying enforcer, as the optimized program produces the same enforcement decisions as the original (Corollary 5.5). When instantiated with a dynamic IFC system that performs sound taint tracking, the preserved guarantee is the sound approximation of non-interference [16] of Section 2, which the optimized program inherits unchanged.

Permission Safety and the warning mechanism of Section 5.1 together yield a practical crash-freedom guarantee: if MinIF accepts a program and emits no diagnostic warnings (i.e., every guard carries an allow prediction), then no guard can fail at runtime, and the program is guaranteed not to crash or terminate due to enforcement rejection.

## 5 Optimizing Label Tracking

This section presents our program transformation procedure, which leverages IFIR's types to remove unnecessary instrumentation from well-typed IFIR programs while preserving semantics. We target two forms of redundancy: guards $\bowtie_s$ predicted to always succeed, and label tracking

outside source-to-sink paths. Two sequential phases remove them: *unguarding* exploits permission safety (Corollary 4.18) to eliminate the redundant guards, and *untracking* exploits tracking safety (Corollary 4.16) to convert the constructs serving them to more efficient untracked variants.

### 5.1 Unguarding

Recall that guards use the enforcer to perform security checks, requiring a tracked value. When type analysis statically predicts the enforcer's decision, the enforcer call becomes redundant. For a guard $\bowtie_s t$, we predict the enforcer's decision using a three-valued logic: the call will always be accepted by the enforcer (allow), the call will always be rejected (deny), or there is not enough information to conclude either case (maybe). These predictions form a partial order with maybe below the incomparable allow and deny. We write $\curlyvee$ for the greatest lower bound in this order, which captures the worst (i.e., most conservative) case.

We formalize enforcer prediction through *prediction rules* (see the extended version [23]), whose judgments have the form $\Gamma; \overline{pc} \Vdash t : p$, with $p \in \{\text{allow}, \text{deny}, \text{maybe}\}$. The distinct turnstile $\Vdash$ distinguishes them from the typing judgments, which they otherwise mirror. Non-compound constructs with no enforcement role are always predicted allow, since they do not trigger enforcement checks. Compound constructs take the $\curlyvee$ of their parts. The main rule driving optimization is P-Guard, which lifts type information into a prediction:

$$\text{P-Guard}\quad \frac{\Gamma; \overline{pc} \vdash t : \tau \qquad \Gamma; \overline{pc} \Vdash t : p}{\Gamma; \overline{pc} \Vdash \bowtie_s t \;:\; p \curlyvee \text{predict}\Big(\bigotimes \overline{pc} \otimes \tau,\; s\Big)},$$

where $\text{predict}(\tau, s)$ distinguishes three cases. It is allow if $\tau \equiv \rho^S$ for some $\rho$ and $S$ with $+s \in S$ and $-s \notin S$, and it is deny if instead $-s \in S$ and $+s \notin S$. Otherwise, it is maybe. The condition for allow is exactly the hypothesis of Permission Safety (Corollary 4.18). In disjunctive normal form, where $\text{nf}(\tau) \equiv \bigoplus_i \rho_i^{S_i}$, allow (resp. deny) means that $+s$ (resp., $-s$) is present in every disjunct. On uninhabited types, both factorizations hold ($\mathbb{0} \equiv \mathbb{0}^{\{+s\}} \equiv \mathbb{0}^{\{-s\}}$) and we give allow priority. This is harmless since no value ever reaches such a guard.

The transformation is straightforward: when $\bowtie_s \; t$ is predicted allow, we replace it with $t$, eliminating the redundant enforcer check while preserving semantics. MinIF rejects programs where some guard is predicted deny, since the enforcer would always refuse the corresponding action at runtime. Guards predicted maybe are left in place, but MinIF emits a compile-time warning for each such guard, guiding developers to add introspection checks, declassification, or similar mechanisms that refine the type environment until all guards are resolved to allow. This avoids enforcement-induced termination.

### 5.2 Untracking

Tracking safety (Corollary 4.16) guarantees that well-typed programs have no tracking mismatches, but says nothing about efficiency. Many programs track labels on values that never reach security-critical operations, e.g., intermediate variables used only for non-sensitive outputs. Since label tracking imposes runtime overhead, we aim to identify and eliminate unnecessary tracking while preserving security guarantees.

The key insight is that tracking is only necessary along *source-to-sink paths*: if a value never flows (directly or indirectly) to a label sink, its tracking can be safely removed. We formalize this intuition using *program slicing* [29], which computes the parts of a program contributing to values at given locations.

*Program Dependence Graphs.* Program slicing operates on a *program dependence graph* (PDG), a tuple $\langle \mathcal{N}, \xrightarrow{\text{ctrl}}, \xrightarrow{\text{data}} \rangle$ where nodes $\mathcal{N}$ are the nodes of the program's abstract syntax tree (AST), both expressions and statements, and edges represent dependencies between them. A node $n'$ has a *control dependence* $n \xrightarrow{\text{ctrl}} n'$ when execution of $n'$ depends on a branching decision at $n$ (e.g., $n$ is an `if` condition and $n'$ is a statement in its body). A *data dependence* $n \xrightarrow{\text{data}} n'$ exists when $n'$ reads a value produced by $n$ (e.g., $n$ assigns to a variable that $n'$ reads, or $n$ is a subexpression whose value flows to $n'$).

