Cocoon: Static Information Flow Control in Rust

ADA LAMBA, Ohio State University, USA
MAX TAYLOR, Ohio State University, USA
VINCENT BEARDSLEY, Ohio State University, USA
JACOB BAMBECK, Ohio State University, USA
MICHAEL D. BOND, Ohio State University, USA
ZHIQIANG LIN, Ohio State University, USA

Information flow control (IFC) provides confidentiality by enforcing noninterference, which ensures that high-secrecy values cannot affect low-secrecy values. Prior work introduces fine-grained IFC approaches that modify the programming language and use nonstandard compilation tools, impose run-time overhead, or report false secrecy leaks—all of which hinder adoption.

This paper presents Cocoon, a Rust library for static type-based IFC that uses the unmodified Rust language and compiler. The key insight of Cocoon lies in leveraging Rust's type system and procedural macros to establish an effect system that enforces noninterference. A performance evaluation shows that using Cocoon increases compile time but has no impact on application performance. To demonstrate Cocoon's utility, we retrofitted two popular Rust programs, the Spotify TUI client and Mozilla's Servo browser engine, to use Cocoon to enforce limited confidentiality policies.

CCS Concepts: • **Security and privacy** → **Information flow control**; • **Software and its engineering** → **Access protection**.

Additional Key Words and Phrases: information flow control, type and effect systems, Rust

ACM Reference Format:
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, OOPSLA1, Article 100 (April 2024), 28 pages. https://doi.org/10.1145/3649817

1 INTRODUCTION

Confidentiality is a fundamental property of secure computing systems. Confidentiality, or secrecy, is usually achieved through *access control*, which regulates *which* entities may access *which* data, but *not* what happens to the data after it is accessed, resulting in a large trusted computing base. As a robust alternative to access control, *information flow control (IFC)* ensures that high-secrecy values cannot flow to (i.e., cannot affect) low-secrecy values [Denning 1976; Rotenberg 1973].[1]

IFC is especially valuable as a *fine-grained* mechanism that protects the flow of data values within applications, ensuring they do not misuse privileges. Consider, for example, a virus checker

[1]IFC generally ensures not only confidentiality but also integrity, by ensuring that low-integrity values cannot flow to high-integrity values, but this paper handles only confidentiality.

Authors' addresses: Ada Lamba, Ohio State University, Columbus, USA, lamba.39@osu.edu; Max Taylor, Ohio State University, Columbus, USA, taylor.2751@osu.edu; Vincent Beardsley, Ohio State University, Columbus, USA, beardsley.49@osu.edu; Jacob Bambeck, Ohio State University, Columbus, USA, bambeck.14@osu.edu; Michael D. Bond, Ohio State University, Columbus, USA, mikebond@cse.ohio-state.edu; Zhiqiang Lin, Ohio State University, Columbus, USA, zlin@cse.ohio-state.edu.

ACM 2475-1421/2024/4-ART100
https://doi.org/10.1145/3649817

application with privileges to both read data from the user's disk and send data to a remote server (to request software updates). However, the virus checker should *not* be allowed to send the contents of the user's disk to the remote server—a confidentiality policy that fine-grained IFC can enforce. As a result, the virus checker does not need to be trusted by the user. In general, fine-grained IFC reduces the trusted computing base from the entire application to the relatively few operations that explicitly violate IFC by *declassifying* secret values.

Fine-grained IFC can be supported either dynamically at run time, which incurs run-time costs [Austin and Flanagan 2009; Roy et al. 2009; Xiang and Chong 2021], or statically at compile time. Static IFC can use whole-program analysis or modular type-based analysis. *Whole-program static analysis* conservatively models the flow of values through expressions and variables and ensures that high-secrecy values do not flow to low-secrecy sinks [Balasubramanian et al. 2017; Hammer et al. 2006; Hedin and Sabelfeld 2012; Zanioli et al. 2012]. However, whole-program analysis is non-compositional, so it scales poorly and supports incremental deployment poorly. A recent analysis called *Flowistry* leverages Rust ownership constraints to compute relatively precise function summaries without analyzing the functions' bodies [Crichton et al. 2022]. In contrast, *static type-based analysis* uses types to represent the security labels of variables and expressions, and a correctly typed program guarantees IFC [Chapman and Hilton 2004; Chong 2008; Gregersen et al. 2019; Kozyri et al. 2016; Myers 1999; Myers and Liskov 1997; Myers et al. 2006; Russo 2015; Sabelfeld and Myers 2003; Simonet 2003; Smith and Volpano 1998; Volpano et al. 1996]. While type-based analysis allows for modular checking and incremental deployment, existing solutions rely on a *modified* language and compiler,[2] impeding adoption.

Why must prior work on type-based analysis *modify* the language and compiler? Because the language's type system is not expressive enough to ensure noninterference statically. The key technical challenge is how to allow programs to define and use functions on secret data without permitting unchecked side effects that might violate noninterference. In theory, an *effect system* for encoding the side effects of a computation would constrain the behaviors of functions on secret data [Nielson and Nielson 1999]. While effect systems have been implemented in *Haskell* for type-based IFC [Abadi et al. 1999; Gregersen et al. 2019; Russo 2015], effect systems are not supported by mainstream imperative languages including Java, C, C++, and Rust.

Contributions

This paper introduces *Cocoon*, the first static type-based IFC for a mainstream imperative language. Cocoon is a Rust library for the unmodified Rust language that works with the standard Rust compiler (although Cocoon relies on some Rust features that are not yet part of stable Rust). Cocoon ensures that a program compiles only if it does not leak secrets, except by explicitly *declassifying* secrets or explicitly using unsafe Rust code. Cocoon provides a rudimentary effect system for noninterference that limits computations' side effects using a novel approach leveraging Rust features including ownership, traits, and procedural macros.

We evaluated Cocoon's effectiveness and efficiency by retrofitting two popular Rust applications, a Spotify client and the Servo web browser engine, to enforce a confidentiality policy for a single secret value in each program. These case studies demonstrate that Cocoon's approach works in real applications and can be incrementally deployed. The evaluation also shows Cocoon does not affect application run time or memory usage, although it can affect compile time if a substantial fraction of the program code handles secret values.

Cocoon thus advances the state of the art in fine-grained IFC support by demonstrating a novel approach that requires no language or compiler modifications. The current design has

[2]An exception is type-based IFC analysis in Haskell (§6).

several limitations, including supporting only static secrecy labels and placing restrictions on the programming model (§ 4.5).

This paper makes the following contributions:

- a fine-grained IFC approach that is the first to provide static type-based IFC for a mainstream imperative language without language or compiler modifications;
- novel support for a rudimentary effect system that captures side effects that could leak secrets;
- two case studies on real Rust programs that demonstrate that Cocoon works for real programs and adds negligible or nonexistent run-time and compile-time overheads; and
- a performance evaluation showing that Cocoon has no detectable run-time impact, and its impact on compile time is commensurate with how much code that deals with secret values.

2 MOTIVATING EXAMPLE

Fig. 1 shows example Rust code that computes the number of overlapping days of availability on two calendars, belonging to Alice and Bob. Each calendar is a map from a day of the week, represented as a string, to a boolean indicating whether Alice or Bob is available on that day. Suppose that Alice and Bob's calendars are secret. Specifically, the values (booleans) in their calendars are secrets, although the keys (strings) are not. For simplicity we assume that the calendars have identical key sets. The example computes and reveals only the number of overlapping days in Alice and Bob's availability.

The syntax `for (day, available) in alice_cal` iterates over all key–value pairs in Alice's calendar. Note that the types of `day` and `available` are `&String` and `&bool`, meaning that they are actually (immutable) *references* to the values. The expression `bob_cal.get(&day)` calls `HashMap::get(&self, &String) -> Option<&bool>`, which is a method that takes a reference to a `String` as a parameter. Rust allows an unlimited number of immutable references, *or* a single mutable reference, to a value at a time. The `get` method returns a value of type `Option<&bool>`, which represents a reference to a boolean, or an error if the key was not found in `bob_cal`. The call to `Option::unwrap(self) -> &bool` returns the reference to the inner boolean, or fails otherwise. Since `unwrap()` returns type `&bool` (i.e., reference to a boolean), the code uses the `*` operator to dereference (and implicitly copy) `unwrap()`'s return value.

To ensure that the code in Fig. 1 does not leak more information about Alice or Bob's schedule than the number of overlapping days, one must trust the code or audit it carefully. Alternatively, fine-grained IFC provides a way to *ensure* that code cannot leak secrets to untrusted entities, by disallowing high-secrecy values from flowing to low-secrecy values.

Information flows can be explicit or implicit. In an *explicit flow*, a data dependency exists between the two values. In Fig. 1 the return value of `bob_cal.get(&day)` flows to `Option::unwrap()`'s parameter via an explicit flow. An *implicit flow* is a flow that involves control dependence. In Fig. 1, an implicit flow exists from `available`'s value (and from the return value of `bob_cal.get(&day)`) to `count`'s value.

Consider Fig. 2, which has the same functionality as Fig. 1, but it ensures IFC using this paper's approach, Cocoon. The booleans in Alice and Bob's calendars are wrapped in a `Secret` type and have labels $\{a\}$ and $\{b\}$, where a and b are *policies* (also called *tags*) corresponding to secrecy privileges of Alice and Bob, respectively. A label L_1 is *at least as secret as* label L_2 if and only if $L_1 \supseteq L_2$.

Applications can read and write secret values only inside of lexically scoped regions called *secret blocks*, denoted by the `secret_block!`[3] macro call. Before the loop, the code uses a secret block to create a new secret value `0` (zero) with label $\{a, b\}$, which is assigned to `count`. The secret block

[3]In Rust, all macro names end with the `!` character.

```
let alice_cal: HashMap<String, bool> = /* { "Monday" -> true, ... } */
let bob_cal:   HashMap<String, bool> = /* { "Monday" -> false, ...} */
let mut count = 0;
for (day, available) in alice_cal {
    if available && *bob_cal.get(&day).unwrap() {
        count += 1;
    }
}
println!("Overlapping days: {}", count);
```

Fig. 1. Rust code that computes the overlapping number of days in two calendars.

```
let alice_cal: HashMap<String, Secret<lat::A,bool>> = /* { "Monday" -> Secret(true), ... } */
let bob_cal:   HashMap<String, Secret<lat::B,bool>> = /* { "Monday" -> Secret(false), ...} */
let mut count = secret_block!(lat::AB { wrap_secret(0) });
for (day, available) in alice_cal {
    secret_block!(lat::AB {
        if unwrap_secret(available) &&
           *unwrap_secret_ref(::std::option::Option::unwrap(
               ::std::collections::HashMap::get(&bob_cal, &day))) {
            *unwrap_secret_mut_ref(&mut count) += 1;
        }
    });
}
println!("Overlapping days: {}", count.declassify());
```

Fig. 2. Rust code that has the same functionality as Fig. 1 but uses Cocoon to ensure IFC.

inside the loop[4] has label $\{a, b\}$; Cocoon ensures that the block's inputs must be no more secret than $\{a, b\}$, and its output(s) must be no less secret than $\{a, b\}$, thus preventing illegal flows.

