Asynchronous I/O for Information-Flow Control in TypeScript

Max Pieter Bezemer & Jakob Splidsboel Eg

July 23, 2026

Abstract

Writing secure software is challenging; implementation errors can leak sensitive data to unauthorized parties. To address this problem, information-flow control tracks how sensitive data propagates through programs. Recently, ifc-ts provides compile-time IFC for TypeScript in the form of a library. We extend ifc-ts to natively support promise-based asynchronous I/O, and make security-labeled values be represented as closures, to improve encapsulation. Together, our modifications make ifc-ts more closely meet the needs of modern JavaScript developers.

AI Usage Disclaimer

In this project, ChatGPT 5.1 and Claude Sonnet 4.5 were used as coding and writing assistants during the preparation of the report. The authors take full responsibility for the content, correctness, and final wording of the manuscript and the accompanying code.

Contents

1 Introduction **2**

2 Background **3**

2.1 Fundamentals of Information-Flow Control . . . 3

2.2 Enforcement Mechanisms . . . 4

2.3 IFC in Modern Programming Languages . . . 5

2.4 ifc-ts: Static IFC for TypeScript . . . 6

2.5 The Gap: ifc-ts and Asynchronous Operations . . . 7

3 Our Contribution **8**

3.1 Two Architectural Approaches . . . 8

3.2 Implementation of LIO<Lpc, L, Promise<V>> . . . 9

3.2.1 Changes to I/O Types . . . 9

3.2.2 The Input Operation . . . 9

3.2.3 The Output Operation . . . 10

3.2.4 The Bind Operations . . . 10

3.3 Preventing Label Erasure Through Closures . . . 12

3.3.1 Integration with Monad Operations . . . 13

3.3.2 Design rationale . . . 14

3.4 Relationship to Gradual Typing . . . 14

3.5 Validation . . . 15

3.5.1 Test Setup . . . 15

3.5.2 Chaining Multiple Async Operations . . . 16

3.5.3 Manipulating data using `bindAsync` . . . 16

4 Discussion **17**

4.1 Did We Solve the Right Problem? . . . 17

4.2 The Cost of Keeping Labels at the Type Level . . . 18

4.3 Closure-Based Labels . . . 19

4.4 The Synchronous-Asynchronous Divide . . . 19

4.5 What Have We Actually Accomplished? . . . 20

5 Conclusion **20**

5.1 Future Work . . . 20

1 Introduction

Consider a web server handling user requests. Each user's session contains sensitive data: authentication tokens, personal information, private messages. The server processes multiple requests concurrently, fetching data from databases, calling external APIs, and logging events. A single implementation error could leak one user's credentials into another user's response or write secrets to public logs. Providing formal guarantees that such leaks cannot occur remains a fundamental challenge in secure systems design.

Information-flow control (IFC) [1] addresses this problem by tracking how sensitive data propagates through programs. Unlike access control, which enforces permissions at the point of data access, IFC enforces security constraints on data propagation throughout program execution. The foundational security property is noninterference: an attacker observing low-sensitivity outputs cannot infer anything about high-sensitivity inputs. IFC systems assign security labels to data organized in a lattice [2] with a partial order representing permissible flows. This lattice enforces that high-sensitivity data cannot flow to low-sensitivity outputs through either explicit flows (direct assignment) or implicit flows (control-flow dependencies).

Early work on IFC established both the theoretical foundations and practical mechanisms for enforcement. Denning and Denning [1] demonstrated that information flow could be tracked dynamically through runtime monitoring, while subsequent work showed that static type systems could enforce these properties at compile-time [3]. Sabelfeld and Myers [4] provided a comprehensive survey of language-based information-flow security, establishing the theoretical framework that guides modern implementations.

The state of the art in IFC has focused on extending programming languages with information-flow labels and providing specialized compilers or checkers for these extended languages. Systems like Jif [5] extended Java with security labels, and FlowCaml [6] added IFC to OCaml. While powerful, these approaches create significant adoption barriers: developers must learn language extensions, install specialized compilers that may not integrate with their IDEs, and work with tools that lack the maturity and ecosystem support of mainstream programming languages.

Library-based approaches emerged as an alternative that leverages the type systems of existing languages. Stefan et al.'s LIO [7] embedded IFC in Haskell using a labeled I/O monad, achieving security enforcement through the host language's type system rather than language extensions. This approach requires no custom compiler as the standard Haskell compiler performs all security checks through normal type-checking. LIO demonstrated that sufficiently expressive type systems (with features like type classes and GADTs) could encode security constraints as library abstractions.

The ifc-ts library [8] adapts this monadic approach to TypeScript, one of the most widely used programming languages for both client and server applications. This represents a significant contribution: it was not clear that TypeScript's type system, which lacks type classes and GADTs, could express the constraints required for compile-time IFC. The library encodes security lattices using TypeScript's native features: principals are string literal types, labels are unions, and can-flow-to relations use subtyping. Developers annotate data sources and sinks with security labels, structure I/O operations within a labeled monad, and rely on TypeScript's type checker to reject insecure flows at compile-time. Well-typed ifc-ts programs satisfy noninterference. This approach integrates with standard development tooling, providing full IDE support, and requires neither specialized compilers nor runtime monitoring nor specialized IDE extensions. The usage

patterns follow familiar idioms (similar to promise chaining) that TypeScript developers already understand.

The library's examples, however, use only synchronous file operations. While async I/O is possible in principle (by defining readers that return promises), the monadic structure makes this awkward in practice. Promises must nest inside labeled values inside the monad, creating layers that JavaScript's `await` cannot penetrate. Developers must manually manage promise resolution, label unwrapping, and re-wrapping at each step. The library also represents labeled values as tuples `[L, V]`, which allows accidental label erasure through destructuring. While not a fundamental security flaw, this makes it easy to extract raw values without recognizing the boundary being crossed.

This paper extends ifc-ts with native support for promise-based asynchronous I/O. We redesign the I/O primitives so that promises wrap labeled values, introduce `bindAsync` for chaining async operations, and replace tuple-based labels with closure-based encapsulation that prevents accidental destructuring. The monad structure and security lattice remain unchanged. We demonstrate that the extended library handles async file I/O while preserving compile-time noninterference guarantees.