*Computing the Slice.* Our procedure identifies which AST nodes require label tracking by computing a backward slice from all label sinks:

(1) Identify *unoptimizable sinks* $\mathcal{N}_0^*$, i.e., AST nodes where label tracking is mandatory:
   - All `if check` statements (which perform introspection on labels)
   - All guard expressions $\bowtie_s t$ not predicted allow (i.e., guards that might fail)

(2) Compute the backward closure: $\mathcal{N}_{i+1}^* := \mathcal{N}_i^* \cup \{n' \mid \exists n \in \mathcal{N}_i^*.\ n' \rightarrow n\}$, iterating until reaching a fixed point $\mathcal{N}^*$.

*Applying the Transformation.* Nodes outside the slice, $\mathcal{N} \setminus \mathcal{N}^*$, do not contribute to any security check. They are transformed to their untracked variants, for statements as well as expressions (e.g., `x := e` becomes `x :=! e` and `f(x)` becomes `f!(x)`). When an expression crosses the boundary between tracked and untracked, we insert explicit conversions using `ᵾ`.

*Example 5.1.* Fig. 7 depicts a simplified PDG of the program in Example 4.1, showing only statement-level nodes for clarity. Each node represents a program statement labeled by its line number, with the *Entry* node representing the program's start point. Red solid edges denote control dependence, while blue dashed edges represent data dependence with variable names as edge labels. In practice, our PDG also includes expression-level nodes, enabling fine-grained analysis of dataflow through subexpressions.

The minimal set of unoptimizable sink nodes $\mathcal{N}_0^* = \{6\}$, highlighted in orange, contains only the **`if check`** statement, which must relay labels from variable `post` to the enforcer for query execution. The guard at line 11 is not an unoptimizable sink, as it is predicted allow. Computing the backward slice from node 6 until a fixed point yields $\mathcal{N}^* = \{\text{Entry}, 1, 3, 4, 5, 6, 9\}$, shown in green. These nodes (and their constituent expressions) must propagate labels to ensure the **`if check`** at line 6 receives label information along all execution paths. The remaining statements $\{2, 7, 11\}$ do not contribute to any security check and are selected for optimization.

Thus, the optimized program does not track labels for `selected_posts` and does not have a guard for the call to `display`:

```
1  posts := db.read_posts() //equivalent to posts := [◇ "Post 1", ◇ "Post 2", ...]
2  selected_posts :=! ᵾ([])
3  i := 0
4  while i < len(posts) and i < 30 {
5      post := posts[i]
6      if check⟨display,current_user⟩ post {
7          selected_posts :=! selected_posts + ᵾ([post])
8      }
9      i := i + 1
10 }
11 display!(selected_posts)
```

*Example 5.2.* IFIR code for the program from Example 3.3 is `display(⋈ sanitize(db.read()))`.

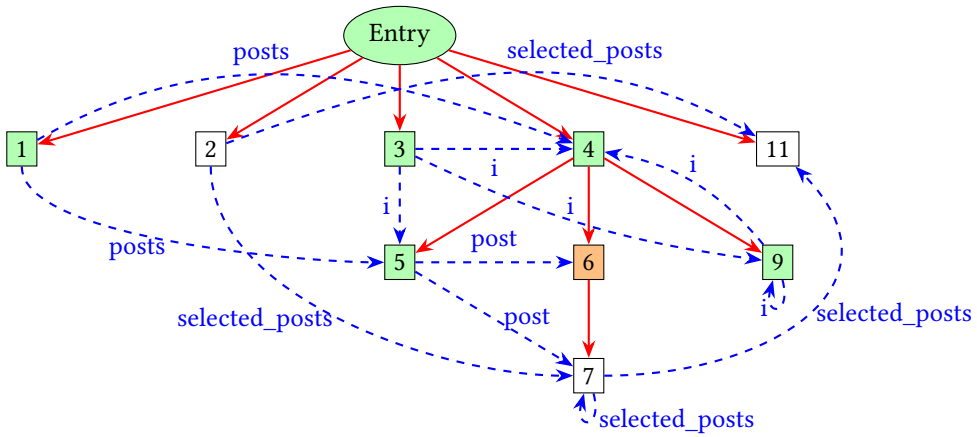


Fig. 7. PDG of the program in Example 4.1

Given the typed terms `db.read` : $\mathbb{1} \to \ell$ and `sanitize` : $\forall\alpha.\ \alpha \to \mathbb{1}$, type inference predicts that the guard will always be accepted, yielding `display!(sanitize!(υ(db.read())))`. Since `sanitize` is label-insensitive, its untracked variant computes the same result.

*Hoisting Boundary Conversions.* Inserting a $\upsilon$ at each tracked-to-untracked boundary can leave a conversion inside a loop, where a runtime that assigns $\upsilon$ a non-zero cost pays it on every iteration. Because $\upsilon(x)$ is a total, side-effect-free function of $x$'s value, evaluating it earlier or more often never changes the result. It may therefore be hoisted out of any loop in which $x$ is invariant and, more generally, to any point that dominates the conversion with no reassignment of $x$ in between. Slice-based untracking thus *removes* tracking from values outside source-to-sink paths, while hoisting *amortizes* the boundary conversions that remain. Only conversions of a variable may be hoisted. A $\upsilon$ whose operand may trigger enforcement termination or emit an event (e.g., a $\bowtie_s$) is never moved, since relocating those effects could alter the program's observable behavior.