The block uses calls to `unwrap_secret`, `unwrap_secret_ref`, and `unwrap_secret_mut_ref` to access secret values. Cocoon ensures that these calls are only allowed if the unwrapped value's label is no more secret than the secret block's label. `unwrap_secret_mut_ref()` gets *mutable* access to a secret value, allowing both read and write access, requiring the value to have the *same* label as the secret block.

The secret block uses fully qualified calls to Rust Standard Library functions (e.g., `::std::option::Option::unwrap(...)`), which Cocoon requires in order to check that the callee cannot leak secrets.

After the loop completes, the program explicitly declassifies the secret value `count` in order to print its value. Declassifications are part of the trusted computing base and must be audited.

3 REQUIREMENTS AND ASSUMPTIONS

Fine-grained information flow control (IFC) is not used widely in practice because it is not practical. We argue that a practical solution must provide IFC for programs written in a mainstream imperative language without modifying the language or its compiler.

3.1 Assumptions and Nonrequirements

This paper focuses on achieving fine-grained IFC for static secrecy labels using an off-the-shelf language and compiler. This work does *not* aim to support integrity labels, dynamic labels, or OS

[4]One could instead wrap the whole loop in a single secret block, but in the spirit of minimizing code in secret blocks—and to showcase more of Cocoon's programming model—the code wraps only the loop's body in a secret block.

integration.[5] It does not protect side channels such as termination and timing channels. Finally, although the programming model is pure Rust, it has several restrictions, a challenge we leave for future work. §4.4 and §4.5 describe Cocoon's guarantees and limitations in more detail.

Our work aims to provide *termination-insensitive noninterference* (explained below), but we do not present a proof of correctness, which would be a notable research contribution by itself. Other work has presented correctness proofs, but only for small, toy languages [Myers et al. 2004; Surbatovich et al. 2023; Zdancewic and Myers 2000; Zheng and Myers 2005]. As Zheng and Myers point out, "[p]roofs for noninterference exist for numerous security-typed languages, but not for any language as expressive as Jif" [Zheng and Myers 2005]. The authors of Viaduct (a compiler for secure distributed programs) note that "a full correctness proof for the Viaduct compiler would be a significant research achievement …" [Acay et al. 2021].

3.2 Requirements

Here we motivate and describe the requirements that our IFC approach needs to meet.

Widely Used, High-Performance Language. A drawback of many prior static IFC solutions is that they require the use of an academic language that inhibits their adoption (§6). Thus, we aim to develop an IFC solution that uses a popular, high-performance language: *Rust*. Rust was designed by Graydon Hoare at Mozilla Research in 2010 and had its first stable release in 2015 [Mozilla Research 2020; Noel 2010]. Since then it has had 95 releases, over 3,500 contributors, and has held the title of Stack Overflow's "most loved language" by developers every year since 2016 [Stack Overflow 2022]. Rust is commonly used in large-scale applications; for example, many core features of Mozilla Firefox are primarily written in Rust [The Rust Foundation 2022].

Off-the-Shelf Language and Compiler Support. In contrast to prior solutions that extend a language and modify the compiler or rely on a custom compiler (§6), we aim to enforce IFC without requiring a customized language or compilation tools. In the context of the Rust language, we aim to provide IFC entirely through a library. Programs are written in unmodified Rust using the IFC library and compiled with the unmodified Rust compiler, although our design relies on some language features that are not yet part of stable Rust. The library is trusted (i.e., is part of the trusted codebase) along with the Rust compiler and standard library. The application is untrusted, except for `unsafe` and declassify operations, which must be trusted or audited. In Rust, `unsafe` operations already forgo guarantees of memory and type safety.

Soundness. We aim to provide *termination-insensitive noninterference*: In the absence of explicit declassification or explicitly `unsafe` code, high-secrecy values cannot affect low-secrecy values but may affect whether the program terminates [Abadi et al. 1999; Askarov et al. 2008]. This model handles explicit and implicit flows, but it does not address side channels created by timing or microarchitectural states.

Incremental Deployment. A goal is for the library to be incrementally deployable: Developers only need to modify parts of their code that deal with secret values. These modifications may still be time consuming and burdensome, a challenge we leave for future work.

Run-Time Performance. Programs using the IFC library should perform indistinguishably from otherwise equivalent programs that do not. The Rust compiler should be able to "optimize away"

[5] Since OS integration is outside our scope, the programming model assumes that every OS-level entity is an untrusted entity. It is the programmer's responsibility to identify secrets coming from OS-level entities such as files and sockets, and the program must declassify a secret value before sending it to any OS-level entity.

the typing and scaffolding introduced by the application's use of the IFC library, reducing the compiled code so it performs the same as an otherwise equivalent program.

4 COCOON: STATIC INFORMATION FLOW CONTROL IN RUST

Cocoon is a static IFC approach that meets §3.2's requirements. §4.1 explains how application developers use Cocoon to ensure noninterference. §4.2 describes how Cocoon ensures that the program will not compile unless it enforces noninterference, and §4.3 gives implementation details. §4.4 and §4.5 discuss guarantees, threats, and limitations.

4.1 Cocoon's Programming Model

An application uses Cocoon's programming model to represent secret values and secrecy labels, to access secret values, and to declassify secret values. Fig. 3 overviews the programming model, and Fig. 2 (introduced in § 2) shows example application code that uses most elements of the programming model.

Secret Values and Secrecy Labels. Our work follows the decentralized IFC model introduced by Myers and Liskov [1997]. A secrecy *label*, which describes the permissions for data, is a set of secrecy *policies* (also called *tags*). For example, a label $L = \{a, b\}$ contains the policies a and b. These policies correspond to entities with different secrecy privileges, such as users of a system. Labels form a partial order on these policies, in which label L_1 is *at least as secret as* label L_2 if and only if $L_1 \supseteq L_2$.

Every expression and variable has a static secrecy label determined by its static type. By default, expressions and variables have the empty secrecy label (lowest secrecy). An expression or variable with higher secrecy is represented by wrapping it in a Cocoon-provided type `Secret<Type,Label>`, presented in Fig. 3a. `Secret` values are opaque and must be accessed through an interface provided by Cocoon, discussed shortly. Note that wrapping values in `Secret` incurs no run-time cost; a `Secret<Type,Label>` instance has the same run-time representation as a `Type` instance.

Fig. 3b shows that labels are implemented as structs, and trait implementations[6] help enforce legal flows. Specifically, a trait `MoreSecretThan<M>` is implemented for label L if and only if $M \subseteq L$.

In Fig. 2, `alice_cal` and `bob_cal` are hash maps whose values are secret boolean values with labels $\{a\}$ and $\{b\}$, respectively.

Accessing Secrets in Secret Blocks. To help restrict illegal flows and to limit the annotation burden, `Secret`-wrapped values may be accessed only inside of lexically scoped blocks of code called *secret blocks*, as shown in Fig. 3c. A secret block is a macro call that specifies a static label that restricts the values the block's body may read and write—the body may only read values with lower or same secrecy, and may only write values with higher or same secrecy—prohibiting illegal explicit and implicit flows. The block evaluates to a secret value with the the same label as the block. Fig. 2 has two secret blocks (line 3 and lines 5–11), each with secrecy label $\{a, b\}$.

The secret block's body can call `unwrap_secret(`e`)` on an expression e with type `Secret<Type,`*label*`>`, as long as *label* is no more secret than the block's secrecy label. Cocoon also provides variants of `unwrap_secret` that take and return references instead of values, called `unwrap_secret_ref` and `unwrap_secret_mut_ref`. (§ 2 explained immutable and mutable references briefly.) Fig. 2's second secret block (lines 5–11) calls `unwrap_secret` and its variants to access secret values with labels $\{a\}$, $\{b\}$, and $\{a, b\}$—all of which are no more secret than the block's label $\{a, b\}$.

Note that code *outside* of a secret block cannot call `unwrap_secret()` or its variants because they are *not* defined functions—they are recognized only by `secret_block!`, as described in §4.2.1.

[6] Rust's traits are analogous to Java's interfaces and C++'s abstract methods, but traits allow more expressive type constraints.

```
struct Secret<Type: SecretValueSafe, SecrecyLabel: Label>
```

(a) Secret expressions and variables with non-empty secrecy have type `Secret<T,L>`. The trait `SecretValueSafe`, defined in §4.2.2, restricts what values can be wrapped in `Secret`.

Struct	Traits implementing struct
`Label_Empty`	`Label`, `MoreSecretThan<Label_Empty>`
`Label_A`	`Label`, `MoreSecretThan<Label_Empty>`, `MoreSecretThan<Label_A>`
`Label_B`	`Label`, `MoreSecretThan<Label_Empty>`, `MoreSecretThan<Label_B>`
`Label_AB`	`Label`, `MoreSecretThan<Label_Empty>`, `MoreSecretThan<Label_A>`, `MoreSecretThan<Label_B>`, `MoreSecretThan<Label_AB>`

(b) All possible labels for two policies, a and b, and traits defining a partial order of labels

```
secret_block!(label { body })
```

(c) Secret block, which executes *body* with secrecy label *label*

Operation	Evaluates to	Type constraint
`unwrap_secret(v: Secret<T,L>) -> T`	Inner value	$L \subseteq L_B$
`unwrap_secret_ref(v: &Secret<T,L>) -> &T`	Immutable ref. to inner value	$L \subseteq L_B$
`unwrap_secret_mut_ref(v: &mut Secret<T,L>) -> &mut T`	Mutable ref. to inner value	$L = L_B$
`wrap_secret(v)`	`Secret<_,`L_B`>::new(v)`	N/A

(d) Operations on `Secret` values in a secret block with secrecy label L_B

```
#[side_effect_free_attr]
fn func(params) -> retType { body }
```

(e) Annotation for side-effect-free functions. Secret blocks may (transitively) call only side-effect-free functions.

```
#[derive(InvisibleSideEffectFree)]
struct s { ... }
```

(f) Annotation for application types that are used in secret blocks

Operation	Evaluates to
`Secret<T,L>::declassify(self) -> T`	Inner value
`Secret<T,L>::declassify_ref(&self) -> &T`	Immutable reference to inner value
`Secret<T,L>::declassify_ref_mut(&mut self) -> &mut T`	Mutable reference to inner value

(g) Declassify operations on `Secret` values

Fig. 3. Cocoon's programming model.

