CARAPACE: Static–Dynamic Information Flow Control in Rust

VINCENT BEARDSLEY, Ohio State University, USA
CHRIS XIONG, Ohio State University, USA
ADA LAMBA, Ohio State University, USA
MICHAEL D. BOND, Ohio State University, USA

Fine-grained information flow control (IFC) ensures confidentiality and integrity at the programming language level by ensuring that high-secrecy values do not affect low-secrecy values and that low-integrity values do not affect high-integrity values. However, prior support for fine-grained IFC is impractical: It either analyzes programs using whole-program static analysis, detecting false IFC violations; or it extends the language and compiler, thwarting adoption. Recent work called *Cocoon* demonstrates how to provide fine-grained IFC for Rust programs without modifying the language or compiler, but it is limited to static secrecy labels, and its case studies are limited. This paper introduces an approach called *CARAPACE* that employs Cocoon's core approach and supports both static and dynamic IFC and supports both secrecy and integrity. We demonstrate CARAPACE using three case studies involving real applications and comprehensive security policies. An evaluation shows that applications can be retrofitted to use CARAPACE with relatively few changes, while incurring negligible run-time overhead in most cases. CARAPACE advances the state of the art by being the first hybrid static–dynamic IFC that works with an off-the-shelf language—Rust—and its unmodified compiler.

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:
Vincent Beardsley, Chris Xiong, Ada Lamba, and Michael D. Bond. 2025. CARAPACE: Static–Dynamic Information Flow Control in Rust. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 93 (April 2025), 29 pages. https://doi.org/10.1145/3720427

1 Introduction

Confidentiality and integrity are fundamental parts of secure computing systems. This paper is focused on *fine-grained* confidentiality and integrity policies, which ensure that applications granted privileges by the OS do not violate confidentiality and integrity. Consider a calendar application granted privileges to (1) read Alice's and Bob's calendars and (2) send data to Alice and Bob. Suppose the application should *not* reveal anything about Alice's calendar to Bob, or vice versa—except the calendars' overlapping availability *may* be revealed. This is a fine-grained confidentiality policy.

Today's applications provide fine-grained confidentiality and integrity by employing *access control*, which limits *which* entities may access *which* data, but does not control what happens to the data after being accessed. Access control *cannot* guarantee the calendar application's confidentiality policy, requiring the application to be part of the trusted computing base (TCB).

A strong alternative to access control is *information flow control (IFC)*, which ensures confidentiality and integrity through *noninterference*: High-secrecy data cannot affect low-secrecy data,

Authors' Contact Information: Vincent Beardsley, Ohio State University, USA, beardsley.49@buckeyemail.osu.edu; Chris Xiong, Ohio State University, USA, xiong.540@buckeyemail.osu.edu; Ada Lamba, Ohio State University, USA, lamba.39@buckeyemail.osu.edu; Michael D. Bond, Ohio State University, USA, mikebond@cse.ohio-state.edu.

and low-integrity data cannot affect high-integrity data [27, 37, 43, 48]. IFC can guarantee that the calendar's confidentiality policy is enforced. IFC also effectively removes applications from the TCB, except for explicit *declassify* and *endorse* operations, which must be trusted and audited.

Despite its benefits, fine-grained IFC is not used in practice because all existing approaches are impractical. As §2.3 covers in detail, prior work that uses whole-program static analysis to compute information flows is non-compositional, resulting in numerous false flows that are difficult for developers to reckon with. Prior work employing type-based static analysis or dynamic analysis avoids false flows, but existing solutions for mainstream imperative languages rely on *changing* the language and compiler (or introducing a source-to-source translator). Relying on a modified language and compiler is a substantial impediment to adoption: Programs must be (re)written in the modified language, and compiler modifications must be maintained and trusted.

An exception to the above limitations is type-based static IFC for Rust, called *Cocoon*, that requires no changes to the Rust language or compiler [35]. Programs using the Cocoon library are unable to violate noninterference, except through explicitly denoted declassification operations or explicitly unsafe Rust code. However, Cocoon has significant shortcomings limiting its practicality:

- Cocoon supports only static IFC, not dynamic IFC. Dynamic IFC is essential for many real security policies, particularly for data originating outside the process, which generally has secrecy and integrity levels not known at compile time. Since *static* IFC avoids run-time costs and run-time IFC violations, *hybrid static–dynamic* IFC is ideal.
- Cocoon supports secrecy but not integrity.
- Cocoon's case studies used narrow security policies that protected a single value with limited information flow, instead of a comprehensive policy such as protecting all user input data.

This paper addresses these limitations with novel support for fine-grained, hybrid static–dynamic IFC called *Carapace*. Carapace innovates by supporting both static labels and dynamic labels, as static types and run-time values, respectively, without sacrificing noninterference guarantees or requiring language or compiler changes.

To demonstrate and evaluate Carapace, we used it to retrofit three real, open-source Rust applications: an application that computes overlap and other information about multiple calendars from disparate sources, a multiplayer video game, and the Servo web rendering engine. We retrofitted the applications with realistic, comprehensive security and integrity policies. The evaluation shows that the retrofitted applications enforce the target security policy, Carapace can be incrementally deployed, and run-time overhead is negligible.

This work has the following contributions:

- The novelty of Carapace is in adding support for dynamic IFC to Cocoon while retaining compatibility with the unmodified Rust language and compiler (albeit relying on some unstable Rust features), paving the way for adoption.
- We apply Carapace to three real applications, demonstrating its potential as a practical solution.

While Carapace overcomes practicality limitations of prior work, practical challenges remain (§4.6). Overall, this work advances the state of the art by demonstrating fine-grained IFC for off-the-shelf Rust and its compiler that is powerful enough to provide real security policies in real applications.

2 Motivation, Background, and Limitations of Prior Work

This section motivates and gives technical background on fine-grained information flow control (IFC). It describes why prior work has failed to provide *practical* fine-grained IFC, articulating the challenges that a solution must address.

```
1   pub fn compute_average(grades: &Vec<i32>) -> i32 {
2       let mut total = 0;
3       for grade in grades {
4           total += grade;
5       }
6       total / grades.len()
7   }
8   ...
9   let grades: Vec<i32> = ...; /* read student-private grades from data store */
10  let avg = compute_average(&grades);
11  ... avg ... /* write student-visible grade average to data store */
```

Fig. 1. Rust code that computes and prints an average of student grades.

2.1 Motivating Example

Fig. 1 shows Rust code for computing the average of student grades that could be part of a gradebook application. Suppose the application process has (coarse-grained) privileges to read student grade data from outside the process (e.g., from a data store; line 9) and to write data back to the data store that will be visible to all students (line 11). The application should *not* be allowed to reveal students' grades to other students—a fine-grained secrecy policy. The application should only use grade data entered by the teacher to compute the average—is a fine-grained integrity policy.

Fine-grained IFC can enforce these policies through noninterference (§ 1): Data shown to a student should not be affected by another student's grade (secrecy), and the grade average shown to students should only be the result of data entered by the teacher (integrity).

If we want to allow the *average* grade to be shared with all students, then Fig. 1's code actually *needs to* violate noninterference. The application developer can allow the average to be "leaked" to student users by using an explicit *declassify* operation in the code, which is a trusted operation that should be audited. A similar *endorse* operation allows low-integrity data to affect high-integrity data. Later we show how this paper's IFC approach supports the grade average example.

2.2 Fine-Grained IFC Model

Our work uses the *decentralized IFC (DIFC)* model from prior work [34, 42–44, 49]. Every data value v has a secrecy label S_v and integrity label I_v. A *label* is a set of *tags*, sometimes called *policies*, which represent levels of secrecy and integrity. For example, if a_{sec} is a secrecy tag for student Alice's grades, then value v having secrecy label $S_v = \{a_{sec}\}$ means that v is secret to Alice. Labels, being sets of tags, naturally define a lattice based on inclusion. For secrecy labels S and S', $S \subseteq S'$ means S' is at least as secret as S. For integrity labels I and I', $I \subseteq I'$ means I' has higher or the same integrity as I. The join operations for secrecy and integrity are union and intersection, respectively: If v'' is derived from v and v' (i.e., $v'' \leftarrow v \oplus v'$), then $S_{v''} = S_v \cup S_{v'}$ and $I_{v''} = I_v \cap I_{v'}$.

A *principal* is an entity that operates on data, such as a user or process. Every principal P has a *capability set* C_P that may include t^- for any secrecy tag t, which allows the principal to *declassify* data (i.e., decrease secrecy), and t^+ for any integrity tag t, which allows the principal to *endorse* data (i.e., increase integrity).[1] For example, suppose the teacher Xenia is a principal with capability set $\{a^-_{sec}, b^-_{sec}, x^-_{sec}, a^+_{int}, b^+_{int}\}$, meaning she can declassify the grades of students Alice and Bob, can declassify grades labeled with her secrecy tag x_{sec}, and can endorse grades coming from students Alice and Bob as trusted. When Xenia runs the gradebook application, the application process is

[1]Some work also allows the capability t^+ for a secrecy tag (increasing secrecy) and t^- for an integrity tag (decreasing integrity). In our model, these tags are implicitly part of every capability set.

Table 1. Qualitative comparison of prior fine-grained IFC approaches targeting mainstream imperative languages. * Cocoon uses type-based static analysis without language or compiler modifications [35].