### 5.3 End-to-End Correctness

We now describe the complete transpiler pipeline and establish its correctness. Given an IFIR program $\xi$, type inference computes typing judgments for all statements, prediction analysis determines guard outcomes from the inferred types, and the transformation applies unguarding and untracking. If type inference fails or any statement is predicted deny, the program is statically rejected. Otherwise, the transformation phases produce the optimized program $\xi'$.

The optimized program $\xi'$ has an identical control-flow structure to $\xi$. The transformation only affects whether constructs use tracked or untracked variants, and whether guards are present; so both programs take the same execution branches and produce memories that agree on all underlying values, differing only in the presence or absence of labels and enforcement checks.

Our optimization selectively removes enforcement checks, moving traces away from perfect synchronization (reducing overhead) while maintaining safety. Actual and observed traces may diverge when checks are elided, which is safe when the enforcer would accept the unlogged events.

More precisely, we compare the events produced by $\xi$ and $\xi'$ by their security scope alone, since labels may differ after optimization. Two events $\langle s, w\rangle$ and $\langle s', w'\rangle$ are *behaviorally equivalent*, written $\langle s, w\rangle \sim \langle s', w'\rangle$, iff $s = s'$. Intuitively, they are the same action from an external observer's point of view. We extend this to traces of equal length: $\pi_1 \sim \pi_2$ iff $|\pi_1| = |\pi_2|$ and $\forall i.\ \pi_1[i] \sim \pi_2[i]$.

Eliding a guard is safe only when the enforcer would have accepted the corresponding event anyway, which we formalize as a property of the original trace.

*Definition 5.3 (Sufficient Logging).* A well-formed marked trace $\pi$ is *sufficiently logged* with respect to enforcer $\mu$ iff: $\forall i.\ \left(\nexists j.\ \pi|_{\uparrow}[j] = \pi|_{\downarrow}[i]\right) \implies \mu(\pi|_{\downarrow}[i]) = \text{allow}$

Theorem 5.4 (Optimization Soundness). *Let $\xi'$ be the program obtained from optimizing the well-typed program $\xi$ for enforcer $\mu$. If $\xi$ produces the well-formed trace $\pi$ from initial memory $m$, then there exists a well-formed trace $\pi'$ generated by $\xi'$ from $m$ such that:*

**(Behavioral Preservation)** $\pi'|_{\downarrow} \sim \pi|_{\downarrow}$*;*
**(Overhead Reduction)** $\pi'|_{\uparrow} \subseteq \pi|_{\uparrow}$*; and*
**(Safe Under-logging)** *if $\pi$ is sufficiently logged, so is $\pi'$.*

Proof (Summary). Behavioral Preservation and Overhead Reduction are immediate from the transformation's structural properties. For Safe Under-logging, unlogged events in $\pi'$ are either (1) already unlogged in $\pi$ (accepted by assumption) or (2) newly unlogged by removing guards predicted allow. See the extended version [23] for the complete proof. □

Thus, the optimized program exhibits the same behavior as the original under enforcement. Ligatti et al. [30] define an enforcer as *sound* if it only accepts executions satisfying a property $\phi$, and *transparent* if it does not suppress executions that already satisfy $\phi$. Together, they characterize a well-behaved enforcer that neither over- nor under-enforces.

Corollary 5.5 (Soundness and Transparency Preservation). *Let $\phi$ be a property of actual traces ($\pi|_{\downarrow}$) that depends only on the scopes of their events.*[2] *If enforcer $\mu$ is sound (resp. transparent) [30] for $\phi$ on program $\xi$, then $\mu$ is sound (resp. transparent) for $\phi$ on the optimized program $\xi'$.*

Proof. By Theorem 5.4, every enforced run of $\xi'$ corresponds to an enforced run of $\xi$ with the same scope sequence ($\pi'|_{\downarrow} \sim \pi|_{\downarrow}$). A run of $\xi'$ that $\mu$ accepts thus shares its scope sequence with an accepted run of $\xi$, which satisfies $\phi$ by soundness on $\xi$. Dually, a run of $\xi'$ that satisfies $\phi$ corresponds to a $\phi$-satisfying run of $\xi$, which $\mu$ does not suppress on $\xi$. By Safe Under-logging, the elided checks would all have been accepted, so no new suppression arises on $\xi'$. □

In particular, consider the sound approximation of non-interference discussed in Section 2.3. Since every removed check is provably accepted, the optimized and original programs produce scope-equal permitted effects and are cut at identical points under the same enforcer. The optimized program thus admits the same enforced behaviors, including approximated non-interference.

## 6 Implementation and Evaluation

Our MinIF transpiler is publicly available[3] and consists of approximately 6,000 lines of Scala code, structured as a sequence of passes: an ANTLR4-generated parser produces an IFIR AST, followed by symbol resolution, information-flow type inference, PDG construction, unguarding, and untracking. The implementation adopts all optional refinements described in this paper and extends the formal model in two ways to support realistic programs. First, *external function declarations* assign IFIR type signatures to library functions and language primitives defined outside the program, enabling type inference to reason about their information-flow behavior without their source code. IFC-related APIs (e.g., declassifiers, label-raising operations) are declared this way, and their signatures

[2]Such properties constrain which actions occur, on which data, and in which order. They abstract only the label metadata, which the optimization changes by design, and the enforcer still consults labels at every remaining check.
[3]Source code and benchmarks are available online: https://github.com/dgalan7/minif