2 Background

2.1 Fundamentals of Information-Flow Control

Information-flow control (IFC) enforces security policies by tracking how data propagates through programs and preventing unauthorized flows. IFC operates at a different abstraction level than access control: while access control mediates whether a principal may read or write an object, IFC constrains how information moves after access is granted [1].

IFC systems formalize security as a lattice of labels with a partial order representing permissible flows [2]. The core enforcement mechanism (whether static or dynamic) rejects programs that permit flows from higher-sensitivity sources to lower-sensitivity sinks.

Data can leak through explicit and implicit flows [1]. An *explicit flow* occurs when sensitive data is directly assigned to a lower-sensitivity variable or output channel. An *implicit flow* occurs when sensitive data influences control flow, thereby indirectly affecting observable outputs. Consider the following example:

```
// Explicit flow: direct assignment
let publicOutput = secret; // ILLEGAL: High -> Low

// Implicit flow: control-flow dependency
if (secret > 0) {
    publicOutput = "positive";
} else {
    publicOutput = "non-positive";
}
// ILLEGAL: publicOutput reveals information about secret
```

Listing 1: Explicit and implicit information flows

The explicit flow violates confidentiality directly. The implicit flow is subtler: no direct assignment from `secret` to `publicOutput` occurs, yet an observer learns `secret > 0` from the output value. IFC systems prevent implicit flows by tracking a program counter (PC) label representing the sensitivity of the current execution context. When a high-sensitivity value affects a conditional, the PC label rises within that branch, prohibiting assignments to low-sensitivity variables.

Denning formalized these intuitions using lattice theory [2]. A security lattice is a tuple $(L, \sqsubseteq)$ where L is a set of security labels and $\sqsubseteq$ (''can-flow-to'') is a partial order. The lattice provides join ($\sqcup$) and meet ($\sqcap$) operators. The join computes the least upper bound: when computation combines data labeled ℓ_1 and ℓ_2, the result must be labeled at least $\ell_1 \sqcup \ell_2$ to preserve security.

A canonical instantiation is the powerset lattice over principals. Given principals $\mathcal{P} = \{\text{Alice}, \text{Bob}\}$, labels are $L = 2^{\mathcal{P}}$ with $\sqsubseteq$ as set inclusion. The label $\{\text{Alice}\}$ means ''data labeled $\{\text{Alice}\}$ is sensitive to Alice'' while $\{\text{Alice}, \text{Bob}\}$ means ''data labeled $\{\text{Alice}, \text{Bob}\}$ is sensitive to Alice and Bob''. Join is set union, meet is intersection, $\bot = \emptyset$ represents public data, and $\top = \mathcal{P}$ represents data no principal may observe. This relationship between labels enable the prevention of illegal operations. For example, writing data labeled Alice, Bob to a sink labeled Alice is not allowed since that leaks Bob data to Alice.

2.2 Enforcement Mechanisms

IFC enforcement varies along two independent dimensions: *granularity* (what carries labels) and *mechanism* (how enforcement is performed). Understanding this design space is essential for evaluating different IFC approaches and their trade-offs.

Granularity: Fine-grained vs. coarse-grained tracking. Fine-grained IFC systems attach labels to individual values, tracking information flow at the level of variables and expressions [4]. When computation combines two labeled values, the resulting label is the join of the inputs. This yields high precision: different values in the same scope may carry different labels, enabling the system to identify exactly which data influenced which outputs. The downside is complexity. In static systems this complexity appears in the type rules and annotations; in dynamic systems, an information-flow monitor may need to propagate labels at runtime. While dynamic fine-grained systems can incur nontrivial overhead, static fine-grained type systems impose no runtime cost because all reasoning occurs at compile time.

Coarse-grained IFC, in contrast, assigns labels to execution contexts rather than individual values [7]. A computation context (e.g., a thread, a monad, or a sandbox) maintains a security level; reading high-sensitivity data raises the context's level, after which the computation may only write to sinks at least as restrictive. This approach simplifies enforcement and reduces bookkeeping, though at the cost of precision: unrelated low-sensitivity values may be implicitly over-tainted once the context is elevated.

Mechanism: Static vs. dynamic enforcement. Static IFC integrates security labels into a language's type system, rejecting programs that could violate policy at compile time. Such systems provide strong guarantees, typically noninterference for well-typed programs [3], with zero runtime overhead. The limitation is conservativeness: because static analyses must over-approximate all possible executions [4], they may reject secure programs whose safety cannot be proven within the type system. Historically, such systems often required specialized compilers or language extensions [6, 5].

Dynamic IFC enforces information-flow policies at runtime using security monitors. Unlike static systems, dynamic monitors can make decisions based on actual runtime values and therefore avoid some conservative rejections. However, dynamic enforcement is fundamentally weaker: it detects illegal flows only along the path taken during execution and cannot prevent leaks via *branches not taken*. For example:

$$\text{low} = 0; \quad \text{if } (high \bmod 2 = 1)\{\text{low} = 1\}; \quad \text{output}_L(\text{low})$$

If $high = 5$, a dynamic monitor halts execution when the assignment to `low` is attempted. But if $high = 6$, the branch is skipped and the program completes normally, yet the observed low output reveals that the branch was not taken, leaking that *high* must be even. This illustrates that dynamic enforcement cannot, in general, rule out implicit or control-flow leaks stemming from unexecuted code paths.

2.3 IFC in Modern Programming Languages

The evolution from specialized research languages to practical libraries represents a shift in IFC adoption, making security guarantees accessible without abandoning existing development ecosystems. Though still not widely adopted, IFC sees use in domain-specific areas where security critical code needs non-interference guarantees, such as the SPARK programming language [9].

Using the classification of the section above, Jif [5] and FlowCaml [6] occupy the fine-grained static quadrant. Both extend statically-typed languages (Java and OCaml respectively) with security type annotations, using custom compilers to verify that programs satisfy information-flow policies. Jif supports a decentralized label model where multiple principals can express policies; FlowCaml integrates security types with ML's type inference. These systems track labels on individual values through the type system, achieving precise compile-time verification.

While powerful, compiler-based approaches created adoption barriers. Developers needed specialized build tools, couldn't leverage standard IDEs fully, and faced challenges integrating with existing libraries. The requirement for language extensions and custom toolchains hindered mainstream adoption despite strong security guarantees.