Kind of IFC	**Enforcement approach**	**OTS language?**	**Compositional?**	**Dynamic labels?**	**Perf. cost**
Static IFC	Interprocedural analysis [7, 22, 26, 63]	Yes	No	No	Zero
	Type-based analysis [16, 18, 33, 42–44, 53, 55, 56, 61]	No*	Yes	No	Zero
Dynamic IFC	Dynamic analysis [5, 6, 13, 24, 49, 54, 62]	No	Yes	Yes	High
Hybrid IFC	Hybrid analysis [12, 15, 17, 20, 23, 51, 64]	No	Depends	Yes	Medium

a principal P that inherits her capabilities. The a^-_{sec} and b^-_{sec} capabilities allow P to declassify the student average. In contrast, the student user Alice is a principal with capability set $\{a^-_{sec}\}$; when Alice runs the gradebook application, the application process cannot successfully declassify the average of Alice and Bob's grades, which has secrecy label $S_v = \{a_{sec}, b_{sec}\}$.

In a nutshell, fine-grained IFC can prevent the gradebook application from violating noninterference without explicit action from the teacher.

2.3 Existing Approaches and Their Limitations

Supporting fine-grained IFC means enforcing noninterference at the granularity of program values. *Static* fine-grained IFC enforces noninterference on the static program (i.e., at compile time) at the level of expressions and variables. *Dynamic* IFC enforces noninterference at run time on memory locations and their values. Static and dynamic IFC are complementary. Unlike dynamic IFC, static IFC adds no run-time overhead and avoids run-time IFC failures. On the other hand, only dynamic IFC supports values with labels that are not known until run time—which are common in real security policies. For example, if Fig. 1 is converted to enforce IFC, the labels of grades will not be known until run time, because the students are not known until run time.

Table 1 compares prior fine-grained IFC approaches qualitatively. Static IFC can be achieved through static analysis that is either whole-program (i.e., interprocedural) or type-based. *Whole-program analysis* is non-compositional—imprecision analyzing one part of a program affects precision analyzing the rest of the program—so it detects false noninterference violations, with little recourse for developers. *Type-based analysis* uses type annotations and inference to perform compositional analysis; intuitively, programmers refine types in order to achieve precise IFC. However, prior type-based approaches modify the language and/or compiler (because mainstream imperative languages do not provide type systems expressive enough for type-based analysis for IFC[2]), thwarting adoption. Recent work called *Cocoon*, which we overview below, introduces type-based static analysis for the unmodified Rust language and compiler, but it has some key limitations.

Dynamic analysis tracks secrecy and integrity labels of program values at run time. It can avoid language modifications by simply tracking every value's label, but it still requires modifying the compiler or creating a standalone tool to instrument the program. Much work on IFC enforcement for JavaScript uses this approach (§ 11). Alternatively, dynamic analysis can rely on programmer annotations of variables, expressions, and code that use high-secrecy or low-integrity values, which reduces run-time overhead but requires both language and compiler modifications.

[2]Prior work shows how to provide type-based static analysis without modifying the language for two *functional* languages, Haskell (using its support for monads) and Idris (using its support for dependent types) [25, 50, 58, 59].

Some prior work supports fine-grained, *hybrid static–dynamic IFC. Jif*, which is primarily static type-based IFC [44], provides support for dynamic labels as first-class values [64]. Jif requires modifying the (Java) language and provides a translator to Java source (§4.2). Other prior work hybridizes static and dynamic IFC by using a combination of static and dynamic analyses to analyze (primarily JavaScript) programs [12, 15, 20, 51] (§11). Tianyu and Siek show how to achieve noninterference guarantees in the context of gradual typing [17]. *RLBox* combines static type-based analysis and run-time isolation to prevent flows between untrusted C++ libraries, but relies on programmers writing correct validation checks [45]. *Sesame* enforces confidentiality policies in Rust programs and provides annotated code regions for accessing secret data [23]. Sesame relies on static analysis implemented as a compiler plugin to ensure regions are side effect free, and it employs dynamic sandboxing as a fallback when static analysis cannot ensure side effect freedom.

Cocoon. Recent work introduces static IFC for Rust called *Cocoon* [35] that uses static type-based analysis. Unlike other static type-based approaches, Cocoon does not alter a language or compiler, but is implemented as a Rust library. Applications use the library and the standard Rust language and compiler, and a program that violates noninterference is not type-correct and will not compile.

To use Cocoon to enforce a static IFC policy, the application developer uses types and code constructs provided by the Cocoon library. Every value with a non-empty secrecy label is wrapped in a Cocoon-provided *secure* type. (Cocoon supports secrecy but not integrity.) Code that operates on secure values must be enclosed in a Cocoon-provided *secure block*, a lexically scoped block of code with a secrecy label. To ensure noninterference in secure blocks, Cocoon leverages Rust's expressive typing and procedural macros, providing a rudimentary type and effect system [46].

Cocoon has three significant limitations: (1) it supports only static IFC, not dynamic IFC; (2) it supports only secrecy labels, not integrity labels; and (3) it was evaluated using narrow security policies. A fourth limitation, which this paper only partially addresses, is that Cocoon restricts the programming model within secure blocks, as a consequence of ensuring side effect freedom (§4.6).

Goals. To address these limitations, the following goals must be met while maintaining noninterference guarantees and supporting an unmodified Rust language and compiler:

- support for dynamic labels in addition to static labels;
- support for integrity labels (both static and dynamic) in addition to secrecy labels; and
- demonstration of realistic, comprehensive IFC policies in real applications.

The first two are design challenges, while the third is a case study and evaluation challenge.

3 Carapace Overview

Carapace is a novel, fine-grained, hybrid static–dynamic IFC approach. It is provided as a Rust library that applications use to ensure IFC. In this context, the trusted computing base (TCB) consists of Carapace, the Rust compiler, and the Rust Standard Library. Application code using Carapace is untrusted (i.e., not part of the TCB) because Carapace ensures noninterference, with the exception of two kinds of code that must be trusted and audited: (1) code explicitly denoted as declassifying or endorsing data and (2) explicitly unsafe code (code that uses Rust's `unsafe` keyword).

Carapace follows the IFC model described in §2.2. Every program value has both *static and dynamic* labels. Program values implicitly have default labels unless they are wrapped in a special Carapace-provided type. Furthermore, application code can only access wrapped values inside of lexically scoped regions called *secure blocks*. A special kind of secure block, called a *trusted secure block*, can declassify and endorse values (in accordance with capabilities inherited from its ancestors); its code must be trusted or audited.

Carapace advances the state of the art by overcoming the limitations of prior work and achieving the goals listed in the last section. The next two sections describe Carapace: §4 presents Carapace's programming model, and §5 explains how Carapace enforces IFC by ensuring noninterference.

4 Carapace's Programming Model

This section describes Carapace at the application programming level and presents example application code. It refers throughout to Figs. 2 and 3, which summarize the programming model.

4.1 IFC Model

Carapace provides the IFC model described in §2.2, with support for static and dynamic labels and for secrecy and integrity labels. Like most prior work on fine-grained IFC, Carapace provides *termination-insensitive noninterference*, which relaxes noninterference by allowing high-secrecy values to affect whether the program terminates [1, 3]. Thus the model does not prevent termination channels. It also does not prevent side channels such as timing and microarchitectural channels.

Carapace provides declassification and endorsement operations to circumvent noninterference; these operations must be audited and trusted. Carapace cannot provide guarantees for *unsafe* code, which Rust requires be explicitly denoted with the `unsafe` keyword and be audited for memory and concurrency safety. When using Carapace, any unsafe code must also be audited for security.

Our model defines integrity as the dual of confidentiality—untrusted inputs should not affect trusted outputs—which follows much of the prior work [27, 41, 43, 44, 53]. Notably, application code has highest integrity. Strengthening integrity to include correctness requires verification [14, 21, 53].

4.2 Static and Dynamic Labels

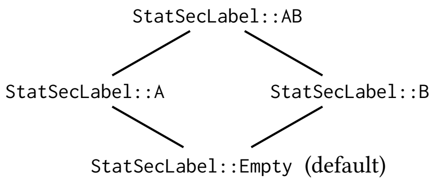
Carapace provides hybrid static–dynamic IFC by supporting both static and dynamic labels. Static and dynamic labels differ in how they are represented and checked. Static labels are represented as static types associated with variables and expressions, which the compiler checks. Dynamic labels are represented as values associated with program run-time values, which are checked by run-time analysis. A label is either static or dynamic—it consists entirely of static or dynamic tags, not both.

Carapace supports both secrecy and integrity. Secrecy labels contain only secrecy tags; integrity labels contain only integrity tags. Thus there are four kinds of labels and tags: static secrecy, static integrity, dynamic secrecy, and dynamic integrity.

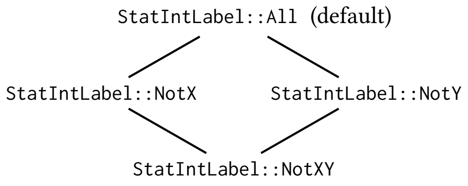
Static tags exist statically (i.e., at compile time). The static capabilities an application needs can be checked at compile time, so the program will fail upon startup if executed with insufficient static capabilities. Dynamic tags exist at run time; they may be created by the application process or another process. A dynamic tag created by process P is automatically added to P's capability set: $C_P \longleftarrow C_P \cup \{t^-\}$ if t is a dynamic secrecy tag, while $C_P \longleftarrow C_P \cup \{t^+\}$ if t is a dynamic integrity tag.