Table 1. Evaluation results

| Scenario | | | Execution (ms) | | | Overhead (ms) | | |
|---|---|---|---|---|---|---|---|---|
| Program (LoC) | Compilation (ms) | $n$ | $T_{\text{BASE}}$ | $T_{\text{ENF}}$ | $T_{\text{OPT}}$ | $OH_{\text{ENF}}$ | $OH_{\text{OPT}}$ | $\Delta OH$ |
| `matrix` (16) | | 10 | 7.5 | 150.6 | 24.9 | 143.1 | 17.3 | **87.9%** * |
| · $\ell = 3, \lvert\mathbb{S}\rvert = 1$, ops = 3 | 58.2 | 50 | 47.2 | 4,627.4 | 136.6 | 4,580.2 | 89.4 | **98.0%** * |
| · PDG (#n/#e) = 154/655 | | 100 | 213.8 | 45,021.9 | 502.8 | 44,808.0 | 289.0 | **99.4%** * |
| `filter` (11) | | 500 | 13.9 | 6,036.7 | 4,650.7 | 6,022.8 | 4,636.8 | **23.0%** * |
| · $\ell = 2, \lvert\mathbb{S}\rvert = 1$, ops = 2 | 29.2 | 1000 | 68.7 | 30,529.1 | 20,959.8 | 30,460.4 | 20,891.1 | **31.4%** * |
| · PDG (#n/#e) = 76/129 | | 2000 | 59.8 | 41,612.8 | 29,381.1 | 41,553.8 | 29,322.1 | **29.4%** * |
| `simple` (6) | | n/a | 6.0 | 33.9 | 29.7 | 27.9 | 23.7 | **15.2%** * |
| · $\ell = 2, \lvert\mathbb{S}\rvert = 1$, ops = 2 | 12.1 | | | | | | | |
| · PDG (#n/#e) = 16/19 | | | | | | | | |
| `sort` (21) | | 10 | 7.6 | 169.5 | 57.5 | 161.8 | 49.9 | **69.2%** * |
| · $\ell = 2, \lvert\mathbb{S}\rvert = 1$, ops = 2 | 17.7 | 100 | 14.1 | *> 60,000* | 832.9 | | 818.9 | |
| · PDG (#n/#e) = 112/377 | | 1000 | 20.3 | *> 60,000* | 12,515.5 | | 12,312.6 | |
| `fibonacci` (14) | | 100 | 6.2 | 49.1 | 29.0 | 42.9 | 22.9 | **46.7%** * |
| · $\ell = 2, \lvert\mathbb{S}\rvert = 1$, ops = 2 | 7.6 | 1000 | 6.4 | 126.3 | 32.0 | 119.9 | 25.7 | **78.6%** * |
| · PDG (#n/#e) = 44/93 | | 10000 | 9.7 | 613.4 | 54.0 | 603.7 | 44.3 | **92.7%** * |
| `MiniTwitter` (172) | | 10 | 2.3 | 19.8 | 15.8 | 17.5 | 13.5 | **22.8%** * |
| · $\ell = 26, \lvert\mathbb{S}\rvert = 1$, ops = 30 | 73.6 | 50 | 2.3 | 68.9 | 33.9 | 66.6 | 31.7 | **55.4%** * |
| · PDG (#n/#e) = 658/859 | | 100 | 2.4 | 70.8 | 61.8 | 68.4 | 59.4 | **13.1%** * |
| `DataFit` (363) | | 10 | 4.1 | 231.9 | 136.5 | 227.8 | 132.3 | **41.9%** * |
| · $\ell = 19, \lvert\mathbb{S}\rvert = 3$, ops = 25 | 274.6 | 100 | 12.6 | 3,453.7 | 1,229.0 | 3,441.1 | 1,216.4 | **64.7%** * |
| · PDG (#n/#e) = 1643/3497 | | 1000 | 108.6 | *> 60,000* | 43,370.1 | | 43,261.5 | |

directly encode the permission information that drives guard elimination, matching the optimization potential of introspection queries even in systems that lack them.

Second, *analysis is inter-procedural*. Type inference is performed per function and type signatures are propagated across call boundaries. Then, backward slicing and optimization also run per function, analyzing flows between arguments and return positions alongside local sources and guards in the function's body. These extensions are conservative and preserve the core theory's soundness.

Our language- and enforcer-agnostic design (Fig. 1) is validated by a Python adapter for the WebTTC platform that extends the Flask web framework with dynamic IFC [7]. The adapter takes a Python source file and an `.ifsig` file encoding *adapter meta-information* (WebTTC function signatures) and produces an optimized Python file. A pre-translation phase converts WebTTC-instrumented Python ASTs to IFIR and a post-translation phase reconstructs executable Python from the optimized output. In total, the adapter comprises roughly 1,900 handwritten lines of Python and 4,300 lines generated by ANTLR from the shared IFIR grammar.

To assess the runtime performance gains on real-world programs, we measure the latencies of Python functions executed by the WebTTC platform [7] under three configurations: BASE (no IFC instrumentation), ENF (full dynamic IFC by WebTTC), and OPT (MINIF-optimized enforcement). We define the enforcement overhead as $OH_\bullet := T_\bullet - T_{\text{BASE}}$, with $\bullet \in \{\text{ENF}, \text{OPT}\}$, and the speedup as $\Delta OH := (OH_{\text{ENF}} - OH_{\text{OPT}})/OH_{\text{ENF}}$, where $\Delta OH = 100\%$ means the optimized program runs at baseline speed. The policy enforced by WebTTC's enforcer is set to $\top$ (i.e., all outputs are allowed). This is the least favorable case for optimization, as any non-trivial policy would induce higher enforcer latency and thereby allow for larger speedups from eliminating redundant enforcer calls. The speedups we measure thus provide a conservative lower bound.