A secret block calls `wrap_secret(e)` to create a new `Secret` value with the same secrecy label as the block. Fig. 2's first secret block (line 3) evaluates to a secret value created by a call to `wrap_secret`. The second secret block (lines 5–11) has no return value.[7]

A secret block can call application-defined functions only if they have been annotated as *side effect free* using a Cocoon-provided attribute macro, `#[side_effect_free_attr]` (Fig. 3e). Side-effect-free functions in turn can only call other side-effect-free functions.

To use a custom type in secret blocks, an application must annotate the type as side effect free by annotating it with `#[derive(InvisibleSideEffectFree)]` (Fig. 3f).

The running example in Fig. 2 does not define any side-effect-free functions or types, but Figs. 12–14 in §5.3 do.

Declassification. Cocoon allows high-secrecy values to flow to low-secrecy values through the `declassify()` method and its variants (Fig. 3g), which return the unwrapped value. Line 13 in Fig. 2 declassifies the secret value of `count`. Declassification is a trusted operation and should be audited by developers.

4.2 How Cocoon Ensures Noninterference

To ensure noninterference, Cocoon restricts the behavior of secret blocks. Given a secret block `secret_block!(`*L*`,` *body*`)`, Cocoon restricts the expression *body* in the following ways:

- *body* must conform to IFC rules, e.g., it cannot read a value with higher secrecy than the block. §4.2.1 describes how Cocoon enforces IFC rules.
- *body* must not have side effects such as sending data to a socket or writing to a variable visible outside of the block. §4.2.2 and §4.2.3 describe how Cocoon enforces side effect freedom, essentially providing a rudimentary effect system.

To enforce these restrictions, Cocoon leverages Rust *procedural macros*, which are procedures that operate on compiled code at compile time. Type information is not available at macro expansion time, and so the procedural macro operates on an untyped abstract syntax tree (AST).

Throughout §4.2.1–§4.2.3, we describe and refer to Figs. 4 and 5, which provide details of the procedural macros' transformations.

4.2.1 Restricting the Flow of Secret Values. Cocoon enforces IFC rules on the flow of secret values using transformations by the `secret_block!` procedural macro.

`secret_block!` expands `unwrap_secret(e)` to a call to a function that requires *e*'s label to be no more secret than the block's label *L* (similarly for `unwrap_secret_ref(e)`). For `unwrap_secret_mut_ref(e)`, the type constraint requires *e*'s label to be *equal to* *L* because mutable references allow both writing and reading the referent. Fig. 5a shows details of these transformations, by defining the function $\tau(e, isExec)$ for when *e* is an `unwrap_secret` (or `wrap_secret`) expression. In general, the text uses $\tau(e, isExec)$ to define compile-time transformations performed by the macros on expression *e* (the *isExec* parameter is explained later).

To ensure that a secret block returns a `Secret` with the same label as the block, the macro transformation of `secret_block!(`*L*` { body })` ensures that *body* returns a `Secret` value with label *L*. Fig. 4a shows how this works: The macros transform the secret block to a call to `Cocoon::call_closure()`, which must have type `Secret<T,`*L*`>` (or a tuple of secrets with label *L*). Note that the macros generate both *executed* and *nonexecuted* versions of the code (hence the `if true {...} else {...}` in Fig. 4a) for reasons explained in §4.2.3.

[7]Technically, the block's body's return type is the empty type, `()`, which implements `SecretTrait<L>`, as required by the secret block (Fig. 4a).

Application code:

```
secret_block!(L, body)
```

Macro-transformed code:

```
if true {
  ::Cocoon::call_closure::<L,_,_>(|| -> _ {
    ::std::panic::catch_unwind(::std::panic::AssertUnwindSafe(|| {
      τ(body, F)
    }))
  }.unwrap_or_default())
} else {
  ::Cocoon::call_closure::<L,_,_>(|| -> _ { τ(body, T) })
}
```

Definition of `call_closure`:

```
fn call_closure<L, F, R>(clos: F) -> R
  where F: FnOnce() -> R + VisibleSideEffectFree, R: SecretTrait<L>
  { clos() }
```

Note: `SecretTrait<L>` is a trait implemented by `Secret<T,L>` and tuples of `Secret<T,L>`, allowing secret blocks to return multiple secret values.

(a) How the `secret_block!` macro expands secret blocks. As the text explains, the macro generates both executed (`if true`) and nonexecuted (`else`) versions of the block's body.

Application code:

```
#[side_effect_free_attr]
fn f(params) -> r { body }
```

Macro-transformed code:

```
fn __f_secret_trampoline_unchecked(params) -> r { τ(body, T) }
fn __f_secret_trampoline_checked(params) -> r { τ(body, F) }
#[inline(always)]
unsafe fn f(params) -> ::Cocoon::Vetted<r> {
    ::Cocoon::Vetted::<r>::wrap(__f_secret_trampoline_unchecked(args))
}
```

(b) How the `#[side_effect_free_attr]` macro expands a side-effect-free function f. As the text explains, the macro generates both executed (`__f_secret_trampoline_unchecked`) and nonexecuted (`__f_secret_trampoline_checked`) versions of f.

Fig. 4. Outer transformations performed by Cocoon's procedural macros on secret blocks and side-effect-free functions. $\tau(expr, isExec)$ generates transformed code for expression $expr$ at macro expansion time, where $isExec$ is a boolean indicating whether the transformation is for the executed or nonexecuted version of generated code. Fig. 5 details how $\tau(expr, isExec)$ is defined for various kinds of expressions.

expr	τ(*expr*, *isExec*)
`unwrap_secret(`e`)`	`{ let tmp =` $\tau(e, isExec)$`; unsafe { Secret::unwrap::<`L`>(tmp) } }`
`unwrap_secret_ref(`e`)`	`{ let tmp =` $\tau(e, isExec)$`; unsafe { Secret::unwrap_ref::<`L`>(tmp) } }`
`unwrap_secret_mut_ref(`e`)`	`{ let tmp =` $\tau(e, isExec)$`; unsafe { Secret::unwrap_mut_ref::<`L`>(tmp) } }`
`wrap_secret(`e`)`	`{ let tmp =` $\tau(e, isExec)$`; unsafe { Secret::new::<_,`L`>() } }`

(a) Transformations performed by Cocoon's procedural macros on secret blocks only (i.e., not on side-effect-free functions). L is the secret block's label. `Secret` is an abbreviation for `::cocoon::Secret`.

expr	τ(*expr*, *isExec*)
`callee(`e_1`, `e_2`, ...)` if `callee` is *not* allowlisted	`let tmp1 =` $\tau(e_1, isExec)$`; let tmp2 =` $\tau(e_2, isExec)$`; ... unsafe { callee(tmp1, tmp2, ...) as ::cocoon::Vetted<_> .unwrap() }`
`callee(`e_1`, `e_2`, ...)` if `callee` is allowlisted	`check_ISEF(callee(`$\tau(e_1, isExec)$`, `$\tau(e_2, isExec)$`, ...))`

(b) Transformations performed by Cocoon's procedural macros on calls.

expr	τ(*expr*, F)	τ(*expr*, T)
`literal` (constant)	`literal`	`literal`
`path` (var. name)	`{ let tmp = &path; unsafe { check_ISEF_unsafe(tmp) } }`	`path`
e_1 `+` e_2	`::Cocoon::SafeAdd::safe_add(`$\tau(e_1, F)$`, `$\tau(e_2, F)$`)`	$\tau(e_1, T)$ `+` $\tau(e_2, T)$
e_1 `&&` e_2	$\tau(e_1, F)$ `&&` $\tau(e_2, F)$	$\tau(e_1, T)$ `&&` $\tau(e_2, T)$
e_1 `=` e_2	`*check_not_mut_secret(&mut` $\tau(e_1, F)$`) =` $\tau(e_2, F)$	$\tau(e_1, T)$ `=` $\tau(e_2, T)$
`&`e	`check_ISEF(&(`$\tau(e, F)$ `))`	`&`$\tau(e, T)$
`*`e	`*(`$\tau(e, F)$ `)`	`*`$\tau(e, T)$
e`.field`	$\tau(e, F)$`.field`	$\tau(e, T)$`.field`
`if` e_1 `{` e_2 `}` `else {` e_3 `}`	`if` $\tau(e_1, F)$ `{` $\tau(e_2, F)$ `} else {` $\tau(e_3, F)$ `}`	`if` $\tau(e_1, T)$ `{` $\tau(e_2, T)$ `} else {` $\tau(e_3, T)$ `}`

(c) Transformations performed by Cocoon's procedural macros on secret blocks and side-effect-free functions.

Fig. 5. Expression-level transformations performed by Cocoon's procedural macros on secret blocks and side-effect-free functions. $\tau(expr, F)$ and $\tau(expr, T)$ represent syntactic expansions for the nonexecuted and executed code paths, respectively. `check_ISEF` stands for "check invisible side effect free."

4.2.2 Restricting Syntactically Visible Side Effects. To ensure that a secret block's body has no side effects that could violate noninterference, Cocoon's procedural macros perform transformations on the body. Here we describe how the macros handle side effects that are *syntactically visible* to the macros, and §4.2.3 describes how the macros handle side effects that are *syntactically invisible* to the macros.

Restricting Mutation-Based Side Effects. To ensure that a secret block cannot perform writes visible outside of the block—except for writes to mutable `Secret` values, which are allowed—Cocoon leverages Rust's ownership and mutability constraints. Cocoon prevents a secret block from modifying non-`Secret` variables in its surrounding environment by forbidding the secret block's body, which is implemented as a closure, from capturing non-`Secret` variables by mutable reference. Cocoon introduces the trait `VisibleSideEffectFree` (Fig. 6) to denote that the block does not capture variables in its environment by mutable reference. (In Rust, a closure implements a trait only if all the variables it captures implement that trait.) Cocoon leverages Rust's auto traits and negative implementation features to define `VisibleSideEffectFree` so that every type automatically implements it except for types that (transitively) contain non-`Secret` mutable references (`&mut T` except for

Trait	Description	Definition
`VisibleSideEffectFree`	Limits mutably captured values of secret blocks	(`Immutable` ∨ (`&mut Secret<_,_>`)) ∧ `&InvisibleSideEffectFree`
`SecretValueSafe`	Restricts `T` in `Secret<T,L>` to immutable, block-safe types	`Immutable` ∧ `InvisibleSideEffectFree`
`Immutable`	Types without interior or regular mutability	¬(`UnsafeCell<_>`) ∧ ¬(`&mut _`)
`InvisibleSideEffectFree`	Types that can be used in secret blocks	Implemented individually for built-in and application types

Fig. 6. Traits used by Cocoon to enforce programming model restrictions to ensure side effect freedom.