The library-based approach emerged as a solution: rather than extending the language, embed IFC constraints within the existing type system using advanced type features. This shift enabled IFC adoption using standard compilers and toolchains.

Stefan et al.'s LIO [7] exemplifies the coarse-grained dynamic approach. LIO provides a monadic API for Haskell where all I/O operations occur within an `LIO` monad tracking security labels. Each computation maintains a floating label representing its current security level; reading high-sensitivity data raises this label, and the library prevents writes to sinks whose labels are not at least as restrictive (i.e., computations may only read from below and write to above). LIO enforces these constraints at runtime, offering flexibility for Haskell's pure functional setting while imposing minimal overhead compared to fine-grained tracking.

This library-based model offers crucial advantages: developers use standard compilers, integration with existing code is smoother (security-checked portions are encapsulated within the monad), and no separate analysis tools are required.

JavaScript's ubiquity in web and server applications makes it a critical target for IFC. JSFlow [10] occupies the fine-grained dynamic quadrant, instrumenting JavaScript to track labels on individual values throughout execution. Every operation (assignments, arithmetic, property access) propagates labels. This provides precise tracking but incurs substantial overhead: JSFlow reports 1.6 to 2.9× slowdown for realistic applications [10]. The approach requires source-to-source transformation and a custom runtime, limiting practical adoption.

2.4 ifc-ts: Static IFC for TypeScript

TypeScript, a typed superset of JavaScript with static type checking, offers unique opportunities for library-based IFC [11]. Though TypeScript's type system was designed for correctness rather than security, its advanced features (conditional types, mapped types, string literal types) prove sufficiently expressive to encode security constraints [12, 13, 14]. TypeScript compiles to standard JavaScript, enabling security-checked code to run in any JavaScript environment without modified runtimes.

The ifc-ts library [8] brings static IFC to TypeScript through a library implementation based on the coarse-grained, static approach pioneered by LIO [15]. Like LIO, ifc-ts uses a monadic structure where computations execute within a labeled context tracking the program counter (PC) label. Unlike LIO's runtime enforcement, ifc-ts encodes these constraints at the type level, achieving compile-time verification through TypeScript's standard type checker. The ifc-ts monad encodes three components in a tuple representation: the PC label (tracking implicit flows from control dependencies), the data label (tracking explicit flows from data dependencies), and the computation's value. The library implements the security lattice directly using TypeScript's type system. A principal is a string literal type (e.g., `"Private"`), and a security level is a union of principals (e.g., `"Private" | "Public"`). The powerset lattice emerges naturally from TypeScript's type operations: union types model set union, intersection types model set intersection, and TypeScript's structural subtyping models the can-flow-to relation. The `lub` (least upper bound) function computes the join of two labels, which in the powerset lattice corresponds to their union.

```
// Labels for private and public data
const privateLevel = lub("Private", publicLevel);
const publicLevel: Public = "Public";

type Private = typeof privateLevel; // "Public" | "Private"
type Public = typeof publicLevel; // "Public"
```

Listing 2: Lattice construction

The monad's `bind` operation enforces information-flow constraints through type-level constraints. When sequencing two computations, the type system verifies that the first computation's data label can flow to the second computation's PC label, preventing implicit flows. Similarly, I/O operations enforce constraints through the type system: reading from a labeled source produces labeled data, and writing to a sink requires that the data's label can flow to the sink's label.

ifc-ts delivers several advantages over compiler-based approaches. First, it requires no specialized tooling:

the standard TypeScript compiler performs all security checking during normal type-checking. Second, it integrates seamlessly with existing development workflows: developers use their standard IDEs with full TypeScript support, seeing security violations as type errors with immediate feedback during development. Third, usage patterns follow familiar TypeScript idioms (similar to promise chaining), reducing the learning curve for TypeScript developers. Fourth, security-checked code compiles to standard JavaScript with zero runtime overhead for security enforcement beyond standard TypeScript compilation.

2.5 The Gap: ifc-ts and Asynchronous Operations

While `ifc-ts` provides static IFC for synchronous TypeScript code, its current API imposes friction when modeling asynchronous execution. The monadic operations (`input`, `output`, and `bind`) expect immediate, non-`Promise`-returning effects. No combinators exist for sequencing labeled asynchronous computations while preserving security guarantees. This restriction renders the library impractical for real-world TypeScript applications.

Modern TypeScript applications are fundamentally asynchronous. Node.js servers handle concurrent requests via an event loop: web applications fetch data via promises; real-time systems rely on callbacks. Blocking operations (such as synchronous I/O) are considered an anti-pattern because they stall the event loop, destroying the scalability benefits of Node's concurrency model [16]. Listing 3 demonstrates the consequence: attempting to use async I/O within ifc-ts is awkward and forces developers to manually manage promise resolving, destructure labeled values, and bypass the monad's security enforcement. Furthermore, because the monadic operations cannot sequence computations that return unresolved promises, asynchronous I/O cannot be expressed within the current interface. Forcing promises to resolve before invoking the next monadic step effectively reintroduces blocking behavior, eliminating the benefits of asynchronous I/O.

```
// Create async source - readers return Promises
const asyncSrcPrivate = src(privateLevel, async () => {
    const buffer = Promise.resolve(readFile("examples/io-examples/private-data.txt"));
    return buffer.toString();
});

// Create async sink - writers accept Promises or handle async values
const asyncSnkPrivate: Snk<Private, string> = snk(privateLevel as Private, async (s:
    string) => {
    await writeFile("examples/io-examples/private-output.txt", s);
});

const privateToPrivate = async () => {
    const [_, __, labeledPromise] = input(asyncSrcPrivate); // Extracting from monadic
        context
    const [l, promiseValue] = labeledPromise; // Separating promise from label
    const resolvedValue = await promiseValue;
    return output(asyncSnkPrivate)(label(l, resolvedValue));
};
```

Listing 3: Manual async decomposition

As the examples above illustrate, developers who need to work with promises or `async/await` must step outside the provided abstractions. This can be considered cumbersome and potentially dangerous since non-interference guarantees no longer hold for data outside of the labeled context. Additionally, we see the constraint of having to resolve the promise before entering monadic computations. In the example above this occurs at the source, but regardless of where the resolution is forced, the effect is the same: asynchronous I/O is reduced to a synchronous operation.