*Empirical Setup.* We consider five microbenchmarks and two real Flask web applications with inputs of varying sizes $n$. The microbenchmarks are: `matrix` (multiply two $n \times n$ matrices), `filter` (the paper's running example: filter $n$ database posts for display), `simple` (an `if check` followed by a matching guarded call), `sort` (bubble sort on $n$ elements), and `fibonacci` (compute the $n$-th Fibonacci number). The Flask applications are `MiniTwitter`, a Twitter-like social media platform [7], and `DataFit`, a data sharing and prediction application. For `MiniTwitter`, we exercise the `timeline` function, which filters and displays the latest $n$ posts; on `DataFit`, we exercise the `predict` function, which fits a linear model on a dataset of $n$ records and computes a prediction on user-provided inputs. Typical IFC policies would forbid timelines computed from the data of users who blocked the reader (`MiniTwitter`) and model training on personal data without consent (`DataFit`). Each workload is executed 10 times with a 60-second timeout on a 14-core Intel Core Ultra 7 5.3 GHz machine with 32 GB RAM running Ubuntu 24.04, and the runtimes are averaged.

*Results.* Table 1 reports our measurements. Rows marked $*$ denote statistically significant speedups (Wilcoxon signed-rank test at significance level $\alpha = 0.05$). All speedups are statistically significant. Timeouts are indicated as *> 60,000* and their corresponding overhead is omitted. Speedups range from 13.1% (`MiniTwitter`, $n = 100$) to 99.4% (`matrix`, $n = 100$), depending on how much redundant tracking and enforcement MinIF removes.

The most dramatic gains occur in compute-intensive loops: for `matrix` at $n = 100$, MinIF reduces enforcement overhead from 44 s to 289 ms; for $n \geq 100$, `sort` times out under enforcement while OPT completes in 833 ms at $n = 100$ and 12.5 s at $n = 1000$. Our two Flask applications show speedups between 13% (`MiniTwitter`, $n = 100$) and 65% (`DataFit`, $n = 100$). In general, speedup grows with $n$ for computation-intensive programs whose gains stem from removing tracking (`matrix`, `sort`, `fibonacci`, `DataFit`), and is constant or slightly decreasing for large $n$ in programs whose gains stem from removing enforcer calls only (`filter`, `MiniTwitter`).

Accordingly, `DataFit` filters data before performing compute-intensive linear algebra, so MinIF operates on untainted data, and at $n = 1000$ the optimized program completes where the enforced one times out. `MiniTwitter` instead peaks at $n = 50$, since for larger $n$ the cost of checking $n$ individual posts (as in `filter`) dominates the removed final enforcer call, so cautious pagination remains necessary to avoid high latencies.

*Compile-Time Overhead.* Optimization is a one-time compiler cost, while the removed instrumentation saves time on every execution. Table 1 annotates each benchmark with the quantities that drive the analysis: its source locations $\ell$, security scopes $|\mathbb{S}|$, label-relevant operations ops, and PDG size (#n nodes, #e edges), together with the resulting compile time. Compilation averages 67.6 ms per benchmark on a warm JVM, measured as the median of 25 runs of the full pipeline on in-memory IFIR, excluding JVM startup, file I/O, and the adapter's Python translation. End to end (through the adapter's Python translation), optimizing the largest case study (DataFit) takes 4.2 s, of which the analysis accounts for under 0.3 s and the rest is JVM startup, file I/O and adapter overhead. Analysis cost is governed by the size of type normal forms. A variable assigned in either branch of a conditional receives a union type, so an expression combining $n$ such variables receives an intersection of $n$ unions, whose normal form contains one disjunct for every way of choosing a branch in each union, up to $2^n$. These disjuncts cannot be merged without losing precision (Section 5.1). In practice, programs do not nest unions adversarially, and normal forms stay small, with at most 16 disjuncts across our benchmarks.

## 7 Related Work

IFC systems face a fundamental tradeoff between static and dynamic analysis. Pure static approaches are efficient but conservatively reject safe programs, while pure dynamic approaches are more

precise but incur substantial overhead. MinIF sits at a distinct point in this design space: rather than replacing runtime enforcement, we optimize existing dynamic systems by proving when instrumentation can be safely eliminated.

Jif [31] reasons statically about information flows in Java, FlowCaml [32] achieves full type inference for OCaml, and Cocoon [33] brings static IFC to Rust. Being fully static, these systems relax their conservative approximation only through coarse, manual mechanisms such as declassification, rather than by consulting the runtime state as an enforcer would. MinIF uses static analysis to optimize existing dynamic systems rather than replacing them.

Jif nonetheless admits a dynamic element. Labels and principals are first-class values, and programs may branch on the runtime tests `l1 <= l2` and `p actsfor q`, whose outcomes refine static checking within each branch. Such a test shares its shape with `if check`, but evaluates a fixed ordering known at compile time against dynamic label values, so Jif's static *pc* mechanism already accounts for its outcome. An `if check` instead consults the enforcer, whose policy is unknown at compile time, and its verdict reveals how that policy regards the checked value's sources, so the verdict itself is sensitive (Section 4.3). Moreover, predicting a test in Jif would be unsound, as the acts-for hierarchy may change at runtime, while MinIF can predict verdicts and remove the checks predicted to succeed, as the enforcer is fixed within an execution. Jif thus certifies a program against the policy declared in its labels, while MinIF's guarantees hold for every enforcer satisfying Assumption 1.