The default static and dynamic secrecy labels are both $\emptyset$ (empty set), representing lowest secrecy. The default static and dynamic integrity labels are $\mathbb{U}_P^{st} \equiv \{t \mid t^+ \in C_P^{st}\}$ and $\mathbb{U}_P^{dyn} \equiv \{t \mid t^+ \in C_P^{dyn}\}$, respectively, consisting of all (static or dynamic) integrity tags that the principal has the capability to add, representing highest integrity. Note that using $\mathbb{U}_P^{st}$ and $\mathbb{U}_P^{dyn}$ as the default integrity labels has the (intended) effect of treating the application code itself as high integrity.

Fig. 2a shows the lattice of Carapace's static secrecy labels, each represented by a static type, for an application using two static secrecy tags. Fig. 2b shows the lattice of static integrity labels when there are two static integrity tags. Since the default static integrity label $\mathbb{U}_P^{st}$ consists of all static integrity tags, static integrity labels are defined in terms of whether they do *not* contain static integrity tags. §5 explains how type constraints enforce the ordering defined by these lattices.



(a) Lattice of static secrecy labels. Higher-secrecy labels are above lower-secrecy labels.



(b) Lattice of static integrity labels. Higher-integrity labels are above lower-integrity labels.

API function	**Return value**
`DynSecLabel::empty() -> DynSecLabel`	Default dynamic secrecy label $S^{dyn} = \emptyset$
`DynIntLabel::all() -> DynIntLabel`	Default dynamic integrity label $I^{dyn} = \mathbb{U}_P^{dyn}$
`DynSecLabel::new(t: DynSecTag) -> DynSecLabel`	Dynamic secrecy label $S^{dyn} = \{t\}$
`DynIntLabel::new(t: DynIntTag) -> DynIntLabel`	Dynamic integrity label $I^{dyn} = \mathbb{U}_P^{dyn} \setminus \{t\}$
`DynSecLabel::join(&self, other: &DynSecLabel) -> DynSecLabel`	Dynamic secrecy label that is union of `self` and `other`
`DynIntLabel::join(&self, other: &DynIntLabel) -> DynIntLabel`	Dynamic integrity label that is intersection of `self` and `other`

(c) Dynamic labels, represented as values of type `DynSecLabel` or `DynIntLabel`, are opaque and immutable.

```
SecureValue<T: SecureValueSafe,
            S_v^st: StatSecLabelType,
            I_v^st: StatIntLabelType>
```

(d) `SecureValue` is the wrapper type for any value v with non-default secrecy and/or integrity. It is type-parameterized on static labels S_v^{st} and I_v^{st}.

Method	**Ret. val**
`SecureValue::get_dyn_sec_label(&self) -> DynSecLabel`	S_v^{dyn}
`SecureValue::get_dyn_int_label(&self) -> DynIntLabel`	I_v^{dyn}

(e) A secure value's dynamic labels S_v^{dyn} and I_v^{dyn} are run-time values, which applications can query with getter methods.

Fig. 2. Carapace's programming model, part 1 of 2: labels and labeled data.

While these lattices only use two tags, there is no firm restriction on the number of tags that can be used. Future work could provide a macro allowing applications to generate custom-sized lattices.

Fig. 2c shows the API for dynamic labels. Dynamic labels are represented by the `DynSecLabel` and `DynIntLabel` types. A dynamic label is a run-time value that represents a set of dynamic tags (`DynTag` instances). Dynamic tags and labels are *immutable* and *opaque*: Applications cannot change them, nor perform comparison operations on them, preventing certain covert channels [44, 64].

4.3 Labeled Data

In an application using Carapace, every program value v has static secrecy S_v^{st}, static integrity I_v^{st}, dynamic secrecy S_v^{dyn}, and dynamic integrity I_v^{dyn}, which default to the values described above. To represent a value with *non-default* labels, Carapace provides a type called `SecureValue`, which acts as a wrapper type for the value. As Fig. 2d shows, a `SecureValue` is type-parameterized on its static labels S_v^{st} and I_v^{st}. Fig. 2e shows `SecureValue`'s getter methods for dynamic labels.

```
untrusted_secure_block!(S_P^st, I_P^st, S_P^dyn, I_P^dyn, body)

trusted_secure_block!(S_P^st, I_P^st, S_P^dyn, I_P^dyn, S_out^st, I_out^st, S_out^dyn, I_out^dyn, body)
```

(a) Syntax for specifying trusted and untrusted secure blocks. S_P^{st}, I_P^{st}, S_P^{dyn}, and I_P^{dyn} represent, respectively, the static secrecy, static integrity, dynamic secrecy, and dynamic integrity of the block P. A trusted secure block takes four additional parameters, S_{out}^{st}, I_{out}^{st}, S_{out}^{dyn}, and I_{out}^{dyn}, which are the labels of the block's output value(s).

Operation and description of its behavior	Type constraints	Run-time checks
`unwrap(`v`: SecureValue<T,`S_v^{st}`,`I_v^{st}`>) -> T`	$S_v^{st} \subseteq S_P^{st}$	$S_v^{dyn} \subseteq S_P^{dyn}$
Evaluates to inner value	$I_v^{st} \supseteq I_P^{st}$	$I_v^{dyn} \supseteq I_P^{dyn}$
`unwrap_ref(`v`: &SecureValue<T,`S_v^{st}`,`I_v^{st}`>) -> &T`	$S_v^{st} \subseteq S_P^{st}$	$S_v^{dyn} \subseteq S_P^{dyn}$
Evaluates to immutable reference to value	$I_v^{st} \supseteq I_P^{st}$	$I_v^{dyn} \supseteq I_P^{dyn}$
`unwrap_mut_ref(`v`: &mut SecureValue<T,`S_v^{st}`,`I_v^{st}`>) -> &mut T`	$S_v^{st} = S_P^{st}$	$I_v^{dyn} = I_P^{dyn}$
Evaluates to mutable reference to value	$S_v^{st} = S_P^{st}$	$I_v^{dyn} = I_P^{dyn}$

(b) Operations for accessing the value in a `SecureValue` inside a secure block. S_P^{st}, I_P^{st}, S_P^{dyn}, and I_P^{dyn} are the labels of the secure block.

Operation	Evaluates to
`wrap(`v`)` in body of `untrusted_secure_block!(...)`	`SecureValue<_,`S_P^{st}`,`I_P^{st}`>::new(`v`, `S_P^{dyn}`, `I_P^{dyn}`)`
`wrap(`v`)` in body of `trusted_secure_block!(...)`	`SecureValue<_,`S_{out}^{st}`,`I_{out}^{st}`>::new(`v`, `S_{out}^{dyn}`, `I_{out}^{dyn}`)`

(c) Operation for creating a `SecureValue`-wrapped value inside of a secure block. S_P^{st}, I_P^{st}, S_P^{dyn}, and I_P^{dyn} are the labels of the secure block; S_{out}^{st}, I_{out}^{st}, S_{out}^{dyn}, and I_{out}^{dyn} are the output labels of the trusted secure block.

Fig. 3. Carapace's programming model, part 2 of 2: secure blocks and accesses to secure values.

4.4 Secure Blocks

Application code may access secure data (i.e., data wrapped in `SecureValue` instances) only in lexically designated blocks called *secure blocks*. A secure block has uniform labels throughout, which restrict the labels of values read, written, and returned by the block, constraining explicit and implicit flows. A secure block is a principal that inherits capabilities from its parent, which may be the application process or another secure block. Although a secure block could, in principle, specify a subset of its parent's capabilities, in our design a secure block has all its parent's capabilities.

To support declassification and endorsement, Carapace provides a variant of the secure block, called a *trusted* secure block. Trusted secure blocks allow flows from high- to low-secrecy values and from low- to high-integrity values, so programmers should audit and trust the block's code.

Fig. 3a shows the API for regular (untrusted) secure blocks and trusted secure blocks. (In §4.7 we show example code that uses secure blocks.) The `untrusted_secure_block!` macro[3] takes the block's four labels as input. Intuitively, the block enforces noninterference (e.g., high-secrecy values cannot flow to low-secrecy values) by ensuring that the accesses in the block are consistent with the block's labels and that the block's output value(s) are labeled with the block's labels.

Likewise, `trusted_secure_block!` takes parameters S_P^{st}, I_P^{st}, S_P^{dyn}, and I_P^{dyn}. It also takes parameters indicating the labels of the block's output value(s) (i.e., the block's return value and any values the block writes): S_{out}^{st}, I_{out}^{st}, S_{out}^{dyn}, and I_{out}^{dyn}. Carapace ensures that declassification and endorsement are

[3]In Rust, macro names are suffixed by the `!` character.

allowed by the secure block P's capability set C_P, by checking the following at run time:

$$(S_P^{dyn} \setminus S_{out}^{dyn}) \subseteq \{t \mid t^- \in C_P^{dyn}\} \ \wedge \ (I_{out}^{dyn} \setminus I_P^{dyn}) \subseteq \{t \mid t^+ \in C_P^{dyn}\}$$

Carapace does not need to perform analogous checks on *static* labels and capabilities because the principal has full capabilities for all static tags (§4.2).