`&mut Secret<_,_>`) or interior-mutable cells (types that implement `UnsafeCell`, e.g., `RefCell<T>`).[8] As a result, a closure that implements `VisibleSideEffectFree` cannot mutably capture non-`Secret` values. The only possible side effect of mutation is through `Secret` values, which Cocoon mediates.

To prevent a secret block from assigning to a mutable `Secret` variable without going through `unwrap_secret_mut_ref` (potentially leaking a more-secret value through an implicit flow), `secret_block!` transforms every assignment expression e_1 `=` e_2 by wrapping e_1 in a call to Cocoon function `check_not_mut_secret()`, as Fig. 5c shows. The call requires statically that e_1's type is (transitively) not `Secret`, by using Rust auto traits and negative implementations.

Cocoon must limit mutation in secret blocks, but not in functions called (transitively) by secret blocks: In Rust, functions (unlike closures) cannot mutate memory accessible outside of the function except through parameters to the function.

Cocoon must also ensure that values of type `T` wrapped in a `Secret<T, Label>` are not interior mutable to avoid, for example, a secret block that reads high-secrecy data using `unwrap_secret()` instead of `unwrap_secret_mut()` to modify an interior-mutable value with low secrecy. Cocoon disallows interior-mutable types in a `Secret` by requiring that `T` implement a Cocoon-defined trait `SecretValueSafe` (Fig. 6) that disallows mutable types.

Prohibiting Calls to Functions with Side Effects. To ensure side effect freedom, Cocoon must ensure that every function called (transitively) by secret blocks is side effect free. Applications can annotate side-effect-free functions with the `#[side_effect_free_attr]` macro (introduced in §4.1 and Fig. 3e), which causes the function's body to be transformed to ensure side effect freedom, as Fig. 4b shows. However, it is not straightfowrard to ensure that every function called from a *side-effect-free context* (i.e., a secret block or side-effect-free function) is side effect free, because type information is not available at macro expansion time.

To address this challenge, the procedural macro `#[side_effect_free_attr]` transforms the return type of the annotated function from `R` to `Vetted<R>`, a Cocoon-provided type. Cocoon's procedural macros also rewrite all function calls within side-effect-free contexts from `f(...)` to `(f(...) as Vetted<R>).value`, mandating that `f` return `Vetted<R>`. As a result, every callee function must have been marked side effect free. Fig. 5b shows how the macros transform calls in side-effect-free contexts.

Note that unverified functions cannot get around this requirement by directly returning `Vetted<R>`. First, a function cannot simply construct a `Vetted` value and return that function's return value because the constructor of `Vetted` is marked as `unsafe` to prevent unverified code from creating `Vetted` return values. Second, a function cannot call another function that has been transformed by

[8] Attempting to construct a custom "unsafe cell" type would require `unsafe` (and would be undefined behavior), thus forgoing Cocoon's guarantees.

the `#[side_effect_free_attr]` macro to obtain a `Vetted` value, because the macro marks each function it generates as `unsafe` to disallow calling the vetted function directly (without using `unsafe`).

Since the `#[side_effect_free_attr]` macro generates a function marked `unsafe`, the untrusted function body could contain expressions that are only permitted in unsafe Rust, although the original function was not marked `unsafe`. The `#[side_effect_free_attr]` macro prevents this by generating a second function that contains the function body. This second function omits the `unsafe` modifier (assuming the original function did the same), thus preventing safety violations. Fig. 4b shows how the the macro transforms an annotated function.

In addition to allowing calls to functions marked side effect free in side-effect-free contexts, the procedural macro allows calls to Rust Standard Library functions that the Cocoon developers (i.e., we) have "allowlisted" as being side effect free. Cocoon's procedural macro includes a list of functions that we have determined to be side effect free. Because the procedural macro operates on tokens and has no type information, calls to allowlisted library functions from side-effect-free contexts must be fully qualified (e.g., `::std::string::String::len()`) to prohibit calls to same-named functions with side effects. Allowlisting of library functions adds to the trusted codebase and could be fragile (§4.4).

Thus, applications can mark any function as side effect free, and the compiler will check that the function is side effect free and that side-effect-free contexts call only side-effect-free functions. In order for a secret block to have a side effect, e.g., by sending data to a socket, it must (transitively) call a function *not* marked side effect free, which cannot pass the procedural macros' checks.

4.2.3 Restricting Syntactically Invisible Side Effects. So far we have described how Cocoon prohibits side effects that are visible to procedural macros. However, Rust provides a few syntactically *invisible* ways to call functions—through overloaded operators, deref coercion, and custom destructors—which are impossible for procedural macros to detect.

To prohibit syntactically invisible side effects in side-effect-free contexts, Cocoon's procedural macros transform every expression in a way that ensures that it cannot compile if it has a syntactically invisible side effect. The procedural macros actually generate *two versions* of code in side-effect-free contexts: one that executes and one that does not. In the *executed* version, overloadable operators and expressions are *not* transformed by the procedural macro (but `unwrap_secret` and `wrap_secret` calls are still expanded). This version is the code that actually executes. In the *nonexecuted* version, the procedural macro *does* transform operators and expressions as described below, but the generated code is unreachable. As such, the expanded version causes compiler errors if Cocoon's requirements are violated, but it never executes. Fig. 4 shows the transformations performed on secret blocks and side-effect-free functions to generate the executed and nonexecuted versions. The executed code and nonexecuted code are generated by $\tau(expr, isExec)$, which Fig. 5 provides details of.

Cocoon generates executed and nonexecuted versions for two reasons. First, transforming a *path expression* (i.e., use of a variable) requires violating borrowing rules: The transformation inserts a call to `check_ISEF_unsafe`, which needs to take a reference to the path but return the expression, as Fig. 5c's `path` row shows. It is generally unsafe to allow this generated code to execute; generating it in the nonexecuted version means it only affects whether the code compiles. Second, avoiding most transformations in the executed version may reduce compile- and run-time costs of the transformations.

Restricting Overloaded Operators. Rust allows applications to overload certain operators (e.g., `+` and `<`) by defining custom functions for their behavior. For example, if an application overloads the `+` operator for certain types, the expression `a + b` might no longer be side effect free, depending on the types of `a` and `b`. Cocoon's procedural macro cannot detect whether `a + b` invokes an overloaded

operator because it operates on a parse tree without type information. To address this problem, Cocoon's procedural macro replaces all calls to *overloadable* operators with a call that implements the default operator behavior.

Specifically, Cocoon provides a series of internal traits (e.g., `cocoon::SafeAdd` to replace `std::ops::Add`) that are essentially copies of the standard Rust traits for each overloadable operator. Cocoon implements these traits for all applicable standard Rust types. Cocoon's procedural macros replace e_1 `+` e_2 with `::Cocoon::SafeAdd::safe_add(`e_1`,` e_2`)` in side-effect-free contexts. In this way, Cocoon allows the use of overloadable operators only on standard Rust types, for which the behavior is the original, Rust-standard side-effect-free behavior. Any use of these operators on application-defined types within a side-effect-free context results in a compilation error. Fig. 5c shows this transformation for e_1 `+` e_2; the transformation is analogous for other overloadable operators.

Code cannot circumvent this restriction by overloading operators on built-in Rust types (e.g., numeric types): Overloadable operators are defined by built-in Rust traits, and Rust does not permit a crate to implement a trait for a type unless the crate defines the trait or the type (or both).

Restricting Deref Coercion and Custom Destructors. One of Rust's features is that, in attempting to find a match for a function or struct signature, the Rust compiler performs *deref coercion*, automatically inserting dereference calls until it finds a match for the signature or cannot dereference any further. For example, the compiler automatically converts `s.area()` to `s.deref().area()` if `s` has type `&Shape` and method `Shape::area()` exists. Likewise, the compiler converts `b.area()` to `b.deref().deref().area()` if `b` has type `&Box<Shape>` and method `Shape::area()` exists. This situation presents a challenge because a call to `s.area()` can implicitly invoke `s.deref()` or `s.deref().deref()` in a way that is invisible to Cocoon's procedural macro. While `Deref::deref()` and `DerefMut::deref_mut()` have no side effects for built-in types, an application could define a `deref()` or `deref_mut()` method that has side effects.

Another problematic Rust feature is that applications can define custom destructor behavior. The compiler inserts a *drop* (i.e., deallocation) of a value in the code where the value dies. The application can provide custom behavior on drop by implementing the `Drop` trait on an application type and providing an implementation of `Drop::drop()`—which could have a syntactically invisible side effect.

To handle these operations, the procedural macros insert code to ensure that *every value used by or created in* a side-effect-free context does not provide a custom implementation of `Deref`, `DerefMut`, or `Drop`. Cocoon provides a trait `InvisibleSideEffectFree` that can be implemented by any type that (1) does not provide a custom implementation of `Deref::deref()`, `DerefMut::deref_mut()`, or `Drop::drop()` and (2) contains only `InvisibleSideEffectFree` values (Fig. 6). Cocoon implements `InvisibleSideEffectFree` for conforming Standard Rust types. For example, Cocoon implements `InvisibleSideEffectFree` for `i64`, for `Box<T: InvisibleSideEffectFree>`, and for many other types. Conforming application types do not automatically implement `InvisibleSideEffectFree`, but an application can annotate a type definition with `#[derive(InvisibleSideEffectFree)]` (Fig. 3f), which invokes a procedural macro that transforms the type's definition to ensure that (1) it does not implement `Deref`, `DerefMut`, or `Drop` (by generating negative implementations of these traits) and (2) all of the type's contained values have types that implement `InvisibleSideEffectFree` (by generating type constraints on the contained values).

To ensure that side-effect-free contexts only use and create `InvisibleSideEffectFree` values, Cocoon's procedural macro in general transforms every expression e to `check_ISEF(`e`)` as Figs. 5b and 5c show. The expression `check_ISEF(`e`)` simply returns the value e evaluates to, but e's type must implement `InvisibleSideEffectFree` to compile.

Expansion of `secret_block!(lat::AB { wrap_secret(0) })`:

```
if true {
    /* Expansion by τ(e, T) */
    ::cocoon::call_closure::<lat::AB, _, _>(
        (|| -> _ {
            let result = ::std::panic::catch_unwind(::std::panic::AssertUnwindSafe(|| {
                /* Expansion of wrap_secret(0) */
                let tmp = 0; unsafe { ::cocoon::Secret::<_, lat::AB>::new(tmp) }
            })).unwrap_or_default();
            result
        }))
} else {
    /* Expansion by τ(e, F) */
    ::cocoon::call_closure::<lat::AB, _, _>(
        (|| -> _ {
          /* Expansion of wrap_secret(0) */
          unsafe { ::cocoon::Secret::<_, lat::AB>::new(0) }
        }))
}
```

Fig. 7. The secret block at line 3 of Fig. 2, after expansion by Cocoon's procedural macro `secret_block!`.