Dynamic IFC systems with label tracking [34, 35] incur substantial runtime overhead. Binary instrumentation systems like libdft [36] and VM-level tracking in TaintDroid [37] achieve low double-digit overhead through implementation-level optimizations. Hardware assistance [38, 39] reduces overhead further but requires custom silicon. MinIF complements these efforts by proving when tracking and enforcement can be eliminated entirely through compile-time analysis.

Gradual security typing [40, 41, 42] uses unknown types ($\star$) to enable incremental program annotation. Chen & Siek [43] resolve the tension between non-interference and the dynamic gradual guarantee by restricting $\star$ to compile-time annotations directing runtime label discovery. MinIF addresses a complementary problem: automatically optimizing existing dynamic systems. Our permission annotations track enforcement outcomes rather than security label precision, orthogonal to gradual typing's focus on enabling gradual migration between static and dynamic checking. Furthermore, MinIF requires no programmer intervention to choose the static-dynamic boundary as type inference determines optimal elimination automatically.

Beyond security, static types have long served to reduce the cast overhead of gradual typing. Rastogi et al. [44] infer types to remove implicit coercions in ActionScript, and Campora et al. [45] optimistically infer function types to generate cast-free specializations backed by unoptimized fallbacks. MinIF shares the spirit of removing dynamic checks that types prove redundant, but neither system tracks information flow, and the removed casts guard type safety within the program rather than the decisions of an external enforcer, which our transformation must preserve.

Russo & Sabelfeld [4] use static analysis as a runtime subroutine for "look aside" analysis of untaken branches. Moore & Chong [46] introduced a "look-ahead" static analysis that stops tracking a variable once its content can no longer flow into a confidential output. Their work is an early blueprint for the hybrid approach we advocate, in a more restricted setup without introspection or function definitions, and with a fixed lattice of security levels. With a known lattice, check outcomes are computed from labels alone, so their type system needs no permissions and a traditional flow-sensitive analysis suffices. Both works [4, 46] remain primarily dynamic systems that optimize the monitor during execution, with only Moore & Chong briefly discussing compile-time optimization. MinIF instead operates entirely at compile time, and its slicing eliminates instrumentation wholesale for functions outside source-to-sink paths rather than selectively deallocating label metadata at runtime.

The Haskell IFC ecosystem spans purely dynamic (LIO [8]) to purely static (MAC [47]). HLIO [22] occupies the middle: programmers use `defer` annotations to explicitly shift constraints from compile-time to runtime checking. MinIF automates this decision through type inference, requiring annotations only for external sinks. HLIO provides explicit programmer control, while MinIF provides automatic optimization with formal guarantees.

Iodine [48] uses profile-guided optimization with fallback paths when invariants are violated. MinIF provides conservative, guaranteed elimination through formal proofs, without profiling or fallbacks. A key distinction is our use of introspection. To the best of our knowledge, we are the first to use introspection queries to predict enforcement check outcomes and enable guard elimination. This is orthogonal to profile-based optimization and the two could be combined.

SelectiveTaint [49] and HardTaint [50] use static analysis to reduce instrumentation overhead at the binary level, but optimize only label propagation. MinIF also optimizes enforcement checks.

Carapace [51] extends Cocoon's static Rust IFC with dynamic checks, using ownership types for no-overhead static checking where possible. MinIF is language-agnostic and targets existing dynamic IFC platforms and its automatic type inference needs no analogue of the explicit Rust annotations that Carapace requires to delineate static-dynamic boundaries.

Introspection mechanisms appear in several dynamic IFC systems [7, 8, 9, 10]. WebTTC [7], for instance, provides an `if check` construct closely resembling the one presented in this work. LIO [8] instead exposes introspection through `labelOf` and `canFlowTo`, pure functions on label metadata, so querying reduces to a pure comparison of static label structure. Our abstract model (Section 2.2) imposes a common structure that these systems can instantiate uniformly: query the enforcer, branch on the result, and track the implicit flow. All satisfy the well-formedness condition that the verdict is always permitted, for distinct structural reasons that mirror this distinction.

## 8 Conclusions

We have presented MinIF, a transpiler that reduces the runtime overhead of dynamic IFC through program transformation, without sacrificing security or precision. The central insight is that a flow-sensitive type system jointly tracking information-flow influences and enforcer permissions can predict enforcement outcomes statically, enabling the elimination of redundant runtime checks and the label-tracking code that serves only those checks. The optimized program provably produces the same observable effects and enforcement decisions as the original, and a warning-free program is statically guaranteed never to terminate due to enforcement rejection. On compute-intensive benchmarks, MinIF almost entirely eliminates enforcement overhead (up to 99%), turning executions that time out under enforcement into millisecond runs. For IFC systems with introspection the optimization is fully automatic, and optional annotations extend its reach to non-introspective systems through a language- and enforcer-agnostic pipeline.