This motivates the contributions made in our project; provide abstractions for ifc-ts that enable a user of the API to work with asynchronous I/O without having to escape the labeled context.

3 Our Contribution

This chapter presents our extension of ifc-ts to support asynchronous IFC. We begin by discussing the architectural design space for integrating promises with the LIO monad, explaining the trade-offs between different approaches. We then describe our implementation of the `LIO<Lpc, L, Promise<V»` architecture, which embeds promises within the monad's value type. A challenge in this design is preventing label erasure when promises resolve; we address this through closure-based encapsulation of labeled values. Finally, we demonstrate how our design enables gradual security typing and validate that compile-time security guarantees are preserved across asynchronous boundaries.

3.1 Two Architectural Approaches

We identified two possible designs, each with distinct trade-offs.

Approach A: Promise<LIO<Lpc, L, V>>

In this approach, promises wrap entire monad computations. A value of type `Promise<LIO<Lpc, L, V»` represents an asynchronous computation that, when resolved, produces a monad containing a value.

This design could potentially leverage existing Promise utilities and follows familiar JavaScript patterns. However, it creates a "suspended computation of a computation" where the security-checking monad itself becomes the subject of asynchronous execution. The program counter and data labels exist inside the promise, potentially becoming detached during promise resolution. Furthermore, it is less clear when security checks occur relative to async execution, as the monad structure is only available after the promise resolves.

Approach B: LIO<Lpc, L, Promise<V>>

In this approach, promises are part of the value type V within the monad. A value of type `LIO<Lpc, L, Promise<V»` represents a monad computation where the contained value is itself a promise.

We chose this design for several reasons. First, it keeps the program counter and data labels attached to the monad at the type level, providing a clear separation between type-level security checking and value-level async execution. The monad structure, including its security labels, is available before any async execution

begins. Second, this approach makes explicit when asynchronous execution happens: the LIO structure exists first, then the promise within it executes.

The primary limitation of this approach is that all monad operations must be promise-aware, requiring modifications to `bind`, `input`, and `output`. However, we view this explicitness as beneficial rather than problematic, as it makes the interaction between security checking and asynchronous execution more transparent.

We implemented Approach B due to its clearer separation of concerns between compile-time security and runtime asynchrony. The remainder of this chapter describes our implementation.

3.2 Implementation of LIO<Lpc, L, Promise<V>>

Our extension modifies the monad's I/O operations and bind function to work with Promise-typed values, while keeping the tuple-based monad representation from the original ifc-ts. The monad structure `[Contravariant<Lpc>, L, V]` remains unchanged; only the type of V changes to be promise-typed for async operations.

3.2.1 Changes to I/O Types

We modify the reader and writer types to return promises:

```
type Reader<I> = () => Promise<I>;
type Writer<O> = (o: O) => Promise<void>;
```

Listing 4: Async I/O types

This change is minimal and directly reflects how modern TypeScript I/O operates. Database queries, HTTP requests, and file operations in TypeScript all return promises, so our async readers and writers can directly wrap these operations without additional adaptation layers.

3.2.2 The Input Operation

The `input` operation reads from a source and labels the resulting value. Our implementation ensures that labels are attached before values enter the monad:

```
export function input<L extends Level, I>([l, r]: Src<L, I>):
    LIO<Top, Bot, Promise<Labeled<L, I>>> {
    const labeledPromise = r().then(i => label(l, i));
    return [toContravariant(topLevel), botLevel, labeledPromise];
}
```

Listing 5: Input operation for async sources

The reader `r()` returns a `Promise<I>`. We transform this into a `Promise<Labeled<L, I»` using the promise's `then` method. The resulting monad has value type `Promise<Labeled<L, I»`, meaning the label L is part of the promised value's type.

This type structure is crucial for security. When the promise resolves, the type system still knows the value is labeled with L. The value is immediately encapsulated in a labeled wrapper, preventing it from ever existing as a bare type I. The mechanism for this encapsulation is described in detail in Section 3.3.

3.2.3 The Output Operation

The `output` operation writes a labeled value to a sink. Our implementation enforces the can-flow-to relation at compile-time:

```
export function output<Lsink extends Level, Ldata extends Lsink, O>(
    [lo, w]: Snk<Lsink, O>
): (lv: Labeled<Ldata, O>) => LIO<Lsink, Bot, Promise<null>> {
    return (lv: Labeled<Ldata, O>) => {
        const o = lv.unsafeGetValue();
        const writePromise = w(o).then(() => null);
        return [toContravariant(lo as any), botLevel, writePromise];
    };
}
```

Listing 6: Output operation for async sinks

The type constraint `Ldata extends Lsink` enforces the can-flow-to relation before any code executes. If a developer attempts to pass `Labeled<Private, string>` to a public sink, TypeScript rejects the code at compile-time. This early detection is particularly important for asynchronous operations, where runtime errors might only manifest under specific timing conditions or execution interleavings. By catching violations at compile-time, we avoid the possibility of security bugs that only appear during concurrent execution.

The method `lv.unsafeGetValue()` extracts the value from its labeled wrapper. The `unsafe` prefix signals that this operation crosses a security boundary. We discuss the design of labeled values and the role of this method in Section 3.3.

3.2.4 The Bind Operations

We supplement the library with a new bind operation to handle asynchronous computations.

Synchronous Bind The current, syncrhonous `bind` operation chains non-promise-based monadic computations:

```
export function bind<
    Lpc extends Level,
    L extends Rpc,
    V,
    Rpc extends Level,
    R extends Level,
    W
>(
    m: LIO<Lpc, L, V>,
```

```
    f: (_: V) => LIO<Rpc, R, W>
): LIO<GLB<Lpc, Rpc>, LUB<L, R>, W> {
    const [lpc, l, v] = m;
    return f(v);
}
```

Listing 7: Synchronous bind operation

This version handles computations where values are not promise-wrapped. It extracts the value `v` from the monad and passes it to the continuation function `f`. The type signature enforces security constraints through the `L extends Rpc` constraint, preventing implicit flows.