Not every kind of expression needs to be wrapped in `check_ISEF()`, as Fig. 5c shows. For example, the field expression `e.field` does not need to be wrapped because it is `InvisibleSideEffectFree` as long as `e` is `InvisibleSideEffectFree` (`e` still needs to be wrapped in `check_ISEF()` in general).

Note that secret blocks are not prevented from *capturing* non-`InvisibleSideEffectFree` types.[9] However, it is sufficient for macro transformations to prevent secret blocks from *creating* or *using* such types.

Handling Exceptional Flow. To avoid illegal implicit flows, a secret block must have a single static exit. Note that even if the block's body executes `return`, `continue`, or `break`, control always continues immediately after the block, since a secret block's body is wrapped in a closure (Fig. 4a).

A remaining problem are *panics*, which Rust generates for unexpected or unrecoverable errors. By default, a panic aborts the execution, creating a termination channel if a secret block (or code it calls) panics. Worse yet, the application might *catch* a panic outside of a secret block, creating an implicit flow. To avoid this, every secret block catches any panics within it (using `std::panic::catch_unwind`). The secret block handles a caught panic by returning a `Secret`-wrapped default value (`Secret`-wrapped types must implement `std::default::Default`) and continuing execution after the block normally, as Fig. 4a shows.

4.2.4 Example Transformation. Figs. 7 and 8 show how the `secret_block!` procedural macro transforms Fig. 2's first and second secret blocks, respectively.

[9]Specifically, `VisibleSideEffectFree`, which defines types that can be captured by secret blocks, does *not* negatively implement all `InvisibleSideEffectFree` types (because this does not seem possible due to limitations of auto traits and negative implementations). Instead, `VisibleSideEffectFree` excludes implementation of all `&T` where `T` does *not* implement `InvisibleSideEffectFree` (Fig. 6), which disallows captures of non-`InvisibleSideEffectFree` variables by reference, but not by value.

```
if true {
    /* Expansion by τ(e, T) */
    ::cocoon::call_closure::<lat::AB, _, _>(
        (|| -> _ {
            let result = ::std::panic::catch_unwind(::std::panic::AssertUnwindSafe(|| {
                if /* Expansion of unwrap_secret(available) */
                   { let tmp = available;
                   unsafe { ::cocoon::Secret::unwrap::<lat::AB>(tmp) } } &&
                   /* Expansion of unwrap_secret_ref(::std::option::Option::unwrap(
                   ::std::collections::HashMap::get(&bob_cal, &day))) */
                   *{ let tmp = ::std::option::Option::unwrap(
                                    ::std::collections::HashMap::get(&bob_cal, &day));
                      unsafe { ::cocoon::Secret::unwrap_ref::<lat::AB>(tmp) } }
                {
                  /* Expansion of *unwrap_secret_mut_ref(&mut count) += 1; */
                  *{ let tmp = &mut count;
                     unsafe { ::cocoon::Secret::unwrap_mut_ref::<lat::AB>(tmp) } } += 1;
                }
            }))
            .unwrap_or_default();
            result
        })
    )
} else {
    /* Expansion by τ(e, F) */
    ::cocoon::call_closure::<lat::AB, _, _>(
      || -> _ {
          if /* Expansion of unwrap_secret(available) */
             unsafe {
               ::cocoon::Secret::unwrap::<lat::AB>(
                 { let tmp = &(available); unsafe { ::cocoon::check_ISEF_unsafe(tmp) } }) } &&
             /* Expansion of unwrap_secret_ref(::std::option::Option::unwrap(
             *unsafe {
              ::cocoon::Secret::unwrap_ref::<lat::AB>({
                ::cocoon::check_ISEF(::std::option::Option::unwrap({
                  ::cocoon::check_ISEF(::std::collections::HashMap::get(
                      { ::cocoon::check_ISEF_ref(&bob_cal) },
                      { ::cocoon::check_ISEF_ref(&day) })) })) }) }
          {
            /* Expansion of *unwrap_secret_mut_ref(&mut count) += 1; */
            ::cocoon::SafeAddAssign::safe_add_assign(
                &mut *(unsafe { ::cocoon::Secret::unwrap_mut_ref::<lat::AB>(
                                  { ::cocoon::check_ISEF((&mut count)) })
                }), 1);
          }
      })
}
```

Fig. 8. The secret block at lines 5–11 of Fig. 2, after expansion by Cocoon's procedural macro `secret_block!`.

4.3 Implementation Details

Our prototype implementation of Cocoon is divided into two Rust crates—one that defines the procedural macros and another that provides all other Cocoon functionality—since procedural macros must be defined in their own crate. The implementation depends on a *nightly* version of Rust (1.69.0-nightly) since it uses a few Rust features that are not yet stable: auto traits, negative traits, function traits, and unboxed closures. The implementation's source code is publicly available.[10]

4.4 Security Guarantees and Threats

Cocoon's general guarantee (which we have not proved) is termination-insensitive noninterference in the absence of declassify operations and unsafe Rust code (§3.2). Here we discuss a few caveats of Cocoon's guarantees.

Trusted Codebase. Since we have not proved that allowlisted Rust Standard Library functions are side effect free, allowlisted functions must be trusted to be side effect free (just like Cocoon's codebase must be trusted). Identifying library functions as side effect free can be tricky. For example, `Vec::sort` should not be allowlisted because it calls an implementation of `Ord::cmp` passed as a parameter, which could be an application-defined function with side effects. As another, hypothetical example, a function with a side-effect-free specification might have side effects internally (and perhaps only in a future implementation of the library).

In general, every component of an application (whether part of the application's crate or an external crate) must be ported to Cocoon, or it must be explicitly trusted by declassifying secret data before sending it to the component. Porting components to Cocoon provides an incremental mechanism for reducing the trusted codebase.

Programmers must audit not only any declassify operations or unsafe Rust code in their applications, but also the application's `Cargo.toml` file, which specifies the names and locations of external crates including Cocoon and the Rust Standard Library.

Macros. What if application code calls an application macro that transforms a secret block (e.g., `evil_macro!(secret_block!(...))`)? Fortunately, the Rust compiler's macro expansion algorithm iteratively expands macros, starting from the outermost macro. The resulting expansion is repeatedly expanded, until no macros are left in the expansion. If an application macro expands to contain a call to Cocoon's macro, Cocoon's macro's expansion is opaque to the application macro.

Cocoon requires all accesses to secret values to occur in macros—except for calls to `declassify()` methods, which could be transformmed by an application macro as described above. However, this security hole is mitigated by the fact that `declassify()` calls must already be audited. Alternatively, one could modify Cocoon's programming model so that declassification uses a macro instead of a method.

4.5 Limitations

Cocoon's design has some drawbacks that limit its practicality as a fine-grained IFC approach. It may be possible to remove these limitations in future work.

Cocoon's design supports only static secrecy labels, not integrity labels or dynamic labels (i.e., labels as run-time values).

The current design does not allow the use of overloaded operators in secret blocks. Likewise, the current design disallows application types that implement custom dereference (`Deref` and `DerefMut`) and destructor (`Drop`) operations. We believe that this is not very restrictive because these traits

[10] https://github.com/PLaSSticity/Cocoon-implementation

are most useful for implementing abstract data types (ADTs). In fact, the Rust documentation[11] emphasizes that `Deref` should only be implemented by programmers when defining smart pointers. As a result, only a few "kernel" ADTs are needed for most software.

Secret blocks cannot use macros. Due to the Rust compiler's macro expansion algorithm, which expands the outermost macro first and iteratively works its way through the resulting expansion (§4.4), Cocoon cannot inspect a macro's expansion within a secret block and thus cannot certify that it is side effect free. It may be possible to extend Cocoon's macro to recognize known macros such as `vec!` and generate code using the appropriate macro expansion, but we have not implemented this.

Secret blocks and side-effect-free functions cannot use interior mutability. In our experience, interior mutability is common, but is not typically necessary: In the evaluated programs, we were able to refactor the code in all side-effect-free code to not use interior mutability.

Calls from side-effect-free code to allowlisted Rust Standard Library functions must be fully qualified (§4.2.2). As an alternative that we have not implemented, Cocoon's procedural macros could rewrite every non-fully qualified call that *syntactically appears to be* a call to a Rust Standard Library function, to a fully qualified call. For example, `s.trim()` would be rewritten as `::std::string::String::trim()`. To avoid ambiguity of function names, Cocoon could disallow side-effect-free application functions from having the same name as any allowlisted library function.

5 EVALUATION

This section evaluates the usability and performance of Cocoon. We integrated Cocoon with two applications (Spotify TUI and Servo) and also wrote Cocoon and non-Cocoon versions of a Battleship program. These case studies show the usability of Cocoon: Cocoon can be used by real applications to enforce security policies, with modifications commensurate with the amount of code that deals with secret values. A performance evaluation shows no detectable impact on executable size or run-time performance, and modest compile-time overheads for real applications.

All experiments were executed on a quiet machine with a 16-core Intel Xeon Gold 5218 at 2.3 GHz with 187 GB RAM running Linux.

5.1 Case Study: Spotify TUI

Spotify TUI (text-based user interface) is a Spotify client written in Rust that allows an end user to interact with Spotify from a terminal. We chose it as a case study because it is a popular project on GitHub[12] with over 13,000 "stars."

We modified Spotify TUI to use Cocoon to protect a value called the *client secret*, which is akin to a user password. The client secret is a 32-digit hexadecimal string that Spotify TUI uses to authenticate with the Spotify API server. Leaking the client secret to an adversary would allow the adversary to make Spotify API calls on behalf of the victim user.

Spotify TUI receives the client secret from the command line, or from a file that a previous execution of Spotify TUI wrote the client secret to. We modified these code locations to wrap the client secret as a `Secret<String,Label_A>`. (For this application we use a simple label scheme in which $\{a\}$ and $\emptyset$ are the only labels, representing secret and non-secret values, respectively.) Spotify TUI supports sending the client secret outside of the application in two ways: (1) writing the client secret to a file; and (2) sending the client secret to two external Rust libraries, RSpotify (a Rust wrapper for the Spotify Web API) and Serde (a Rust serialization library).