By requiring secure values to be accessed in lexically scoped blocks with uniform labels, Carapace avoids many of the challenges related to implicit flows encountered by prior work (§5.3 and §11).

4.5 Accessing Secure Values in Secure Blocks

An application may access secure values (values wrapped in `SecureValue`) only in secure blocks. The code in a secure block uses *unwrap* operations, shown in Fig. 3b, to access the wrapped value. The unwrap operations differ in whether they return the value (`unwrap`) or a reference to it (`unwrap_ref` and `unwrap_mut_ref`). The *Type constraints* column shows the static IFC checks that Carapace's type constraints enforce. The *Run-time checks* column shows the run-time checks that execute on each unwrap call. Since `unwrap` and `unwrap_ref` allow read access only, their checks mitigate flows from the value to the block, thus requiring subset checks. In contrast, `unwrap_mut_ref` allows both read and write accesses, requiring equality checks to mitigate flows in both directions.

A violation of static type constraints is detected at compile time, causing a compiler error. A violation of run-time checks (including trusted secure blocks' capabilities checks) triggers a *panic*, which is Rust's mechanism for unexpected or unrecoverable run-time errors. Carapace handles panics in a way that prevents high-bandwidth covert channels, as explained in §5.3.

Applications can *create* `SecureValue` instances only in secure blocks.[4] As Fig. 3c shows, the `wrap` operation creates a new `SecureValue` instance with the same output labels of the block. Note that `wrap` is not a privileged operation—it requires no type constraints or run-time checks—because it always evaluates to a value with the same labels as the containing secure block.

4.6 Practicality Limitations

Carapace addresses some key practical limitations of Cocoon. However, Carapace has remaining practical limitations resulting from the requirement that all code in secure blocks be statically guaranteed to be side effect free (§5.4). This requirement limits what values can be used in secure blocks (e.g., values with custom destructors cannot be used). Furthermore, only side-effect-free functions can be called from secure blocks.

Secure blocks can make calls to Rust Standard Library functions that have been "allowlisted" by the Carapace implementation. A production-ready implementation of Carapace should allowlist as many Rust Standard Library functions as possible. To call a third-party library function, application developers have two options. They can extend the library function to use Carapace. Alternatively, developers can choose to trust the function, by declassifying/endorsing data sent to the function (if called outside of a secure block) or by using a special trusted operation provided for calling ordinary functions from side-effect-free contexts (if called inside of a secure block).

Future work can address these limitations by introducing static analysis that analyzes Rust Standard Library and third-party library functions to ensure they are side effect free.

4.7 Programming Model Examples

The following examples show application code that could be part of a gradebook application. Suppose there are multiple students known only at run time, but statically only one teacher, so the students' tags are represented dynamically, while the teacher's tags are represented statically.

[4]Otherwise, an untrusted macro could transform the `SecureValue`-creating expression.

```
type TeacherSec        = StatSecLabel::A;
type TeacherPub        = StatSecLabel::Empty;
type TeacherEndorsed   = StatIntLabel::All;
```

Fig. 4. Type aliases (for static label types from Fig. 2a and Fig. 2b) used in Figs. 5–7.

Our examples use non-default labels for static secrecy, dynamic secrecy, and dynamic integrity—but default labels for static integrity, since the only source of untrusted data is from students. Fig. 4 shows intuitively named type aliases for static labels used by the examples in this section. Throughout the example executions in this section, there are two students at run time, Alice and Bob, who have dynamic secrecy tags a_{sec} and b_{sec}, respectively. Low-integrity data produced by Alice and Bob is represented by dynamic integrity tags a_{int} and b_{int}, respectively.

Adding a bonus to a grade. Fig. 5a shows a function containing a secure block. The function's parameters are secure values: a grade and a bonus to be added to the grade. The type of each secure value is a 32-bit integer (`i32`). The static secrecy label of `grade` is `GradeSec`, which is a type parameter of the function, allowing the function to support grades with different static secrecy labels. The other static labels of `grade` and `bonus` are fixed types defined in Fig. 4. The block's code unwraps `grade` mutably and `bonus` immutably, and adds the bonus to the grade.

Figs. 5b and 5c show scenarios involving different static secrecy labels of `grade`. In Fig. 5b's scenario, `grade`'s S^{st}_{val} is `TeacherSec` (`StatSecLabel::A`), meaning that the grade is secret to the teacher. The secure block *blk* compiles successfully because `grade`'s $S^{st}_{val} = S^{st}_{blk}$ and `bonus`'s $S^{st}_{val} \subseteq S^{st}_{blk}$. At run time, the unwrap calls succeed because `grade`'s $S^{dyn}_{val} = S^{dyn}_{blk}$ and `bonus`'s $S^{dyn}_{val} \subseteq S^{dyn}_{blk}$.

In contrast, Fig. 5c supposes that `grade`'s S^{st}_{val} is `TeacherPub` (`StatSecLabel::Empty`), meaning that the grade is public. However, the bonus is *not* public: `bonus`'s S^{st}_{val} is `TeacherSec` (`StatSecLabel::A`), so the type constraint on `unwrap_ref(bonus)` fails since $S^{st}_{val} \not\subseteq S^{st}_{blk}$, generating a compiler error.

Declassifying or endorsing a grade. Fig. 6 shows code that uses a trusted secure block that declassifies and/or endorses `grade`'s value depending on the values of static labels `GradeSec` and `OutSec` and dynamic labels `out_sec` and `out_int`, returning the grade as a new value with the block's labels.

Figs. 6b and 6c show two run-time scenarios. In Fig. 6b, the teacher is the principal P and has capabilities to remove Alice's and Bob's dynamic secrecy tags ($\{a^-_{sec}, b^-_{sec}\}$). This scenario supposes that `grade` is Alice's grade, so it is labeled with the secrecy labels of Alice ($\{a_{sec}\}$) and the teacher (`TeacherSec`). The declassification successfully produces a new value with static secrecy label `TeacherPub` because principals have capabilities for all static tags (§4.2).

In Fig. 6c, the teacher is the principal P and has capabilities to add Alice's and Bob's integrity tags ($\{a^+_{int}, b^+_{int}\}$). In this scenario, the proposed new grade is already readable by Alice (it has static secrecy label `TeacherPub`). The proposed grade comes from Alice, so its integrity label I^{dyn}_{val} lacks a_{int}. Since `out_int` includes a_{int} but grade's I^{dyn}_{val} does not, the block returns the grade value endorsed with the a_{int} tag, which succeeds because $a^+_{int} \in C_{blk}$.

Computing and declassifying an average of grades. Fig. 7 shows a secure block that computes the average of a vector of grades, which a trusted secure block then declassifies. The code is comparable to Fig. 1's insecure code.

The code assumes that the grades have been endorsed by the teacher (each grade's I^{st}_{val} is `TeacherEndorsed`), and the teacher has released all or no grades (the type parameter `GradeSec` may be `TeacherSec` or `TeacherPub`). Because static labels are represented as static types, all grades in the vector must have the same static labels, which is a limitation of static labels.

```
pub fn add_bonus_to_grade<GradeSec: StatSecLabelType>(
           grade: &mut SecureValue<i32, GradeSec,   TeacherEndorsed>,
           bonus:     &SecureValue<i32, TeacherSec, TeacherEndorsed>) {
    untrusted_secure_block!(GradeSec,                 TeacherEndorsed, // static labels
                        grade.get_dyn_sec_label(), grade.get_dyn_int_label(), { // dyn. labels
        *unwrap_mut_ref(grade) += *unwrap_ref(bonus);
    });
}
```

(a) The function takes parameters that are `SecureValue` instances, which it operates on using a secure block. Because `grade` is unwrapped mutably, allowing both read and write access, its labels must match the block's.

`GradeSec` ≔ `TeacherSec`
$C_{blk} \coloneqq \{a^-_{sec}, b^-_{sec}, a^+_{int}, b^+_{int}\}$
Block labels: $S^{dyn}_{blk} \coloneqq \{a_{sec}\}$, $I^{dyn}_{blk} \coloneqq \mathbb{U}^{dyn}_P$

Value	S^{dyn}_{val}	I^{dyn}_{val}
grade	$\{a_{sec}\}$	$\mathbb{U}^{dyn}_P$
bonus	$\{\}$	$\mathbb{U}^{dyn}_P$

(b) Scenario: Teacher adds unreleased bonus to student Alice's *unreleased* grade.

`GradeSec` ≔ `TeacherPub`

Won't compile: `unwrap_ref (bonus)` requires `bonus`'s $S^{st}_{val} \subseteq S^{st}_{blk}$, but `bonus`'s S^{st}_{val} is `TeacherSec` (`StatSecLabel::A`), while S^{st}_{blk} is `TeacherPub` (`StatSecLabel::Empty`).

(c) Scenario: Teacher adds unreleased bonus to student Alice's *already-released* grade.

Fig. 5. Example application code demonstrating Carapace's programming model. Parts (b) and (c) each show a usage scenario, `GradeSec`'s concrete type, and run-time behavior. In (b), C_{blk} is the block's capabilities, and S^{dyn}_{blk} and I^{dyn}_{blk} are the block labels. The table shows the dynamic labels of `grade` and `bonus`. In (c), there is no run-time behavior since static type checking fails.