Asynchronous Bind

The `bindAsync` operation handles promise-based computations:

```
export function bindAsync<
    Lpc extends Level,
    L extends Rpc,
    V,
    Rpc extends Level,
    R extends Level,
    W
>(
    m: LIO<Lpc, L, Promise<Labeled<L, V>>>,
    f: (lv: Labeled<L, V>) => LIO<Rpc, R, Promise<W>>
): LIO<GLB<Lpc, Rpc>, LUB<L, R>, Promise<W>> {
    const [lpc, l, promiseLV] = m;

    const resultPromise = promiseLV.then((labeledValue) => {
        const [_, __, promiseW] = f(labeledValue);
        return promiseW;
    });

    return [toContravariant(topLevel as any), l, resultPromise]
        as LIO<GLB<Lpc, Rpc>, LUB<L, R>, Promise<W>>;
}
```

Listing 8: Asynchronous bind operation

This function uses JavaScript's `then` method to chain promises at the value level, while using the same type signature to perform security checking at the type level.

A critical design decision is that `bindAsync` does not automatically unwrap labeled values. When used with the result of `input`, which returns `LIO<Top, Bot, Promise<Labeled<L, I>>>`, the continuation function `f` receives `Labeled<L, V>` rather than the raw value `V`. This preserves the labeled wrapper, ensuring that developers cannot accidentally access sensitive data without explicitly acknowledging the security

boundary through `unsafeGetValue()`.

The continuation function `f` must return `LIO<Rpc, R, Promise<W>>`, making it clear that asynchronous operations are being performed. When the input promise resolves to a labeled value, the continuation is invoked with that labeled value, and its resulting promise is extracted and returned as part of the new monad.

All label computation happens at compile-time through TypeScript's type system. At runtime, the implementation performs only standard promise chaining with no label manipulation or security checks. This achieves near-zero overhead abstraction.

3.3 Preventing Label Erasure Through Closures

The original ifc-ts library represents labeled values as tuples `[L, V]`, which permits straightforward destructuring to access the underlying value. While this is a known limitation in the synchronous setting mentioned by the author of the library [8], asynchronous operations exacerbate the problem: JavaScript's `await` operator encourages developers to extract values from promises, making accidental label erasure more likely.

```
const computation = input(privateSource); // LIO<..., Promise<Labeled<Private, string>>>
const labeledValue = await unsafe_runLIO(computation); // Labeled<Private, string>

// Tuple destructuring extracts bare value:
const [_, value] = labeledValue; // value: string (label erased)
console.log(value); // No compile-time error
```

Listing 9: Label erasure through await and destructuring

As listed above, after destructuring, `value` has type `string` with no type-level connection to the label Private. The type system loses the ability to enforce security policies on this value. The `await` operation itself does not compromise security; the vulnerability arises when tuple destructuring extracts the value from its labeled wrapper. Asynchronous code makes this pattern more likely because developers routinely use `await` to inspect intermediate results.

We address this challenge using closure-based encapsulation. Instead of storing labels and values in tuples or object properties, we store them in function closures:

```
export type Labeled<L extends Level, V> = {
    getLabel: () => L;
    unsafeGetValue: () => V;
};

export function label<L extends Level, V>(l: L, v: V): Labeled<L, V> {
    return {
        getLabel: () => l,
        unsafeGetValue: () => v
    };
}
```

Listing 10: Closure-based labeled values

The label `l` and value `v` are captured in the closure's lexical scope. The returned object exposes only two methods; no properties exist that can be destructured or accessed directly. Attempting to destructure or access fields returns `undefined`:

```
const labeledValue = await unsafe_runLIO(input(privateSource)); // Labeled<Private,
    string>

const [_, value] = labeledValue; // undefined
labeledValue.value; // undefined
labeledValue.unsafeGetValue(); // "secret" - explicit extraction
```

Listing 11: Closures prevent destructuring

This is particularly valuable for asynchronous code. When promises resolve, developers receive `Labeled<L, V>` objects that cannot be accidentally destructured. They must explicitly call `unsafeGetValue()` to extract contents. The method name signals that they are performing a security-critical operation, making such code easier to audit and review.

3.3.1 Integration with Monad Operations

Closures integrate naturally with our monad operations. Internal functions use `unsafeGetValue()` when they need to extract values, but the type system still enforces security constraints.

For example, the `output` function (shown earlier) uses `lv.unsafeGetValue()` to extract the value for writing. However, the type constraint `Ldata extends Lsink` still enforces the can-flow-to relation at compile-time. The closure does not interfere with type-level checking; it adds a runtime boundary that prevents accidental misuse.

Similarly, developers writing their own functions can use `unsafeGetValue()` within monad operations:

```
const transformData = bindAsync(
    input(privateSource),
    (lv: Labeled<Private, string>) => {
        const value = lv.unsafeGetValue();
        const transformed = value.toUpperCase();
        const newLabeled = label(lv.getLabel(), transformed);
        return output(privateSink)(newLabeled);
    }
);
```

Listing 12: Using closures in custom operations

This code uses `bindAsync` to chain the asynchronous input operation with a transformation. The continuation receives the labeled value `lv` and extracts the raw value using `unsafeGetValue()`, transforms it, and re-wraps it with the same label using `getLabel()`. The explicit method calls make it clear that labels are being managed, and the type system ensures that the re-labeled value respects the can-flow-to relation when passed to `output`.

3.3.2 Design rationale

Our goal is to prevent accidental errors, not to prevent deliberate circumvention. A developer who understands the security model and deliberately calls `unsafeGetValue()` may have legitimate reasons. The `unsafe` prefix marks operations where normal guarantees do not hold.

To our knowledge, closures are the strongest encapsulation mechanism available in JavaScript and TypeScript. Unlike languages such as OCaml (with module signatures) or Rust (with privacy modifiers), TypeScript does not provide true compile-time privacy for object properties. Closures leverage JavaScript's function scope to create a runtime encapsulation boundary that cannot be bypassed through normal object or array operations.

This design is similar to how the `unsaferunLIO` function explicitly strips the monad wrapper to execute a labeled computation and extract its final value into untracked code. Both `unsaferunLIO()` and `unsafeGetValue()` are necessary escape hatches that allow IFC code to interact with the broader TypeScript ecosystem. Both are clearly marked to facilitate code review.

3.4 Relationship to Gradual Typing

The ifc-ts library, like its Haskell predecessor LIO, provides an escape mechanism for extracting values from the monad. The original library offers `unsafe_runLIO` for this purpose. This function strips type-level security information, returning a standard `Promise<V>` that can integrate with non-IFC code.