By retrofitting the Spotify TUI application to use Cocoon, we implicitly treat Spotify TUI as untrusted (except for its declassify calls and any `unsafe` blocks). Spotify TUI must declassify the

[11] https://doc.rust-lang.org/1.23.0/std/ops/trait.Deref.html
[12] https://github.com/Rigellute/spotify-tui

```rust
fn validate_client_key(key: &str) -> Result<()> {
    if key.len() != EXPECTED_LEN {
        Err(Error::from(std::io::Error::new(
            std::io::ErrorKind::InvalidInput,
            format!("invalid length: {} (must be {})", key.len(), EXPECTED_LEN,),
        )))
    } else if !key.chars().all(|c| c.is_digit(16)) {
        Err(Error::from(std::io::Error::new(
            std::io::ErrorKind::InvalidInput,
            "invalid character found (must be hex digits)",
        )))
    } else {
        Ok(())
    }
}
```

Fig. 9. Spotify TUI code that validates that the client secret is a 32-digit hexadecimal string.

```rust
fn validate_client_key(key: &Secret<String, Label_A>) -> Result<()> {
  const EXPECTED_LEN: usize = 32;
  let sec_error_string = secret_block!(Label_A {
    let u_key = unwrap_secret_ref(key);
    let mut is_hex = true;
    for c in ::str::chars(&u_key) {
      if !::char::is_digit(c, 16) {
        is_hex = false;
      }
    }
    let mut error_string = ::std::string::String::from("");
    if ::std::string::String::len(&u_key) != EXPECTED_LEN {
      error_string = /* create string "invalid length: {key.len()} (must be {EXPECTED_LEN})" */;

    } else if !is_hex {
      error_string = ::std::string::String::from("invalid character found (must be hex digits)");
    }
    wrap_secret(error_string)
  });

  let error_string = sec_error_string.declassify();
  if !error_string.is_empty() {
    Err(::std::error::Error::from(::std::io::Error::new(
      ::std::io::ErrorKind::InvalidInput,
      error_string,
    )))
  } else {
    Ok(())
  }
}
```

Fig. 10. Modified version of Fig. 9's that uses Cocoon.

client secret in order to write it to a file and to send it to RSpotify and Serde (because we have not integrated these components with Cocoon), implicitly treating the OS, RSpotify, and Serde as part of the trusted codebase. Removing RSpotify and Serde from the trusted codebase would require retrofitting them with Cocoon by annotating their side-effect-free functions (§4.4).

As an example, Fig. 9 shows a Spotify TUI function that validates that the client secret is 32 hexadecimal digits, returning a `Result` with value `Ok` or `Err`. Fig. 10 shows how we changed the code to account for the client secret being a `Secret` value.

To perform the validation logic, the modified code creates a secret block to check that each digit within the client secret is a hexadecimal character. Note that we had to manually implement the

Table 1. Programmer burden for integrating three applications with Cocoon, in terms of lines of code modified, and number of calls to declassify methods.

	Lines of source code				
Program	Original	Cocoon	Inserted	Deleted	Declassify calls
Spotify TUI	12,169	12,282	167	54	17
Servo	397,141	397,174	77	44	1
Battleship	383	410	47	20	2

Table 2. Comparison of performance costs for applications without Cocoon and versions that use Cocoon. Execution times include 95% confidence intervals based on multiple trials.

	Compile time (s)		Run time		Executable size (bytes)	
Program	Original	Cocoon	Original	Cocoon	Original	Cocoon
Spotify TUI	14.395 ± 0.031	14.395 ± 0.053	2868.711 ± 5.612 ns	2822.784 ± 5.662 ns	15,779,320	15,779,088
Servo	312.579 ± 4.354	309.983 ± 4.262	183.817 ± 0.540 s	184.086 ± 0.529 s	403,548,416	403,547,984
Battleship	2.068 ± 0.003	2.321 ± 0.002	30.240 ± 0.254 ms	29.490 ± 0.272 ms	6,727,464	6,740,248

string formatting and hexadecimal character check, as opposed to using `|c| c.isdigit(16)`, since the Cocoon implementation does not support closure calls within secret blocks.

The secret block populates an error string, which it leaves empty in the event of no error, and the retrofitted code declassifies the error string because, in the event of an error, it must report the error. Declassifying the error string is acceptable leakage because knowing *whether* the client secret is correctly formatted is not security sensitive. The retrofitted code in Fig. 10 ultimately returns a `Result`, just as in Fig. 9. Thus, Fig. 10 achieves the same behavior for the client secret without declassifying the secret—but rather by declassifying only whether the input client secret is not in the correct format, which is not exploitable information.

Empirical Results. Table 1 shows that the annotation burden of protecting the client secret with Cocoon is small: To retrofit Spotify TUI, we inserted 167 lines of code and removed 54 lines in a codebase of over 12,000 lines.

Table 1's last column shows how many times a call to a `Secret::declassify` method appears in the retrofitted code of Spotify TUI. In 14 out of 17 cases, the code declassifies secret data because the data needs to be sent outside of the Spotify TUI codebase. In these cases, the client secret leaves the codebase to be passed to RSpotify or Serde, or to be written to the configuration file, all as intended. In the other 3 out of 17 cases, the code declassifies values that do not consitute a secrecy leak. For example, revealing the `Result` in Fig. 10 does not present a security risk.

Table 2 shows the performance cost of retrofitting Spotify TUI to use Cocoon. We compiled Spotify TUI 10 times, without and with integration with Cocoon, using the "release" optimization level. According to the results, Cocoon has no detectable impact on compile time. Cocoon has a modest impact on the size of the compiled binary; in fact, Cocoon's code size is actually smaller than the original's by 232 bytes, for reasons that are unclear to us. In theory, the compiled code should be identical, but the additional code generated by Cocoon's macros could affect compiler decisions such as how aggressively to inline.

To measure run time, we timed just the part of the code—configuration and authentication—that deals with the client secret, in both the original and retrofitted Spotify TUI. As the results show, the measured time is around 3 μs. We ran and timed this part of the code 100,000 times for each version of Spotify TUI to account for high run-to-run variability. The Cocoon version actually outperforms the original version by about 1%. We are unsure of the reason for this (statistically significant and

```
1  pub struct Response {
2  -  pub body: Arc<Mutex<ResponseBody>>,
3  +  pub body: Arc<Mutex<Secret<ResponseBody,Label_A>>,
4     ...
5  }
6
7  pub fn get_body(&self) -> Arc<Mutex<ResponseBody>> {
8      if self.response_type == ResponseType::Opaque {
9          Arc::new(Mutex::new(ResponseBody::Empty))
10     } else {
11         self.body.declassify_transmute()
12     }
13 }
14
```

Fig. 11. Servo's response structure. We modified the structure to use Cocoon to enforce the same-origin policy.

repeatable) performance difference. Like the executable code size differences, the performance difference could be the result of compiler decisions impacted by the extra code generated by Cocoon's macros.

5.2 Case Study: Servo

Servo is Mozilla's browser engine written in Rust [Linux Foundation 2022]. We selected Servo as an evaluation subject for two reasons. First, Servo is the sixth-most popular Rust project on GitHub, with over 20,000 "stars." Second, Servo is an influential project, e.g., Rust's early design was informed by Mozilla's experiences creating Servo. We modified Servo to use Cocoon to help enforce a security policy that prevents different JavaScript programs from reading each others' user data.

To make an HTTP request, JavaScript programs call the Fetch API, which the JavaScript engine implements by calling into Servo's implementation to perform the request and return a response. Critically, JavaScript programs are only allowed to read HTTP responses if the programs share the same origin web server, unless the response explicitly allows cross-origin sharing. Otherwise a malicious JavaScript program such as a malicious advertisement could read sensitive user data such as a password field from a login form from a trusted JavaScript program originating from a different web server. This policy is called the *same-origin policy*. Responses from different origin servers are considered *opaque* and cannot be read by JavaScript programs.

Fig. 11 shows how we modified Servo to use Cocoon. First, we modified the `Response` type, which is instantiated after the client receives an HTTP response from an origin server following an HTTP request, to use Cocoon's `Secret` type to protect the response body from being read except by using declassification (lines 2 and 3).

Following this change, the application can read the response body only by declassifying it. To implement the same-origin policy, the code should only return the declassified value if the response type is not opaque, as shown in lines 8–12 of Fig. 11. If the response is opaque, the code returns an empty body (line 9), in accordance with the Fetch API specification. To return an `Arc<Mutex<ResponseBody>>`, the code calls a `declassify_transmute()` method (line 11) provided by the Cocoon implementation; it is like other `declassify` methods but returns a `T<Secret<U>>` value as a `T<U>`. Cocoon implements `declassify_transmute()` safely without creating new data because `T<Secret<U>>` is guaranteed to have the same layout as `T<U>`.

Instead of adding this same logic at the dozens of locations in Servo's source code that read `Response::body`, we changed all such locations to instead call `Response::get_body()` (line 7).

Following this change, an opaque response's body is not readable without a declassify operation (or an `unsafe` block). Our changes explicitly reflect developers' intentions and prevent future changes from accidentally allowing reads when the response body is opaque.

Empirical Results. Table 1 (page 19) shows the effort to integrate Servo with Cocoon to protect the response body. We modified dozens of lines of code to integrate Servo with Cocoon. A significant number of the lines we modified reside in Servo's test suite (55). Many edits were made by our IDE's search-and-replace feature. At other source code locations, we had to introduce temporary variables because of Rust's lifetime analysis: Normally, the value returned by `get_body` immediately dies (i.e., it is not usable by expressions later in the program). Storing the return value in a temporary variable keeps the value alive while the response body is read.

Table 2 compares performance before and after integration with Cocoon. As for Spotify TUI, we compiled Servo 10 times, with and without integration with Cocoon. We observe that there is no statistically significant compile-time overhead imposed.[13] We ran Servo's provided tests 20 times, with and without integration with Cocoon. We find that Cocoon imposes no run-time burden that is statistically significant (the confidence intervals overlap). The executable code sizes differ by 432 bytes; as for Spotify TUI, the exact reasons are unclear to us because of the substantial amount of code involved and the complexity of the compiler.

5.3 Case Study: Battleship

Battleship is a classic game in which two players place ships at secret locations on a grid. The object of the game is to sink the opponent's ships by correctly guessing their locations. A player wins by sinking all of the other player's ships first.

JRIF and Jif each employed Battleship as a case study [Kozyri et al. 2016; Myers et al. 2006]. It is interesting as a case study not only because it requires confidentiality, but also because secret data (ship placements) are revealed throughout the game. An implementation of Battleship should not reveal any of the locations of a player's ships unless the opponent correctly guesses it first. We wrote a multithreaded implementation of Battleship that uses Cocoon to enforce this secrecy policy.

Fig. 12 shows the data structure that represents each player's game state. Line 1 uses the `#[derive(InvisibleSideEffectFree)]` macro to assert that the data structure is safe to use from within side-effect-free contexts. The struct uses a generic secrecy label (line 2) so that the `Player` data structure can represent both players using distinct labels. Line 3 shows how the `ship_positions` field, which stores secret ship placement values, has `Secret` type. The player's guesses are low-secrecy values since they are announced to their opponent, and thus the value of `guesses` is not wrapped in `Secret`.