The block's dynamic labels, `blk_sec` and `blk_int`, must be the join of all grades' dynamic secrecy and integrity labels, respectively. We assume that, for efficiency, `blk_sec` and `blk_int` are computed just once and passed to the function on each call. (Alternatively, the function could compute `blk_sec` and `blk_int` by joining the grades' dynamic secrecy and integrity labels, respectively.)

Figs. 7b and 7c show two scenarios. In both scenarios, the vector contains the grades for two students, Alice and Bob. The difference between the scenarios lies in who the principal is: In Fig. 7b, the principal is the teacher; in Fig. 7c, the principal is Alice. In both scenarios, the (untrusted) secure block successfully computes the average, which has dynamic secrecy label $\{a_{sec}, b_{sec}\}$. In Fig. 7b, the trusted secure block succeeds because the teacher has the capabilities to remove all of the dynamic secrecy label's tags. In contrast, in Fig. 7c the trusted secure block triggers a run-time error because Alice does not have the capabilities to remove Bob's dynamic secrecy tags.

5 How Carapace Ensures Noninterference

5.1 How Carapace Represents Labels and Secure Values

Static labels. §4.5 and Figs. 2a and 2b show how Carapace represents static labels using static types. To ensure that the application obeys static IFC rules, Carapace extends prior work Cocoon's approach [35], using type constraints so that the application will not compile unless static secrecy and integrity IFC rules are obeyed.

Dynamic labels. Carapace represents dynamic labels, which are run-time sets of tags, using the types `DynSecLabel` and `DynIntLabel` (§4.2). While `DynSecLabel` stores the label's tags, `DynIntLabel` stores the tags that are *not* part of the label. This behavior makes sense because of how integrity works:

```
pub fn declassify_and_or_endorse_grade<GradeSec: StatSecLabelType, OutSec: StatSecLabelType>(
           grade: &SecureValue<i32, GradeSec, TeacherEndorsed>,
           out_sec: DynSecLabel, out_int: DynIntLabel
) -> SecureValue<i32, OutSec, TeacherEndorsed> {
    trusted_secure_block!(GradeSec,                   TeacherEndorsed, // blk static labels
                          grade.get_dyn_sec_label(), grade.get_dyn_int_label(), // blk dyn. labels
                          OutSec,                     TeacherEndorsed, // output static labels
                          out_sec,                    out_int, { // output dyn. labels
        wrap(*unwrap_ref(grade))
    })
}
```

(a) The function uses a trusted secure block to declassify and/or endorse a grade value.

`GradeSec` ≔ `TeacherSec`
`OutSec` ≔ `TeacherPub`
$C_{blk} = \{a^-_{sec}, b^-_{sec}, a^+_{int}, b^+_{int}\}$
`grade`: $S^{dyn}_{val} = \{a_{sec}\}$, $I^{dyn}_{val} = \mathbb{U}^{dyn}_P$
$S^{dyn}_{blk} = \{\, a_{sec} \,\}$, $I^{dyn}_{blk} = \mathbb{U}^{dyn}_P$
`out_sec` $= \{a_{sec}\}$, `out_int` $= \mathbb{U}^{dyn}_P$
Return value: $S^{dyn}_{val} = \{a_{sec}\}$, $I^{dyn}_{val} = \mathbb{U}^{dyn}_P$

(b) Scenario: Teacher releases student Alice's grade.

`GradeSec` ≔ `TeacherPub`
`OutSec` ≔ `TeacherPub`
$C_{blk} = \{a^-_{sec}, b^-_{sec}, a^+_{int}, b^+_{int}\}$
`grade`: $S^{dyn}_{val} = \{a_{sec}\}$, $I^{dyn}_{val} = \mathbb{U}^{dyn}_P \setminus \{a_{int}\}$
$S^{dyn}_{blk} = \{\, a_{sec} \,\}$, $I^{dyn}_{blk} = \mathbb{U}^{dyn}_P \setminus \{a_{int}\}$
`out_sec` $= \{a_{sec}\}$, `out_int` $= \mathbb{U}^{dyn}_P$
Return value: $S^{dyn}_{val} = \{a_{sec}\}$, $I^{dyn}_{val} = \mathbb{U}^{dyn}_P$

(c) Scenario: Teacher endorses student Alice's proposed new grade.

Fig. 6. Example application code demonstrating declassification and endorsement. Parts (b) and (c) each show a usage scenario. Each scenario first presents the concrete types of generics `GradeSec` and `OutSec`. Then each part shows run-time behavior: the block's capabilities (C_{blk}), `grade`'s labels, the block's labels (S^{dyn}_{blk} and I^{dyn}_{blk}), the run-time values of parameters `out_sec` and `out_int`, and the return value's labels.

the default dynamic integrity label is $\mathbb{U}^{dyn}_P$, the set of all dynamic integrity tags in the process's capability set (§4.2); that label is represented by a `DynIntLabel` containing no tags. Thus intersection of `DynIntLabel` instances is implemented as union, while superset is implemented as subset.

The implementation supports dynamic labels with any number of tags, thus avoiding label creep. To optimize time and space for the case in which dynamic labels are small (i.e., few tags), the size of a `DynSecLabel`/`DynIntLabel` instance is just one 64-bit word, which represents either a few tags directly or many tags via indirection. For a `DynSecLabel`/`DynIntLabel` representing a label with 0–2 tags, the tag(s) are stored directly in the 64-bit word: Each `DynSecTag`/`DynIntTag` is 31 bits, and the lowest bit of the word is set to 1 to indicate that the representation stores tag(s) directly.

The implementation represents a `DynSecLabel` with >2 tags as an `Arc<HashSet<DynSecTag>>` (similarly for `DynIntLabel` and `DynIntTag`) indexed by tags' unique IDs. The `HashSet` implementation (from the Rust Standard Library) uses an array of buckets and handles collisions by mapping each bucket to an automatically resizing list of values. By implementing large labels in this way, the `Arc<HashSet<DynSecTag>>` is equivalent to a pointer to a reference-counted `HashSet`. The word's lowest bit is 0 because the pointer's value is word-aligned, differentiating the two representations.

Dynamic labels in secure values. Figs. 2d and 2e implied that Carapace maintains dynamic labels for every `SecureValue` instance. This would add unnecessary space and time overheads for values that have *statically known* default dynamic labels. To avoid these costs, Carapace supports secure

```
pub fn compute_and_declassify_average<GradeSec: StatSecLabelType>(
           student_grade_vec: &Vec<SecureValue<i32, GradeSec, TeacherEndorsed>>,
           blk_sec: DynSecLabel, blk_int: DynIntLabel,
           out_sec: DynSecLabel
) -> SecureValue<i32, TeacherPub, TeacherEndorsed> {
    let classified_average = untrusted_secure_block!(GradeSec, TeacherEndorsed, // static labels
                                                     blk_sec,  blk_int, { // dynamic labels
        let mut total = 0;
        for secure_grade in student_grade_vec {
            total += unwrap(secure_grade);
        }
        wrap(total / student_grade_vec.len() as i32)
    });
    trusted_secure_block!(GradeSec,   TeacherEndorsed, blk_sec, blk_int, // block labels
                          TeacherPub, TeacherEndorsed, out_sec, blk_int, { // output labels
        wrap(unwrap(classified_average))
    })
}
...
let grades: Vec<SecureValue<i32, TeacherPub, TeacherEndorsed>> = ...; /* read from data store */
let declassified_avg: SecureValue<i32, TeacherPub, TeacherEndorsed> =
  compute_and_declassify_average(&grades, grades_dyn_sec, grades_dyn_int, DynSecLabel::empty());
... declassified_avg ... /* write student-visible grade average to data store */
```

(a) This code uses a secure block to compute the average grade of a set of grades, followed by a trusted secure block to declassify the average. The code outside of `compute_and_declassify_average` shows how the function could be called (and it corresponds to Fig. 1).

`GradeSec` := `TeacherPub`

$C_{blk} := \{a^{-}_{sec}, b^{-}_{sec}, a^{+}_{int}, b^{+}_{int}\}$

Value	S^{dyn}_{val}	I^{dyn}_{val}
`student_grade_vec[0]`	$\{a_{sec}\}$	$\mathbb{U}^{dyn}_{P}$
`student_grade_vec[1]`	$\{b_{sec}\}$	$\mathbb{U}^{dyn}_{P}$

`out_sec` $= \{\}$

`blk_sec` $= \{a_{sec},\ b_{sec}\}$, `blk_int` $= \mathbb{U}^{dyn}_{P}$

`blk_sec` $\setminus$ `out_sec` $= \{a_{sec},\ b_{sec}\} \subseteq \{t \mid t^{-} \in C_{blk}\}$

`out_int` $\setminus$ `blk_int` $= \{\} \subseteq \{t \mid t^{+} \in C_{blk}\}$

(b) Scenario: Teacher successfully computes and releases the average of Alice and Bob's grades.