This design reflects a broader pattern in TypeScript development. TypeScript itself embraces gradual typing, allowing developers to mix typed and untyped code within the same codebase. The `any` type serves as an explicit boundary where type guarantees cease to hold. Similarly, our library provides explicit boundaries where security guarantees cease: `unsafe_runLIO` marks the transition from security-typed monadic code to regular asynchronous TypeScript.

The practical implication is that developers need not adopt IFC wholesale. Security-critical sections can be written within the LIO monad, with `unsafe_runLIO` used at well-defined boundaries to integrate with existing code:

```
function secureDatabaseRead(): LIO<Private, Bot, Promise<UserData>> {
    return bind(input(privateSource), (lv) => output(privateSink)(lv));
}

async function main() {
    await unsafe_runLIO(secureDatabaseRead()); // Boundary: security guarantees end here
    // Regular TypeScript code continues
```

```
}
```

Listing 13: IFC integrated with regular TypeScript

The placement of `unsafe_runLIO` calls determines the security perimeter. This mirrors TypeScript's philosophy: add types where they provide value, use `any` at boundaries. For security typing, the analogous approach is: add IFC where it matters, use `unsafe_runLIO` at boundaries. The key difference is that while TypeScript's `any` is often implicit or legacy code, our escape mechanisms are explicit and searchable, facilitating security audits.

3.5 Validation

We validate our implementation through examples demonstrating that the type system enforces information-flow security policies at compile-time for asynchronous operations. Our examples use a simple two-point lattice with Private and Public labels, where Private is more restrictive than Public.

3.5.1 Test Setup

Using the security levels and lattice defined in Listing 2. We define our async sources and sinks:

```
// Async private source (simulates delayed sensitive read)
const privateSource = src<Private, string>(privateLevel, async () => {
    const buffer = await fs.readFile("private-data.txt");
    return buffer.toString("utf8");
});

// Async public source (non-sensitive read)
const publicSource = src<Public, string>(publicLevel, async () => {
    const buffer = await fs.readFile("public-data.txt");
    return buffer.toString("utf8");
});

// Second async public source (same backing file for simplicity)
const publicSource2 = src<Public, string>(publicLevel, async () => {
    const buffer = await fs.readFile("public-data.txt");
    return buffer.toString("utf8");
});

// Private sink overwrites the file so only the latest write remains
const privateSink = snk<Private, string>(privateLevel, async (data: string) => {
    await fs.writeFile("private-sink-output.txt", `PRIVATE OUTPUT: ${data}\n`);
});
```

Listing 14: Asynchronous I/O setup

The private source simulates an asynchronous read (e.g., a database or file) that returns sensitive data after a delay; a public source does the same for non-sensitive data. The sink models a output chanel at the private level. In our case, the top of the lattice.

3.5.2 Chaining Multiple Async Operations

Listing 15 illustrates how `bindAsync` enables sequential composition of asynchronous IFC computations. The example reads from two public sources in turn and combines their contents before writing the result to a private sink. Each call to `bindAsync` ensures that the label on the intermediate value is compatible with the continuation that follows. The final `output` call writes the combined public string to the private sink, and the entire computation type-checks because the can-flow-to relation (Public $\sqsubseteq$ Private) is verified at compile-time.

```
const twoPublicSourcesToPublicSink = bindAsync(
    input(publicSource),
    (public1) => bindAsync(
        input(publicSource2),
        (public2) => {
            const combined = label(
                publicLevel,
                `${public1.unsafeGetValue()} | ${public2.unsafeGetValue()}`
            );
            return output(privateSink)(combined);
        }
    )
);
```

Listing 15: Chaining async operations

3.5.3 Manipulating data using `bindAsync`

Listing 16 shows how asynchronous IFC computations may perform intermediate transformations on labeled values inside before producing a result and writing to a sink. The helper function `dropEveryOtherWord` operates on a labeled private string and returns a new private value. Since the function is asynchronous, the caller uses `ret` to lift the promise into the monad. The `bindAsync` combinators then sequence these steps: the private source is read, the value is transformed, and the result is written to the private sink. Throughout the computation, the label on the data remains `Private`.

```
async function dropEveryOtherWord(lv: Labeled<Private, string>): Promise<Labeled<Private,
     string>> {
    const words = lv.unsafeGetValue().split(/\s+/).filter(Boolean);
    const kept = words.filter((_, idx) => idx % 2 === 0).join("␣");
    return label(privateLevel, kept);
}
```

```
6
7  // read private, drop every other word, write to private sink
8  const sourceManipulateSinkToPrivate = bindAsync(
9      input(privateSource),
10     (lv) => bindAsync(
11         ret(dropEveryOtherWord(lv)),
12         output(privateSink)
13     )
14 );
```

Listing 16: Manipulating data within the monad

This example shows that developers can define their own helper functions to transform labeled data and incorporate them into monadic computations. Because `bindAsync` sequences these transformations while tracking the associated label, custom logic such as string processing, filtering, or restructuring can be performed without compromising the information-flow guarantees.

4 Discussion

This work extends ifc-ts to support asynchronous I/O operations using TypeScript's Promise abstraction. While we successfully preserve compile-time noninterference guarantees across asynchronous boundaries, the design reveals fundamental tensions in library-based IFC for practical programming languages. This section critically examines our architectural decisions, questions whether the problems we solved were necessary to solve, and evaluates whether the resulting system justifies its complexity.

4.1 Did We Solve the Right Problem?

The central claim motivating this work is that the original ifc-ts could not handle asynchronous I/O operations. This claim requires scrutiny. The original ifc-ts `Reader<I>` type is simply `() => I`, a synchronous function returning a value. Nothing in the type system prevents defining `Reader<Promise<I»`, where the reader returns a promise that later resolves to a value. A developer could write:

```
1  const asyncReader: Reader<Promise<string>> =
2      () => fetch('/api/secret').then(r => r.text());
3  const source = src(privateLevel, asyncReader);
4  const computation = input(source); // LIO<Top, Bot, Labeled<Private, Promise<string>>>
```

This works with the original ifc-ts. The promise lives inside the labeled value. To extract it, one must first extract the labeled value from the monad using `unsafe_runLIO`, then destructure the tuple to get the promise, then await it. The security labels are preserved throughout: the promise has type `Promise<string>`, but it came from a `Labeled<Private, ...>` value, so the type system knows its provenance.