Fig. 13 shows the `Player` initialization routine, which preserves the confidentiality of `Player::ship_positions`. Line 4 begins the secret block that initializes a player's ship placements. The secret block iterates through the ships in the game and places them onto a grid. The secret block calls `random_placement`, which is annotated with `#[side_effect_free_attr]`, demonstrating Cocoon's effect system. Note that `random_placement` calls more application functions (`legal_placement` and `random_maybe_illegal_placement`) whose definitions are not shown.

The main game logic is shown in Fig. 14. The players communicate over the channel `chan`. This code prints Player A's guesses (public information), then prompts Player A to make a new guess (public information). The code then sends the guess to Player B over `chan`. Player B uses the same channel to communicate whether the guess hit a ship or ended the game. Next, Player A receives

[13] For Servo, compile time includes the time for compiling all dependent crates because Servo does not use the standard Rust build tools. For all of our other evaluated programs, compile time includes only the time to compile the target application.

```rust
#[derive(InvisibleSideEffectFree)]
struct Player<L: Label> {
    ship_positions: Secret<Grid<bool>, L>,
    guesses: Grid<CellStatus>,
}
```

Fig. 12. Data structure representing each Battleship player's game state.

```rust
impl<L: Label> Player<L> {
    fn new() -> Player<L> {

        let ship_positions = secret_block!(L {
            let ships = [Ship::Carrier, Ship::Battleship, Ship::Cruiser, Ship::Submarine, Ship::Destroyer];
            let mut ship_positions: Grid<bool> = [[false; GRID_SIZE]; GRID_SIZE];
            for ship in ships {
                let placement: Placement = random_placement(&ship_positions, &ship);
                place_ship(&mut ship_positions, &ship, &placement);
            }
            wrap_secret(ship_positions)
        });

        Player {
            ship_positions,
            guesses: [[CellStatus::Unguessed; GRID_SIZE]; GRID_SIZE],
        }
    }
}
#[side_effect_free_attr]
fn random_placement(grid: &Grid<bool>, ship: &Ship) -> Placement {
    let mut ship_placement: Placement = random_maybe_illegal_placement(grid, ship);
    while !legal_placement(&grid, ship_placement) {
        ship_placement = random_maybe_illegal_placement(grid, ship);
    }
    ship_placement
}
```

Fig. 13. Battleship player initialization routines.

```rust
fn game_loop_a(mut player: Player<lat::Label_A>, chan: session_types::Chan<(), PlayerA>) {
    let mut c = chan.enter();
    let mut player_b_guesses = [[CellStatus::Unguessed; GRID_SIZE]; GRID_SIZE];
    loop {
        // Not shown: Display user's previous guesses, prompt user to make new guess, send guess to opponent.

        // Receive opponent's guess.
        let (c3, guess) = c.recv();

        // Decide if the opponent hit our ship.
        let is_hit: bool = secret_block!(lat::Label_A {
            wrap_secret(is_occupied(unwrap_secret_ref(&player.ship_positions), guess.0, guess.1))
        }).declassify();

        // Not shown: Update opponent's guesses. Exit loop if guess won the game.
    }
}

#[side_effect_free_attr]
fn is_occupied(grid: &Grid<bool>, row: usize, col: usize) -> bool { (&grid[row])[col] }
```

Fig. 14. Main Battleship game loop.

Player B's guess (line 8). The game loop uses a secret block on line 11 to decide if Player B scored a hit. This is necessary because the player's ship placements are secret.

Empirical Results. Table 1 compares two versions of our Battleship implementation: one with Cocoon, and one without it. The Cocoon changes affect a significantly larger fraction of the code than for Spotify TUI and Servo. There are two calls to `declassify` in Battleship. Both occur in a player's game loop to decide if their opponent correctly guessed a ship location.

Table 2 shows the effect of Cocoon on compilation times, run times, and executable sizes, using the same methodology as for Spotify TUI and Servo. Cocoon slows compilation time by 12%—more than for Spotify TUI and Servo, which makes sense because a larger fraction of Battleship's code is in secret blocks and side-effect-free functions. Cocoon adds no run-time overhead—in fact, as for Spotify TUI, it decreases run time inexplicably, perhaps as a result of different compiler optimization decisions—showing that the compiler effectively optimizes away the type-checking scaffolding added by Cocoon. Finally, Cocoon increases executable size by 0.1%, for unknown reasons, possibly again due to compiler optimization decisions.

5.4 Performance-Focused Evaluation

To measure the worst-case performance impact of Cocoon, we evaluated wrapping entire programs in secret blocks. We used benchmarks from The Computer Language Benchmarks Game [Debian benchmarksgame-team 2022], for which developers create the most efficient implementations possible, in order to compare the performance of different programming languages. We evaluated 9 out of 10 available benchmarks [Debian benchmarksgame-team 2022], excluding reverse-complement because it concurrently modifies a shared mutable buffer, which constitutes an inherent side effect.

To wrap entire programs in secret blocks, we placed the functions that perform the benchmark's primary calculations inside of a secret block, and annotated the blocks' transitive callees as side-effect-free functions. We refactored all functions' code to adhere to Cocoon's programming model constraints. We directed Cocoon's procedural macros to ignore calls to functions in external crates, by wrapping them in `unsafe` blocks, which the procedural macro does not transform.

Table 3 shows results of each benchmark with and without our modifications to use Cocoon. We observe that there are few statistically significant differences in run time (wall clock time) between the original and Cocoon versions, and confidence intervals are tight enough that we can conclude with high confidence that Cocoon adds negligible (< 1%) or nonexistent overhead—and it speeds up programs as often as it slows them down, suggesting that statistically significant run-time differences are due to indirect effects such as compiler inlining decisions or microarchitectural sensitivities. (We also measured run-time memory usage, i.e., maximum resident set size, but omitted the results from the table for brevity. The results show that space overhead is negligible or nonexistent for all programs, i.e., confidence intervals are consistently small and overlapping.)

Cocoon increases compile time because its procedural macro adds significant "scaffolding" code for type checking, which is generally optimized away, so it does not translate to an increase in run time. The table shows that compile-time overhead ranges from 7% to 16% for all programs except n-body, which has 53% compile-time overhead. These overheads represent an extreme case in which all code is transformed by Cocoon's procedural macros. In practice, only a program's computations on secret values need to be transformed, which is why the compile-time overheads for Spotify TUI and Servo are significantly lower (Table 2).

6 RELATED WORK

In the mid-1970s, Lampson first discussed information confinement as preventing even the partial leak of confidential information [1973], while Rotenberg introduced a mechanism to constrain

Table 3. Performance of nine benchmarks whose entire computation is wrapped in a secret block, compared with the originals. Performance results include the mean and a 95% confidence interval from at least 10 trials.

	Run time (s)		Compile time (s)		Executable size (bytes)	
Benchmark	Original	Cocoon	Original	Cocoon	Original	Cocoon
binary-trees	0.364 ± 0.023	0.365 ± 0.023	0.608 ± 0.010	0.647 ± 0.003	4,560,608	4,560,640
fannkuch-redux	1.828 ± 0.027	1.821 ± 0.011	0.489 ± 0.147	0.545 ± 0.002	4,488,712	4,489,800
fasta	1.116 ± 0.070	1.093 ± 0.049	0.417 ± 0.008	0.468 ± 0.004	4,339,184	4,339,640
k-nucleotide	1.539 ± 0.007	1.559 ± 0.007	0.464 ± 0.003	0.503 ± 0.003	4,355,808	4,360,544
mandelbrot	0.352 ± 0.003	0.352 ± 0.003	0.603 ± 0.030	0.699 ± 0.006	4,546,848	4,548,456
n-body	46.657 ± 0.008	46.335 ± 0.005	0.386 ± 0.007	0.591 ± 0.007	4,305,872	4,318,312
pidigits	0.554 ± 0.005	0.556 ± 0.004	0.334 ± 0.004	0.382 ± 0.002	4,528,080	4,529,800
regex-redux	0.957 ± 0.003	0.967 ± 0.003	0.644 ± 0.005	0.686 ± 0.010	6,114,664	6,115,904
spectral-norm	0.111 ± 0.002	0.112 ± 0.003	0.426 ± 0.005	0.477 ± 0.005	4,514,568	4,515,176

systems from allowing secret data to flow to untrusted entities [1973]; these works became the concept of information flow control (IFC). Denning first defined a lattice structure for applying information flow policies [1976], and Denning and Denning and later observed that static analysis can be used for IFC [1977]. Building on the static analysis models of the 1970s, Myers and Liskov introduced a decentralized model in which applications declassify their own data as opposed to relying on a third-party authority to do so [1997].

Prior systems have provided *coarse-grained* IFC between processes, sockets, files, and other OS-level entities using dynamic tracking [Krohn et al. 2007; Zeldovich et al. 2008]. In contrast, *fine-grained* IFC at the level of variables and expressions presents unique challenges. First, communication within a process, which involves the flow of data and control between program-level elements such as values and expressions, is less explicit than the flow between processes and other OS-level entities, making it difficult to track information flow soundly and precisely. Second, fine-grained flow has higher bandwidth, so the cost of tracking it at run time can be substantial.

Existing fine-grained IFC operates either dynamically at run time or statically at compile time.

6.1 Dynamic IFC

Dynamic IFC tracks and checks labels of variables and values at run time. Austin and Flanagan showed how to speed up dynamic tracking of labels for JavaScript programs, but their approach still adds significant run-time overhead [2009]. Laminar tracks labels at run time for Java programs and handles implicit flows by consigning secret operations to lexically scoped regions, but still adds up to 56% run-time overhead [Roy et al. 2009]. Co-Inflow provides dynamic coarse-grained IFC for Java programs through the use of implicitly inserted labels [Xiang and Chong 2021]. Converting a Java program to use Co-Inflow requires few annotations and achieves relatively high precision, but introduces an average run-time overhead of 15%.

6.2 Static IFC

Static IFC can be provided by whole-program analysis or modular type-based analysis.

Whole-Program Static Analysis. Whole-program static analysis computes information flow conservatively by abstracting data and control flow across program expressions and variables [Balasubramanian et al. 2017; Hammer et al. 2006; Hedin and Sabelfeld 2012; Zanioli et al. 2012]. It minimizes programmer effort by eschewing the need for type annotations, but it is non-compositional: Precision loss analyzing one part of the program affects precision analyzing other parts of the program. Non-compositionality makes it hard to scale to large programs and hard to deploy incrementally.

Recent work introduces whole-program static analysis for information flow in Rust called Flowistry [Crichton et al. 2022]. Flowistry leverages Rust's ownership constraints to compute relatively precise function summaries without actually analyzing the functions' bodies. As such, Flowistry is scalable enough to use interactively, and it naturally handles calls to functions for which source code is unavailable. However, Flowistry is inherently *intraprocedural*; Crichton et al. suggest that future work "could build an interprocedural analysis by using Flowistry's output as procedure summaries in a larger information flow graph" [Crichton et al. 2022]. As an intraprocedural analysis, Flowistry is not applicable to checking for illegal flows in our evaluated programs, which have secret values that flow interprocedurally.