`GradeSec` := `TeacherPub`

$C_{blk} := \{a^{-}_{sec}, b^{-}_{sec}, a^{+}_{int}, b^{+}_{int}\}$

Value	S^{dyn}_{val}	I^{dyn}_{val}
`student_grade_vec[0]`	$\{a_{sec}\}$	$\mathbb{U}^{dyn}_{P}$
`student_grade_vec[1]`	$\{b_{sec}\}$	$\mathbb{U}^{dyn}_{P}$

`out_sec` $= \{\}$

`blk_sec` $= \{a_{sec},\ b_{sec}\}$, `blk_int` $= \mathbb{U}^{dyn}_{P}$

`blk_sec` $\setminus$ `out_sec` $= \{a_{sec},\ b_{sec}\} \not\subseteq \{t \mid t^{-} \in C_{blk}\}$,
so a run-time failure occurs!

(c) Scenario: Alice is not allowed to compute and release the average of her and Bob's grades.

Fig. 7. Example code demonstrating a sequential pair of blocks allowing declassification/endorsement of information after computation. Parts (b) and (c) each show a usage scenario, first showing the type of generic `GradeSec`. Then each part shows run-time behavior: the block's capabilities (C_{blk}); the labels of student grades; and the block's labels, which come from parameters `blk_sec`, `blk_int`, and `out_sec`.

values that statically have default dynamic secrecy and/or integrity labels, implementing secure values as follows (effectively extending Fig. 2d with two type parameters):

Algorithm 1 Dynamic IFC operations performed by a secure block P with dynamic labels S_P^{dyn} and I_P^{dyn} and (for trusted secure blocks only) output dynamic labels S_{out}^{dyn} and I_{out}^{dyn}.

```
1: At beginning of block:
2:     if $P$ is trusted secure block then
3:         check $(S_P^{dyn} \setminus S_{out}^{dyn}) \subseteq \{t \mid t^- \in C_P^{dyn}\} \wedge (I_{out}^{dyn} \setminus I_P^{dyn}) \subseteq \{t \mid t^+ \in C_P^{dyn}\}$
4:     else
5:         $S_{out}^{dyn} := S_P^{dyn}$        ▷ Set output secrecy to block's secrecy
6:         $I_{out}^{dyn} := I_P^{dyn}$        ▷ Set output integrity to block's integrity
7: At unwrap ($v$) or unwrap_ref ($v$):
8:     check $S_v^{dyn} \subseteq S_P^{dyn} \wedge I_P^{dyn} \subseteq I_v^{dyn}$
9: At unwrap_mut_ref ($v$):
10:    check $S_v^{dyn} = S_P^{dyn} \wedge I_P^{dyn} = I_v^{dyn}$
11: At wrap ($v$):
12:    Evaluate to new SecureValue with value $v$ and dynamic labels $S_{out}^{dyn}$ and $I_{out}^{dyn}$
13: At end of block with return value $v$:
14:    check $S_v^{dyn} = S_{out}^{dyn} \wedge I_v^{dyn} = I_{out}^{dyn}$
```

```
struct SecureValue<T: SecureValueSafe, S_v^st: StatSecLabelType, I_v^st: StatIntLabelType,
                   DynSecLabelType: DynSecLabelOrDefault, DynIntLabelType: DynIntLabelOrDefault> {
    value: T, S_v^dyn: DynSecLabelType, I_v^dyn: DynIntLabelType,
}
```

where `DynSecLabelOrDefault` is a trait implemented by two types: `DynSecLabel` and the empty type `()` (similarly for `DynIntLabelOrDefault`). For example, the type `SecureValue<T,` S_v^{st}`,` I_v^{st}`, DynSecLabel, ()>` represents a secure value that can have any dynamic secrecy label but that statically has the default dynamic integrity label.

5.2 How Carapace Handles Explicit Flows

Here we describe *how* Carapace enforces §4.5's rules for accesses to secure values in secure blocks.

Static IFC. Carapace enforces static IFC rules using static type constraints, e.g., using the `LowerIntegrityThan<DynIntLabel>` trait described above. Prior work Cocoon already uses this approach for static secrecy checks [35], and Carapace adds similar support for static integrity checks.

Dynamic IFC. Algorithm 1 shows the dynamic analysis performed by Carapace on operations in secure blocks. As line 3 shows, the analysis checks at the beginning of a *trusted* secure block that the principal has the capabilities to perform the specified declassification and/or endorsement (the same check as in §4.4). At the beginning of an *untrusted* secure block, the analysis initializes S_{out}^{dyn} to S_P^{dyn} and I_{out}^{dyn} to I_P^{dyn} (lines 5 and 6), since the block has the same output labels as the block's labels, allowing the rest of the analysis to use S_{out}^{dyn} and I_{out}^{dyn} regardless of block type.

At unwrap operations, the analysis performs checks that the accessed values' labels are compatible with the block's labels (lines 7–10). These checks correspond to the dynamic checks in Fig. 3b.

The rest of the algorithm shows the analysis for `wrap` calls and the block's return value, which ensures that every outputted value has the same labels as the block's output labels (S_{out}^{dyn} and I_{out}^{dyn}).

If a **check** operation fails at run time, it generates a panic (which does not create a high-bandwidth covert channel because of how Carapace handles panics; §5.3). Carapace implements Algorithm 1's dynamic analysis by implementing `untrusted_secure_block!` and `trusted_secure_block!` as Rust procedural macros, which can perform arbitrary compile-time syntax transformations.

```
let x = ??; // x is secret
let mut y = true; // y is public
let mut z = true; // z is public
if x {
    y = false;
}
if y {
    z = false;
}
return z;
```

(a)

```
let x: SecureValue<bool, ...> = /* SecureValue<...>(??, ...) */
let mut y: SecureValue<bool, ...> = /* SecureValue<...>(true, ...) */
let mut z: SecureValue<bool, ...> = /* SecureValue<...>(true, ...) */
secure_block!(..., {
    if unwrap(x) { *unwrap_mut_ref(&mut y) = false; }
});
secure_block!(..., {
    if unwrap(y) { *unwrap_mut_ref(&mut z) = false; }
});
return z;
```

(b)

Fig. 8. (a) Implicit flow example adapted from prior work [6]. (b) CARAPACE handles implicit flows using lexically scoped secure blocks with uniform labels.

Restricting accesses outside of secure blocks. CARAPACE ensures that code cannot read or write secure values outside of secure blocks, by making `SecureValue`'s data field private and providing access methods that are `unsafe`. The only way to access the data field (without using explicitly `unsafe` code, forgoing CARAPACE's guarantees) is to use unwrap operations inside of a secure block (Fig. 3b). The `untrusted_secure_block!` and `trusted_secure_block!` procedural macros expand unwrap operations to use `unsafe` blocks. CARAPACE reuses this functionality from prior work Cocoon [35].

5.3 How CARAPACE Handles Implicit Flows

Information flow can be either explicit or implicit. An explicit flow involves only data dependence, while an implicit flow includes control dependence [32]. As an example, consider Fig. 8a, adapted from Austin and Flanagan's paper on dynamic IFC for JavaScript [6]. Austin and Flanagan's and other work, particularly on IFC for JavaScript programs, must account for both *direct* and *indirect* implicit flows from `x` to `y` to `z` [5, 6, 12, 13, 15, 20, 24, 51, 54].

CARAPACE prohibits illegal implicit flows by using lexically scoped blocks with labels that are uniform throughout the block's code. Prior work Cocoon [35] and Laminar [47, 49] prohibit illegal implicit flows in a similar way. By requiring every secure value to be accessed in a lexically scoped, single-exit block (i.e., a secure block), illegal implicit flows are rendered infeasible. In Fig. 8a, since `x` is secret, CARAPACE requires `if x { y = false; }` and `if y { z = false; }` to each be wholly in a secure block (the two could optionally be combined into one block). This forces `y` and `z` to be secure values—otherwise the program cannot compile. The resulting code resembles Fig. 8b.

To ensure no illegal implicit flows occur between secure blocks and their surrounding code, secure blocks must have a single control-flow exit. Consider a loop containing a secure block that `break`s out of the loop depending on a secure value, which may violate noninterference. Secure blocks cannot exit via `return` or `break` because CARAPACE wraps a secure block's body in a closure.

Handling exceptional control flow. What about implicit flows created by exceptional control flow? A secure block might panic due to a dynamic label check failure or for another reason, e.g., if the block calls a Rust Standard Library function that panics. Rust's default behavior for a panic is to terminate the application, which satisfies termination-insensitive noninterference (§4.1). However, application code might *catch* a secure block's panic, creating a high-bandwidth covert channel.

Askarov and Sabelfeld address this issue with type restrictions on effects that could occur after any operation that might generate an exception [4]. Laminar addresses the issue by requiring all secure blocks to catch exceptions explicitly [49]. Similarly, in Cocoon every secure block catches any panics automatically and evaluates to a default value [35]. However, this behavior is still problematic because the application can set a noninterference-violating "panic hook" that executes on any panic.

A possible solution is for every secure block to set an empty panic hook when it starts and restore the old panic hook when it ends, but we found that this approach adds noticeable run-time overhead for frequently executed secure blocks. Instead, Carapace calls `std::panic::always_abort()` when the application starts, which ensures that any panic exits the program without calling the panic hook. Note that Carapace only terminates silently on panic when compiled in `release` mode. When compiled in `debug` mode, Carapace uses the default panic hook behavior for better debuggability.