So what problem are we actually solving? The issue is not that async I/O is impossible in the original ifc-ts, but that it is awkward. Promises are nested inside labeled values inside monads, creating a layering

mismatch. JavaScript's `await` operator cannot reach through the labeled wrapper to extract the promise. Developers must manually unwrap labels, await promises, then re-wrap results, a ceremony that undermines the monad's abstraction.

Our contribution is thus not enabling something fundamentally new, but rather providing a more ergonomic integration. We invert the nesting: instead of `Labeled<L, Promise<I»`, we use `Promise<Labeled<L, I»`. This allows `bindAsync` to chain promises while preserving labels, letting developers work with standard promise patterns inside security-checked code.

This raises an uncomfortable question: did we make ifc-ts better at handling async I/O, or did we merely make it handle async I/O *the way we wanted to write it*? The original design forces developers to think carefully about labels when working with promises. Our design makes promises easier to use, but at the cost of additional complexity in the implementation and API surface (`bind` vs `bindAsync`, synchronous vs asynchronous I/O functions). Whether this trade-off improves the library or merely shifts complexity around is debatable.

4.2 The Cost of Keeping Labels at the Type Level

Our architectural choice, `LIO<Lpc, L, Promise<V»` rather than `Promise<LIO<Lpc, L, V»`, prioritizes keeping security labels visible to the type checker before any asynchronous execution begins. This seems principled: security properties should be checkable statically, not dependent on runtime promise resolution. But this decision cascades through the entire implementation.

Because promises live inside the monad's value type, every operation that touches values must become promise-aware. The original `input` and `output` functions worked with plain values; ours must work with promises. The original `bind` was a simple function application `f(v)`; ours requires `then`-chaining and promise unwrapping. We effectively forked the monad's API into synchronous and asynchronous variants.

The alternative, `Promise<LIO<...»`, would avoid this. Promises would wrap entire computations, and the monad's internal operations would remain unchanged. The original `bind`, `input`, and `output` would work without modification. Async handling would be external to the monad, matching JavaScript's philosophy where async is a wrapper around normal code (`async function` wraps normal functions, `Promise.resolve` wraps values).

Why did we reject this? Our argument is that labels must be type-checkable before promises resolve. But this argument assumes that type-checking happens once, at compile time, on the entire program. In reality, TypeScript's type system cannot reason about the contents of a `Promise<T>` until the promise resolves. Whether that `T` is `LIO<Lpc, L, V>` or just `V`, the type checker treats promises as opaque until await or then-chaining occurs. The security labels are in the type either way.

The honest assessment is that we chose `LIO<Lpc, L, Promise<V»` because it felt more principled. Security checking should be the outer layer, with async execution nested inside. But this aesthetic preference came at a real cost: API complexity, implementation overhead, and forcing all operations to handle promises. A more pragmatic design might have accepted `Promise<LIO<...»` and focused on providing good combinators for working with promised monadic values.

4.3 Closure-Based Labels

The shift from tuple-based labeled values (`[L, V]`) to closure-based values (`{getLabel(), unsafeGetValue()}`) is presented as part of our async extension, but it is not actually related to asynchronous operations. The problem with tuple destructuring, exists equally in synchronous code. Nothing about promises or async I/O makes this worse beyond a psychological effect: developers debugging async code naturally want to `await` and inspect results, making destructuring more tempting.

This means we are claiming credit for solving a problem that was already present in the original ifc-ts. The closure-based approach is an improvement to labeled values in general, not specific to async support. One could implement closures in synchronous ifc-ts with identical benefits and costs.

Why conflate these changes? The practical reason is that async code makes the destructuring problem more visible. In synchronous ifc-ts examples, developers rarely extract values from the monad; they compose operations and only call `unsafe_runLIO` at the boundary. Async code invites more intermediate inspection because debugging asynchronous systems requires examining promise states and resolved values. Closures prevent a mistake that async workflows make more likely, even if the mistake was always possible.

4.4 The Synchronous-Asynchronous Divide

By introducing `bindAsync` alongside the original `bind`, we have bifurcated the monad's API. Developers must now understand which operations are synchronous and which are asynchronous, and use the corresponding bind function. This is explicit and unsurprising in TypeScript, where async/await makes the distinction visible, but it complicates the mental model.

In Haskell's LIO, there is one `bind` operation (written `>>=`). The IO monad handles both synchronous operations (like pure computation) and asynchronous operations (like network requests) uniformly. The distinction is internal to the IO system, not exposed in the API. Why can't we do this in TypeScript?

The answer involves TypeScript's type system limitations. Haskell's type system can express "this returns an IO action that produces an A, regardless of whether that A comes from a pure computation or an async operation." TypeScript's type system can only express "this returns a value V" or "this returns a Promise<V>," and these are incompatible. We cannot write a single `bind` that handles both `LIO<Lpc, L, V>` and `LIO<Lpc, L, Promise<W>>` without weakening the asynchronous nature of the promised version.

So we chose explicitness: two functions, clear types, obvious at the call site. But this explicitness has a cost. Code that mixes synchronous and asynchronous operations must use different bind functions, breaking the uniformity of monadic composition. The monad does not abstract over sync vs async; it exposes the difference and forces developers to manage it.

This issue stems from both a design limitation on our side and an inherent constraint of the TypeScript type system. TypeScript's type system lacks the expressiveness to hide this distinction, but there might have been other ways to provide a unified interface. For example, we could have deprecated synchronous operations entirely and made *everything* promise-based, using `Promise.resolve` to wrap synchronous values. This would eliminate the bifurcation at the cost of wrapping all operations in promises, even when unnecessary. We chose not to do this, prioritizing performance and explicitness over API uniformity. Whether this was the

right choice depends on whether one values consistency or efficiency more highly.

4.5 What Have We Actually Accomplished?

Stepping back, we have demonstrated that TypeScript's type system is expressive enough to enforce IFC across asynchronous boundaries. Promises can be integrated into a library-based IFC system without losing compile-time guarantees. This is not trivial. It requires careful design to ensure labels are not erased when promises resolve, and to chain asynchronous operations while propagating security constraints.