As a static analysis approach, Flowistry may report false flows, which are hard for programmers to deal with. In contrast, Cocoon uses static type-based analysis, which requires program modifications but provides a way to eliminate false illegal flows.

Flowistry and Cocoon are potentially complementary: Because Flowistry targets Rust and is fairly precise, it could potentially be adapted to infer some labels in applications using Cocoon.

Modular Static Type-Based Analysis. Static type-based approaches extend the type system with IFC labels, so the static type of every expression and variable includes an IFC label. In contrast with whole-program static analysis, static type-based analysis is compositional because typing rules can be checked locally (i.e., one expression at a time) instead of globally. As a result, type-based analysis offers the potential to allow large programs to provide IFC without false leak reports or run-time overhead. A downside of type-based analysis is that it creates extra work for programmers to express types, although this burden can be limited to parts of the program that touch data with non-empty labels. A key drawback of type-based analysis is that mainstream imperative languages such as Java, C, C++, C#, and Rust do not provide type systems that are expressive enough to ensure that operations on secret values do not have unchecked side effects. As a result, existing solutions either use a functional language [Gregersen et al. 2019; Russo 2015] or extend an imperative language [Chapman and Hilton 2004; Chong 2008; Kozyri et al. 2016; Myers 1999; Myers and Liskov 1997; Myers et al. 2006; Simonet 2003; Smith and Volpano 1998; Volpano et al. 1996], as described in the following paragraphs.

MAC is a static type-based IFC library for Haskell that leverages the fact that operations in pure functional languages are side effect free [Russo 2015]. Likewise, DepSec is a dependently typed static IFC library for Idris [Gregersen et al. 2019], a dependently typed language influenced by Haskell [Brady 2007]. High-performance programs are typically not written in these languages.

Flow Caml extends Objective Caml with a type system to track variable secrecy levels without requiring programmers to annotate the source code [Simonet 2003]. Chapman and Hilton similarly extended SPARK Ada and the SPARK Examiner to include variable annotations [2004]. This extension does not support concurrency, and their declassification mechanism requires hiding the body of the declassifying subprogram from the Examiner.

To add static type-based IFC to an imperative language, prior work extends the language and requires a modified or custom compiler. Jif is an extension of Java that provides static type-based IFC [Myers 1999; Myers and Liskov 1997; Myers et al. 2006]. It introduces types that represent secrecy and integrity labels, which programs use to annotate variables and expressions. Jif provides a compiler for the Jif language that checks types and flows, including implicit flows, and produces pure Java code with the same functionality as the original program. Jif includes a variety of features such as support for dynamic labels and automatic inference. Follow-on research has extended Jif to solve a wide variety of IFC-related problems, such as erasure policies and reactive information flow [Chong 2008; Kozyri et al. 2016]. The key drawback of Jif is that it requires programs to be

written in a nonstandard language and to be processed with a nonstandard compiler, hampering adoption.

Volpano, Smith, et al. developed a model for IFC type systems in imperative languages that they later extended to support multithreading and non-interference, but (as far as we know) this model has not been implemented [1998; 1996].

RLBox is an approach for sandboxing libraries to prevent cross-library leaks [Narayan et al. 2020]. It uses a combination of static type-based analysis and run-time isolation to prevent data and control from transferring between libraries. RLBox, which targets C++ libraries, cannot provide a guarantee of side effect freedom, but rather requires programmers to write a validation check when handling tainted values. RLBox adds run-time and space overheads, unlike Cocoon. Another difference is that Cocoon's secrecy labels allow a hierarchy of labels, while RLBox has only binary labels: A value is "tainted" or it is not.

In concurrent work, Curricle provides a type system for ensuring noninterference, in order to ensure region idempotence for intermittent programs [Surbatovich et al. 2023]. Like Cocoon, Curricle employs Rust procedural macros to ensure that only well-typed programs can compile. Unlike Cocoon, Curricle uses procedural macros to perform type analysis at the syntactic level; in contrast, Cocoon transforms the region's code so that the compiler performs the type analysis. As a result, Curricle requires idempotent regions to be functions, and it cannot support unbounded loops or recursion in idempotent functions. Furthermore, Curricle is unsound with respect to syntactically invisible side effects such as implicit destructor calls, dereference operations, and overloaded operator calls (§4.2.3) since it operates at the syntactic level. The paper proves noninterference with respect to a formal model, not the Rust language or the Curricle implementation [Surbatovich et al. 2023].

7 CONCLUSION

Cocoon is the first static type-based IFC approach for an imperative language that uses the standard language and compiler. The key technical contribution of Cocoon is its establishment of an effect system that tracks whether Rust functions have side effects. As a result, Cocoon allows programmers to define and perform arbitrary operations on secret data while ensuring that these operations do not violate IFC rules, all in pure Rust. Cocoon can be incrementally deployed on existing systems without significantly affecting performance.

DATA-AVAILABILITY STATEMENT

An artifact reproducing this paper's results is publicly available [Lamba et al. 2024].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable feedback, Ethar Qawasmeh for help in the early stages of the project, Chris Xiong for help with code and ideas, and Chujun Geng for helpful feedback and discussions. This work is supported by NSF grants CSR-2106117, XPS-1629126, and CNS-2207202.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 740–755. https://doi.org/10.1145/3453483.3454074

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security* (Málaga, Spain) *(ESORICS '08)*. Springer-Verlag, Berlin, Heidelberg, 333–348. https://doi.org/10.1007/978-3-540-88313-5_22

Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (Dublin, Ireland) *(PLAS '09)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1554339.1554353

Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *SIGOPS Oper. Syst. Rev.* 51, 1 (sep 2017), 94–99. https://doi.org/10.1145/3139645.3139660

Edwin Brady. 2007. Idris. Archived from http://www-fp.cs.st-and.ac.uk/ eb/darcs/Idris/.

Roderick Chapman and Adrian Hilton. 2004. Enforcing Security and Safety Models with an Information Flow Analysis Tool. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies* (Atlanta, Georgia, USA) *(SIGAda '04)*. Association for Computing Machinery, New York, NY, USA, 39–46. https://doi.org/10.1145/1032297.1032305

Stephen Nathaniel Chong. 2008. *Expressive and Enforceable Information Security Policies*. Ph. D. Dissertation. Cornell University. "http://people.seas.harvard.edu/~chong/pubs/chong_dissertation.pdf"

Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular Information Flow through Ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3519939.3523445

Debian benchmarksgame-team. 2022. The Computer Language 22.05 Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. Accessed 2 November 2022.

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (may 1976), 236–243. https://doi.org/10.1145/360051.360056

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (jul 1977), 504–513. https://doi.org/10.1145/359636.359712

Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov. 2019. A Dependently Typed Library for Static Information-Flow Control in Idris. In *Principles of Security and Trust*, Flemming Nielson and David Sands (Eds.). Springer International Publishing, Prague, Czech Republic, 51–75. https://doi.org/10.1007/978-3-030-17138-4_3

Christian Hammer, Jens Krinke, and Gregor Snelting. 2006. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *Proceedings IEEE International Symposium on Secure Software Engineering*. IEEE, Arlington, Virginia, USA, 10 pages.

Daniel Hedin and Andrei Sabelfeld. 2012. A Perspective on Information-Flow Control. In *Software Safety and Security - Tools for Analysis and Verification*, Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 33. IOS Press, Amsterdam, 319–347. https://doi.org/10.3233/978-1-61499-028-4-319

Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. 2016. *JRIF: reactive information flow control for Java*. Technical Report 1813–41194. Cornell University Computing and Information Science. https://ecommons.cornell.edu/handle/1813/41194

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 321–334. https://doi.org/10.1145/1294261.1294293

Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. 2024. *Cocoon artifact*. https://doi.org/10.5281/zenodo.10798978

Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (oct 1973), 613–615. https://doi.org/10.1145/362375.362389

Linux Foundation. 2022. Servo. https://servo.org

Mozilla Research. 2020. The Rust Language. Archived from https://research.mozilla.org/rust/.

A.C. Myers, A. Sabelfeld, and S. Zdancewic. 2004. Enforcing robust declassification. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* (Pacific Grove, CA, USA). IEEE, New York City, NY, USA, 172–186. https://doi.org/10.1109/CSFW.2004.1310740

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/268998.266669

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif 3.0: Java information flow. http://www.cs.cornell.edu/jif

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 40, 18 pages.

Flemming Nielson and Hanne Riis Nielson. 1999. *Type and Effect Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–136. https://doi.org/10.1007/3-540-48092-7_6

Noel. 2010. The Rust Language. http://lambda-the-ultimate.org/node/4009.

Leo J. Rotenberg. 1973. *Making Computers Keep Secrets*. Ph. D. Dissertation. Massachusetts Institute of Technology, Boston, MA.

Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. 2009. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, Michael Hind and Amer Diwan (Eds.). ACM, Dublin, Ireland, 63–74. https://doi.org/10.1145/1542476.1542484

Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 280–288. https://doi.org/10.1145/2784731.2784756

A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Vincent Simonet. 2003. *The Flow Caml System: documentation and user's manual*. Technical Report 0282. Institut National de Recherche en Informatique et en Automatique (INRIA). ©INRIA.

Geoffrey Smith and Dennis Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/268946.268975

Stack Overflow. 2022. Annual Developer Survey. https://survey.stackoverflow.co/2022/.

Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. 2023. A Type System for Safe Intermittent Computing. *Proc. ACM Program. Lang.* 7, PLDI, Article 136 (jun 2023), 25 pages. https://doi.org/10.1145/3591250

The Rust Foundation. 2022. Prodution Users — Rust Programming Language. https://www.rust-lang.org/production/users

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2–3 (jan 1996), 167–187.

Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 18–35. https://doi.org/10.1109/SP40001.2021.00002

Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. 2012. SAILS: Static Analysis of Information Leakage with Sample. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (Trento, Italy) *(SAC '12)*. Association for Computing Machinery, New York, NY, USA, 1308–1313. https://doi.org/10.1145/2245276.2231983

Steve Zdancewic and Andrew C. Myers. 2000. *Confidentiality and Integrity with Untrusted Hosts: Technical Report*. technical report. Cornell University.

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing Distributed Systems with Information Flow Control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, USA, 293–308.

Lantian Zheng and Andrew C. Myers. 2005. Dynamic Security Labels and Noninterference (Extended Abstract). In *Formal Aspects in Security and Trust*, Theo Dimitrakos and Fabio Martinelli (Eds.). Springer US, Boston, MA, 27–40.

Received 15-OCT-2023; accepted 2024-02-24