5.4 Preventing Side Effects in Secure Blocks

So far we have described how Carapace ensures that accesses to *secure values* do not violate IFC rules. However, preventing illegal explicit and implicit flows requires additional restrictions to ensure that accesses to *ordinary values* do not violate noninterference. Carapace reuses prior work Cocoon's approach for disallowing side effects [35]. In the following, we discuss three kinds of potential side effects that Carapace disallows in secure blocks.

Side effects via mutation. What if a secure block writes to an unwrapped (and thus low-secrecy) value whose scope is larger than the block? Carapace (and Cocoon) use a closure for the block's code and disallow the closure from capturing mutable references except for mutable references to `SecureValue` values. This constraint is enforced using Rust auto traits and negative implementations.

Side effects in callees. What if a secure block performs I/O to leak a secret, e.g., to write a high-secrecy value to an untrusted socket? Carapace (and Cocoon) ensure that secure blocks can transitively call only *side-effect-free* functions. Every call in a secure block or side-effect-free function is transformed by the `untrusted_secure_block!` and `trusted_secure_block!` macros so that it can compile only if the callee is either (1) a Rust Standard Library function that has been manually "allowlisted" as side effect free by Carapace or (2) an application function that has been annotated as side effect free using the `#[side_effect_free_attr]` attribute macro. In Cocoon, a consequence of these limitations was that every method call had to be fully qualified, e.g., `::std::vec::Vec::len(student_grade_vec)`, increasing code complexity [35]. In contrast, Carapace adds support for simple (i.e., unqualified) method calls in secure blocks, so programmers can simply write `student_grade_vec.len()`, as in Fig. 7.

Invisible side effects. Even with the above restrictions, there are a few kinds of side effects that Carapace's procedural macros cannot detect because they are not visible at the level of syntax: overloaded operators, custom dereference operations, and custom destructor calls. Carapace (and Cocoon) use their procedural macros to transform every expression in secure blocks and side-effect-free functions so it cannot compile if it uses overloaded operators or uses a type that implements a custom dereference operation or custom destructor. Specifically, types can implement a Carapace-provided trait called `InvisibleSideEffectFree` only if they do not implement syntactically invisible operations; Carapace implements `InvisibleSideEffectFree` for a wide variety of Rust library types. An application can mark an application-defined type with the `#[derive(InvisibleSideEffectFree)]` macro, which ensures that the type (transitively) does not provide syntactically invisible operations.

5.5 Implementation Details

Our implementation of Carapace extends the publicly available Cocoon implementation [35, 36]. Carapace is a Rust library; it is implemented as two Rust crates since procedural macros must go in their own crate. Carapace requires the *nightly* version of Rust in order to use a few unstable features including auto traits and negative trait implementations. The evaluation uses version `1.69.0-nightly` of the Rust compiler.

Table 2. Microbenchmark results. Run times have three significant digits and 95% confidence intervals.

	Statistics			Run times		
Benchmark	Secure blocks	(trusted)	Label comps.	Original	W/Carapace	Slowdown
add_bonus	10,000,000	(0)	20,000,000	12,800 ± 225 μs	15,600 ± 315 μs	1.22 ×
average	10,030,000	(10,000)	40,080,000	2,120 ± 36.9 μs	606,000 ± 4,470 μs	286 ×

6 Evaluation Overview

We evaluated the effectiveness and performance of Carapace using microbenchmarks and real applications. § 7 describes evaluation using microbenchmarks based on the paper's gradebook examples. § 8–§ 10 describe how we extended three applications with comprehensive security policies and evaluated the resulting effectiveness and performance.

All experiments ran on a MacBook Pro laptop with a quad-core Intel Core i7-7920HQ at 3.1 GHz with 16 GB of RAM, running macOS 12.7.6.

7 Microbenchmarks Evaluation

To evaluate the behavior and performance of Carapace in isolation, we implemented two microbenchmarks, called add_bonus and average, based on the paper's running examples. The add_bonus benchmark adds values to grades by calling `add_bonus_to_grade` from Fig. 5a on students' grades. The average benchmark computes the average of all grades and declassifies the average, by calling `compute_and_declassify_average` from Fig. 7a.

For each benchmark, we created both non-Carapace and Carapace versions. Each benchmark manages a gradebook of 1,000 student grades. The gradebook is a vector (`Vec`) of student records, which each consist of a student name (string) and a numerical grade (32-bit integer). In the Carapace versions, each grade is wrapped in a `SecureValue` with a dynamic secrecy label with a single tag unique to the student and a dynamic integrity label with a single tag unique to the student.

To factor out noise and one-time costs (e.g., compulsory cache misses), the add_bonus benchmark executes `add_bonus_to_grade` 10,000,000 times. To prevent the compiler from applying vectorization optimizations that we think would not apply in most real-world gradebook scenarios, the benchmark iterates over the grades by calling `add_bonus_to_grade` on every 17th grade (mod 1,000) in the vector. The average microbenchmark executes the `compute_and_declassify_average` function 10,000 times.

To get the run time of each benchmark, we measured the wall-clock time of only its outer loop, and ran 100 trials for each benchmark to account for run-to-run variation. We compiled the code with `rustc` using `release` mode with link-time optimization (LTO), which enables cross-crate optimization and inlining. To collect statistics such as number of label comparisons, we used separate executions compiled with support for incrementing statistics counters.

Table 2 shows the run times of the microbenchmarks without and with Carapace. As the results show, Carapace adds moderate run-time overhead to add_bonus, despite its heavy use of secure blocks and label comparisons. We determined that the benchmark's 22% run-time overhead is attributable to its 20,000,000 label comparisons, which check whether the grade's labels are equal to the block's labels, which are executed as part of `unwrap_mut_ref(grade)` in Fig. 5a. Each equality check takes a fast path: It checks the lowest bit of one of the 64-bit values, which is always set (each label contains only one tag), so it performs a simple comparison of two 64-bit values. Note that `unwrap_ref(bonus)` adds no run-time overhead since no dynamic checks are required: `bonus` has statically default dynamic labels in our implementation of add_bonus.

In contrast to add_bonus, average represents a worst case for Carapace performance. The read of each grade value by `unwrap(secure_grade)` in Fig. 7a requires two checks (one for secrecy and one for integrity) that the grade's label (which contains a single tag) is a subset of the block's label, which contains 1,000 tags and is represented by a `HashSet`. These `HashSet` lookups account for the majority of the 286× slowdown reported in Table 2. In addition, a nontrivial fraction of the slowdown comes from the trusted secure block in Fig. 7a, even though it is outside the inner loop, since it performs expensive subset operations to check that the process's capabilities are sufficient to execute the block and that `classified_average`'s labels are subsets of the block's labels. These checks are expensive: Each check compares two labels that each have 1,000 tags.

8 Case Study: Avail

We retrofitted *Avail*, an open-source Rust application that computes available times across multiple calendars [2], to use Carapace. Avail accesses user-specified Google and Outlook calendars and returns times that are available across all calendars.

We used Carapace to enforce a secrecy policy that each calendar is secret to its owner. Each calendar c is assigned a unique secrecy tag c_{sec}, and thus c events have secrecy label $\{c_{sec}\}$. Since the calendars and their number are unknown at compile time, the tags and labels are dynamic.

8.1 Modifications to Avail to Implement the IFC Policy

Avail's primary computation calculates the available times across multiple calendars. We modified this computation to operate on `SecureValue`-wrapped calendars with dynamic secrecy labels. We modified Avail's handling of responses from calendar servers to wrap each calendar c's collection of events in a `SecureValue` with a dynamic label $\{c_{sec}\}$.

Avail relies heavily on a third-party Rust library called *Chrono* [19]. In general, application developers have two choices for a third-party library used by retrofitted code: Retrofit the library to use Carapace; or choose to trust the library, by declassifying/endorsing data sent to the library. We chose the former option because Avail relies so heavily on Chrono. As a result, we retrofitted both Avail and Chrono to use Carapace. Retrofitting Chrono involved marking its functions and types as side effect free, and refactoring to account for Carapace's restrictions on code in side-effect-free contexts, so Chrono's functions and types could be used in Avail's secure blocks. We chose to audit and trust certain uses of overloaded operators (which are disallowed in side-effect-free contexts; §5.4) in Chrono, by using a special trusted operation that provided by Carapace.

After collecting all events from the calendar servers, Avail groups them by date. Then it iterates over each day; for each day, it creates a vector representing the day's available times. The code iterates over each event in the day, adding an available time slot to the aforementioned vector whenever there is a gap between consecutive events. We initially tried wrapping the above code in multiple disjoint secure blocks, but Carapace did not allow it to compile due to noninterference errors. This process helped us realize that, to avoid certain illegal implicit flows, the entire computation on calendar events should be wrapped in a single secure block. So we wrapped the entire computation in a secure block, the end result of which is a vector of available times. Following this long secure block, we added a short *trusted* secure block to declassify this vector of available times.

8.2 Evaluation

To enforce the IFC policy described above, we added 313 lines and removed 165 lines of code from Avail. These changes affected 8 files. As part of these changes, we converted code to use `SecureValue`-wrapped types and secure blocks. In total we added 9 *untrusted* secure blocks.