A broader lesson is that programmers use introspection, sanitization, and declassification to avoid policy violations at runtime, encoding the information our approach needs to eliminate unnecessary enforcement. There is no conflict between writing correct, well-instrumented code and enabling aggressive static optimization; in fact, the two concerns reinforce each other. More broadly, how much dynamic enforcement a program requires is an analyzable property of the program itself, not a fixed consequence of the IFC system chosen.

In the future, combining MinIF with Iodine-style profile-guided optimization [48] could yield more aggressive optimizations by exploiting runtime invariants that static analysis alone cannot establish. A gradual variant [43] could extend coverage to regions where type inference falls short, and an ownership discipline [33] would handle programs with shared mutable state.

## Acknowledgments

David Basin was supported in part by the Novo Nordisk Foundation through a RECRUIT Sabbatical Grant NNF24OC0089993. We thank the anonymous OOPSLA reviewers for their careful reading and constructive feedback, and Abhiroop Sarkar for his comments and guidance on an early draft of this paper.

## Data-Availability Statement

The MinIF transpiler, the WebTTC runtime, and all benchmark programs are archived on Zenodo [52]. Development continues at https://github.com/dgalan7/minif. The extended version of this paper, which additionally includes the appendices with the complete operational semantics, definitions, and proofs, is also archived on Zenodo [23].

## References

[1] Daniel Hedin and Andrei Sabelfeld. 2012. A Perspective on Information-Flow Control. In *Software Safety and Security*. IOS Press, 319–347. doi:10.3233/978-1-61499-028-4-319.

[2] Fred B. Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security*, 3, 1, 30–50. doi:10.1145/353323.353382.

[3] Andrei Sabelfeld and Andrew Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 1, 5–19. doi:10.1109/JSAC.2002.806121.

[4] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *23rd IEEE Computer Security Foundations Symposium (CSF)*, 186–199. doi:10.1109/CSF.2010.20.

[5] Andrew Bedford, Josée Desharnais, Théophane G. Godonou, and Nadia Tawbi. 2014. Enforcing Information Flow by Combining Static and Dynamic Analysis. In *Foundations and Practice of Security*. Jean Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquin Garcia-Alfaro, and Nur Zincir Heywood, (Eds.) Springer, 83–101. ISBN: 978-3-319-05302-8. doi:10.1007/978-3-319-05302-8_6.

[6] Eduardo Geraldo, José Fragoso Santos, and João Costa Seco. 2021. Hybrid Information Flow Control for Low-Level Code. In *Software Engineering and Formal Methods*. Radu Calinescu and Corina S. Păsăreanu, (Eds.) Springer, 141–159. ISBN: 978-3-030-92124-8. doi:10.1007/978-3-030-92124-8_9.

[7] François Hublet, David Basin, and Srđan Krstić. 2024. User-Controlled Privacy: Taint, Track, and Control. *Privacy Enhancing Technologies*, 2024, 1, 597–616. doi:10.56553/popets-2024-0034.

[8] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *4th ACM Symposium on Haskell*. ACM, 95–106. ISBN: 978-1-4503-0860-1. doi:10.1145/2034675.2034688.

[9] Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for Practical Programming with Information-Flow Control. In *Programming Languages and Systems*. Chung-chieh Shan, (Ed.) Springer, 217–232. ISBN: 978-3-319-03542-0. doi:10.1007/978-3-319-03542-0_16.

[10] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies. *SIGPLAN Not.*, 47, 1, 85–96. doi:10.1145/2103621.2103669.

[11] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. *ACM SIGPLAN Notices*, 41, 1, 79–90. doi:10.1145/1111320.1111045.

[12] François Hublet, David Basin, and Srđan Krstić. 2023. Enforcing the GDPR. In *28th European Symposium on Research in Computer Security (ESORICS)*. Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, (Eds.) Vol. 14345. Springer, 400–422. doi:10.1007/978-3-031-51476-0_20.

[13] Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4, 1, 2–16. doi:10.1007/s10207-004-0046-8.

[14] Gordon Plotkin. 2004. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60–61, 17–139. doi:10.1016/j.jlap.2004.05.001.

[15] Robin Milner. 1980. *A Calculus of Communicating Systems*. Number 92 in *LNCS*. Springer. ISBN: 978-3-540-10235-9 978-3-540-38311-6. doi:10.1007/3-540-10235-3.

[16] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 11–11. ISBN: 978-0-8186-0410-2. doi:10.1109/SP.1982.10014.

[17] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security*, 18, 6, 1157–1210. doi:10.3233/JCS-2009-0393.

[18] Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *4th Workshop on Programming Languages and Analysis for Security* (PLAS). ACM, 113–124. doi:10.1145/1554339.1554353.

[19] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[20] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM*, 19, 5, 236–243. doi:10.1145/360051.360056.

[21] Joseph W. Cutler et al. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.*, 8, 670–697, OOPSLA1. doi:10.1145/3649835.

[22] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: mixing static and dynamic typing for information-flow control in Haskell. In *20th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2015). ACM, 289–301. ISBN: 978-1-4503-3669-7. doi:10.1145/2784731.2784758.

[23] Daniel Galán Pascual, François Hublet, Srđan Krstić, Roman Fischer, Colin Pfingstl, and David Basin. 2026. A type system for optimizing dynamic IFC (extended version). (2026). doi:10.5281/zenodo.21891444.

[24] Philip Wadler. 1991. Is there a use for linear logic? *SIGPLAN Not.*, 26, 9, 255–273. doi:10.1145/115866.115894.