The question for evaluation is not whether our system is perfect, but whether it is useful. The answer depends on the threat model and development context. If preventing accidental data leaks through type-checked code is worth the ergonomic cost of monadic programming, then yes. If developers can tolerate the limitations of static labels and the inability to address covert channels, then yes. But for general-purpose application development, where productivity and maintainability outweigh formal security guarantees, probably not.

This is a niche solution for a niche problem. That does not make it unimportant. Security-critical applications exist, and they need tools like this. We are providing a tool for developers who already accept the costs of library-based IFC and need async support. The contribution is incremental, not revolutionary, and its value lies in demonstrating feasibility rather than transforming practice.

5 Conclusion

This work demonstrates that TypeScript's type system is expressive enough to enforce compile-time IFC across asynchronous boundaries. By integrating promises into the LIO monad as `LIO<Lpc, L, Promise<Labeled<L, V>>>` and introducing `bindAsync` for chaining asynchronous operations, we extended ifc-ts to handle async I/O while preserving noninterference guarantees. Closure-based labeled values prevent accidental label erasure through destructuring, making security boundaries explicit. The result is a library-based IFC system that works with standard TypeScript tooling, requires no runtime overhead for security checking, and integrates with promise-based APIs used throughout the JavaScript ecosystem.

The contribution is incremental but meaningful. We show that async and IFC can coexist in a mainstream typed language without compiler modifications or specialized build systems. For security-critical TypeScript applications requiring async I/O, this extension makes library-based IFC practically viable.

5.1 Future Work

Several technical questions remain open, offering directions for strengthening this approach.

Unifying synchronous and asynchronous bind. Our design splits `bind` into synchronous and asynchronous variants, forcing developers to track which operations return promises. We attempted a unified `bind` accepting both `LIO<Lpc, L, V>` and `LIO<Lpc, L, Promise<W>>` using the union of those return types, but this broke type inference for promise chaining. Whether TypeScript's conditional types,

higher-kinded type encodings, or other type-level techniques can achieve true unification remains an open question. If possible, it would eliminate a significant source of API complexity.

Formalizing promise interactions with the security lattice. The original ifc-ts lacks a formal semantics, and our extension inherits this limitation. Formalizing how promises interact with the security lattice would provide stronger guarantees about noninterference preservation. Madsen et al. [17] provide formal semantics for JavaScript promises; adapting their model to include security labels and proving that `bindAsync` preserves noninterference would place this work on firmer theoretical ground. Such formalization could also reveal subtle information leaks through promise resolution order or exception propagation that our informal reasoning misses.

Promise-like interface for the monad. If the `LIO` type were a record rather than a tuple, we could add a `.then` method that invokes `bindAsync` internally. This would let developers write `computation.then(x => ...)` instead of `bindAsync(computation, x => ...)`, matching the promise-chaining style familiar to TypeScript developers. Keeping the functional `bindAsync` (possibly renamed to `then`) alongside the method-based API would let programmers choose their preferred style. Whether this improves ergonomics enough to justify the added complexity is worth exploring. It might bridge the gap between monadic abstraction and idiomatic JavaScript patterns.

Empirical validation at scale. Our examples demonstrate basic functionality but do not stress-test the system under realistic async concurrency. Non-blocking sinks might introduce race conditions where interleaved writes leak information through observable ordering. Event loop scheduling could allow callback interleaving that violates noninterference in ways not captured by the type system. Systematic testing with adversarial schedulers, fuzzing tools, or model checking could uncover such violations. Applying the extended ifc-ts to real security-critical codebases would measure whether the ergonomic costs are tolerable in practice and whether developers can correctly use the API without introducing security bugs.

References

[1] Dorothy E. Denning and Peter J. Denning. “Certification of Programs for Secure Information Flow”. In: *Communications of the ACM* 20.7 (1977), pp. 504–513.

[2] Dorothy E. Denning. “A Lattice Model of Secure Information Flow”. In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[3] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. “A Sound Type System for Secure Flow Analysis”. In: *Journal of Computer Security* 4.3 (1996), pp. 167–187.

[4] Andrei Sabelfeld and Andrew C. Myers. “Language-Based Information-Flow Security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19.

[5] Andrew C. Myers. “JFlow: Practical Mostly-Static Information Flow Control”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1999, pp. 228–241.

[6] Vincent Simonet. “A Type System for Secure Information Flow”. In: *Proceedings of the 12th European Symposium on Programming (ESOP)*. Springer, 2003, pp. 147–161.

[7] Deian Stefan et al. “Addressing Covert Termination Channels in Concurrent Information Flow Systems”. In: *Proceedings of the 21st USENIX Security Symposium*. USENIX Association, 2012, pp. 19–35.

[8] Willard Rafnsson. *IFC-TS: Information Flow Control for TypeScript*. 2023. URL: https://github.com/willardthor/ifc-ts (visited on 11/01/2024).

[9] Sandip Ghosal and R. K. Shyamasundar. “Information Flow Security Certification for SPARK Programs”. In: *Data and Applications Security and Privacy XXXIV*. Ed. by Anoop Singhal and Jaideep Vaidya. Cham: Springer International Publishing, 2020, pp. 137–150.

[10] Daniel Hedin and Andrei Sabelfeld. “Information-Flow Security for a Core of JavaScript”. In: *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, June 2012, pp. 3–18.

[11] TypeScript Team. *TypeScript: JavaScript With Syntax for Types*. https://www.typescriptlang.org/. Accessed: 2025-12-09.

[12] TypeScript Team. *Conditional Types*. https://www.typescriptlang.org/docs/handbook/2/conditional-types.html. Accessed: 2025-12-09.

[13] TypeScript Team. *Mapped Types*. https://www.typescriptlang.org/docs/handbook/2/mapped-types.html. Accessed: 2025-12-09.

[14] TypeScript Team. *Advanced Types*. https://github.com/Microsoft/TypeScript-Handbook/blob/master/pages/Advanced%20Types.md. Accessed: 2025-12-09.

[15] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. “HLIO: Mixing static and dynamic typing for information-flow control in Haskell”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015, pp. 289–301.

[16] Kyle Simpson. *You Don't Know JS: Async & Performance*. O'Reilly Media, 2015.

[17] Magnus Madsen, Ondřej Lhoták, and Frank Tip. "A Model for Reasoning about JavaScript Promises". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), 86:1–86:24. DOI: 10.1145/3133910.