[25] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th Symposium on Principles of Programming Languages* (POPL). ACM, 238–252. ISBN: 978-1-4503-7350-0. doi:10.1145/512950.512973.

[26] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *9th Symposium on Principles of Programming Languages (POPL)*. ACM, 207–212. ISBN: 978-0-89791-065-1. doi:10.1145/582153.582176.

[27] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA. ISBN: 978-0-262-16209-8.

[28] John Rushby. 1992. Noninterference, Transitivity, and Channel-Control Security Policies. Tech. rep. CSL-92-2. SRI International, Menlo Park, CA, USA.

[29] Mark Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.*, 10, 4, 352–357. doi:10.1109/TSE.1984.5010248.

[30] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-Time Enforcement of Nonsafety Policies. *ACM Trans. Inf. Syst. Secur.*, 12, 3, 19:1–19:41. doi:10.1145/1455526.1455532.

[31] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *26th Symposium on Principles of Programming Languages (POPL)*. ACM, 228–241. ISBN: 978-1-58113-095-9. doi:10.1145/292540.292561.

[32] Vincent Simonet. 2003. The Flow Caml System. https://pauillac.inria.fr/~simonet/soft/flowcaml/flowcaml-manual.pdf.

[33] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. 2024. Cocoon: Static Information Flow Control in Rust. *Proc. ACM Program. Lang.*, 8, 166–193, OOPSLA1. doi:10.1145/3649817.

[34] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st Symposium on Security and Privacy (S&P)*. IEEE, 317–331. doi:10.1109/SP.2010.26.

[35] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *16th International Symposium on Software Testing and Analysis* (ISSTA). ACM, 196–206. doi:10.1145/1273463.1273490.

[36] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *8th International Conference on Virtual Execution Environments (VEE)*. ACM, 121–132. doi:10.1145/2151024.2151042.

[37] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems*, 32, 2, 1–29. doi:10.1145/2619091.

[38] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 85–96. doi:10.1145/1037187.1024404.

[39] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: a flexible information flow architecture for software security. In *34th International Symposium on Computer Architecture (ISCA)*. Dean M. Tullsen and Brad Calder, (Eds.) ACM, 482–493. doi:10.1145/1273440.1250722.

[40] Tim Disney and Cormac Flanagan. 2011. Gradual Information Flow Typing. In *International Workshop on Scripts to Programs (STOP)*.

[41] Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *26th Computer Security Foundations Symposium (CSF)*. IEEE, 224–239. doi:10.1109/CSF.2013.22.

[42] Luminous Fennell and Peter Thiemann. 2016. LJGS: Gradual Security Types for Object-Oriented Languages. In *30th European Conference on Object-Oriented Programming (ECOOP)*. Shriram Krishnamurthi and Benjamin S. Lerner, (Eds.) Vol. 56. LIPIcs, 9:1–9:26. ISBN: 978-3-95977-014-9. doi:10.4230/LIPIcs.ECOOP.2016.9.

[43] Tianyu Chen and Jeremy G. Siek. 2024. Quest Complete: The Holy Grail of Gradual Security. *Proc. ACM Program. Lang.*, 8, 1609–1632, PLDI. doi:10.1145/3656442.

[44] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *39th Symposium on Principles of Programming Languages (POPL)*. ACM, 481–494. doi:10.1145/2103656.2103714.

[45] John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. 2024. Type-Based Gradual Typing Performance Optimization. *Proc. ACM Program. Lang.*, 8, POPL. doi:10.1145/3632931.

[46] Scott Moore and Stephen Chong. 2011. Static Analysis for Efficient Hybrid Information-Flow Control. In *IEEE 24th Computer Security Foundations Symposium (CSF)*. IEEE, 146–160. ISBN: 978-1-61284-644-6. doi:10.1109/CSF.2011.17.

[47] Marco Vassena. 2017. *MAC, A Verified Information-Flow Control Library*. Licentiate. Chalmers Tekniska Hogskola Sweden. 114 pp. ISBN: 978-1-392-64647-2. Retrieved Feb. 17, 2026 from https://www.proquest.com/docview/2399697222/abstract/ED91154E67AE458EPQ/1.

[48] Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. 2019. Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In *40th Symposium on Security and Privacy (S&P)*. IEEE, 490–504. doi:10.1109/SP.2019.00043.

[49] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. In 30th USENIX Security Symposium (USENIX Security), 1665–1682. ISBN: 978-1-939133-24-3.

[50] Yiyu Zhang, Tianyi Liu, Yueyang Wang, Yun Qi, Kai Ji, Jian Tang, Xiaoliang Wang, Xuandong Li, and Zhiqiang Zuo. 2024. HardTaint: Production-Run Dynamic Taint Analysis via Selective Hardware Tracing. *Proc. ACM Program. Lang.*, 8, 1615–1640, OOPSLA2. doi:10.1145/3689768.

[51] Vincent Beardsley, Chris Xiong, Ada Lamba, and Michael D. Bond. 2025. Carapace: Static–Dynamic Information Flow Control in Rust. *Proc. ACM Program. Lang.*, 9, 364–392, OOPSLA1. doi:10.1145/3720427.

[52] [SW] Daniel Galán Pascual, François Hublet, Srđan Krstić, Roman Fischer, Colin Pfingstl, and David Basin, Artifact for "A Type System for Optimizing Dynamic IFC" 2026. doi:10.5281/zenodo.21773325.

Received 17 March 2026; revised 21 July 2026; accepted 7 August 2026