To retrofit Chrono, we added 122 lines and removed 44 lines of code from the Chrono crate. These changes, which affected 17 files, consist entirely of marking types and functions as side effect free so they can be used in Avail's secure blocks.

Table 3. Run-time results for Avail. Run times are reported with 95% confidence intervals.

Statistics				**Run times**	
Secure blocks	(trusted)	Label comps.	Unwraps	Original	W/Carapace
10	(1)	27	19	310 ± 2 ms	310 ± 2 ms

Trusted secure blocks are a measure of how much code must be audited and trusted. In total our modifications include adding 1 trusted secure block to Avail, totaling 2 lines of code. This block is used to declassify the vector of available times computed by Avail.

To evaluate the performance effect of our changes, we compared the performance of unmodified Avail with the Carapace version of Avail. For each version of Avail, we measured the time to compute the shared availability between two different calendars (excluding the nontrivial time Avail spends on I/O). We ran 2,000 trials of each version of Avail.[5]

The right half of Table 3 shows the performance results. We did not detect any run-time overhead added by the Carapace version; with high confidence, the overhead is negligible (<1%).

The left half of Table 3 shows how many Carapace operations were performed by the Carapace version of Avail in the performance experiment (based on a separate, statistics-gathering run). The table reports counts of secure blocks executed, label comparisons, and unwrap operations. While these counts are small, we note that a significant fraction of Carapace-retrofitted Avail's computation happens in secure blocks on (unwrapped) secret data.

9 Case Study: Mk48.io

To evaluate the use of Carapace to support real-world integrity policies, we modified a real-world Rust application, the *Mk48.io* online multiplayer game [57], to use Carapace to enforce a dynamic integrity policy. Mk48.io is a web-based multiplayer 2D video game written mainly in Rust. In the game, each client controls a ship and attempts to shoot ships and upgrade their own ship.

MK48.io consists of two parts: (1) a client-side Rust program compiled to WebAssembly hosted on a local web browser and run by the player and (2) a server-side Rust program compiled to native code that manages the game map and communicating data between players. A client sends messages to the server consisting of orders to move a ship, fire weapons, or upgrade a ship. The server integrates these instructions into the game map and notifies other clients of these changes.

The server maintains the state of the game, such as the position and heading of each ship. A message from one client should only be able to affect the game state for the client ship, not for any other ship—which is an integrity policy. A violation of this policy would compromise the quality of the game. We use Carapace to enforce this policy by modifying the server program to treat incoming data from clients as low integrity. (All other data has default, or highest, integrity.) The implemented policy allows client data to impact server data by using an explicit endorse operation.

We modified Mk48.io's server program so that each client c is assigned a unique integrity tag t_c. Since the clients and the number of them are unknown at compile time, this tag is dynamic. Upon receiving a message from a client, the message's content is labeled with $\mathbb{U}_P^{dyn} \setminus \{t_c\}$, signifying that it comes from untrusted source c. Because trusted server data have the default label of $\mathbb{U}_P^{dyn}$, the untrusted client message is unable to affect server data without an explicit endorse operation.

[5]For both versions, around 0.2% of trials had very long run times (at least one second). Since these outliers are unrelated to our changes, we removed them from the results to avoid having much larger confidence intervals.

9.1 Modifications to Mk48.io to Implement the IFC Policy

We modified three sections of the Mk48.io codebase to use CARAPACE: (1) server-side client representation; (2) initial processing of client messages by the server; and (3) applying client messages to the world state.[6]

Server-side client representation. Each client is associated with a `ClientTuple` struct on the server, which uses the struct to identify which client sent a message. To provide each client with a unique tag for the server to use, we modified the `ClientTuple` struct to contain a dynamic integrity tag. Each tag is initialized along with the `ClientTuple`, and is uniquely associated with a single client.

Initial processing of client messages by the server. Messages are passed from client to server though a `Command` enum, which has variants for moving a ship, upgrading a ship, or creating a ship. When the server receives a `Command`, it calls a dynamically dispatched method to downcast it to the correct variant. The server then calls a function to apply the `Command`'s changes to the world state.

We modified the code the server uses to receive a `Command`, so that the first thing the server does with the `Command` is wrap it in a `SecureValue`, marked with the sending client's integrity tag. This marks the `Command` as low integrity, and guarantees that without an explicit endorsement, the untrusted client data cannot affect the server's world state. Additionally, because the CARAPACE prototype does not support dynamically dispatched methods in side-effect-free contexts, we refactored the code so it passes the `Command` through a `match` block to determine which enum variant it is. In each `match` case, the code calls the appropriate handler function as described next.

Applying client messages to the world state. The relevant handler function applies the changes from the `Command` to the world state. At each point where the world state is changed, we modified the code to use two secure blocks: (1) an untrusted block to compute the (still-low-integrity) modified data and (2) a trusted block to endorse the data and change the world state. Endorsement is necessary because the world state, representing the server's view of the game, has highest (default) integrity.

9.2 Evaluation

To enforce the IFC policy described above, we added 924 lines and removed 72 lines of code from Mk48.io's code base, affecting 16 files. As part of these changes, we converted code to use `SecureValue`-wrapped types and secure blocks, including 15 *untrusted* secure blocks.

We used 14 *trusted* secure blocks in Mk48.io, totaling 67 lines of code. Of these, one block identifies *which* variant a `Command` enum is—a reasonable endorsement that avoids us having to move a lot more logic into secure blocks. The other 13 trusted secure blocks endorse (low-integrity) client updates so they can be applied to (high-integrity) world state.

We did not do a quantitative performance evaluation since Mk48.io is an interactive game. We did test the experience of playing the game with multiple clients, both without and with CARAPACE. We perceived that the two versions behaved the same, with no change in functionality or performance.

10 Case Study: Servo

We retrofitted a large Rust application, the *Servo* web rendering engine [38], to use CARAPACE.

Servo background and IFC policy. Servo is a web rendering engine written primarily in Rust [38]. It is an open-source project initially developed by Mozilla starting in 2011. Servo is one of the 20 most popular Rust projects on GitHub, with over 20,000 "stars." It is a complex application, having over 1.15 million lines of Rust source code. We applied our changes to commit `d79876` of Servo.

[6]Mk48.io relies on an older version of Rust than the oldest version CARAPACE can use. The Mk48.io evaluation thus uses an older version of Rust, `1.65.0-nightly`, and a modified version of CARAPACE compatible with the older version of Rust.

First we describe the fine-grained security policy that we modified Servo to enforce using CARAPACE. The OS gives the Servo application process privileges to read data from web servers, write data to web servers, and receive input (e.g., keyboard input) from the user. However, we want to ensure that data entered into a page generated by one domain cannot leak to other domains. For example, credit card data (or data derived from it) entered into a form generated by https://icrc.org should *not* be allowed to be sent to another server such as https://evil.com. This confidentiality policy is comprehensive and helps avoid leaks of confidential data. (To be clear, this policy is not part of current Web standards and is violated by some existing benign web pages.)

Our target security policy generally considers user input to be secret, but it considers a few attributes of the input to be non-secret. First, the policy considers the *length* of input into a text box (i.e., the number of characters and the number of lines of text) to be non-secret. Second, the policy does not consider whether a key press is a "special" key (e.g., the Alt key) to be secret. Allowing these exceptions to the policy seems reasonable—we do not expect the "leaked" information to be harmful—and avoids a lot of additional propagation of secret values.

We achieve the target security policy using the following fine-grained IFC policy. Each web server's domain is assigned a unique secrecy tag. Since the domains and their number are unknown at compile time, the secrecy tag is dynamic. Let the dynamic secrecy tag for https://icrc.org be $icrc_{sec}$. If the user enters data into a web form, the data has a secrecy label consisting of the secrecy tag corresponding to the domain it was entered into. Data entered by the user into a form generated by https://icrc.org would have dynamic label $S_v^{dyn} = \{icrc_{sec}\}$. Data labeled with $\{icrc_{sec}\}$ can be sent to https://icrc.org but not https://evil.com (or any other domain).

Providing this IFC policy is complicated by the fact that when user input is delivered to the Servo process (in the form of raw keyboard input), it is not associated with a particular domain. The input data can however be labeled with the *user's* secrecy label. Since there is only a single user per Servo application invocation, the user's secrecy can be represented with a static secrecy label, $S_v^{st} = \{user_{sec}\}$, where $user_{sec}$ is a static secrecy tag for the user. Ideally, we could define labels so that S_v^{st} would be more secret than d_{sec} for *every* domain d. However, CARAPACE does not support this relationship, because CARAPACE's set-based IFC model does not allow mixing of static and dynamic tags. Instead, we modified Servo so that when the code processes the input data and determines that the input is being entered into a form generated by https://icrc.org, the code declassifies and reclassifies the input data by changing its label from $\{user_{sec}\}$ to $\{icrc_{sec}\}$.

10.1 Modifications to Servo to Implement the IFC Policy

The IFC policy ensures that all values affected by user keyboard input are secret, and can only be leaked at explicit declassification points. To support this policy, we modified Servo to use CARAPACE to label keyboard input data as secret. This involved wrapping secret values in `SecureValue` instances and enclosing computations on secret values in secure blocks. We modified three main components of Servo: (1) processing of keyboard events; (2) handling of text input; and (3) marshaling of text input, in preparation for sending to a web server. These components are represented in Fig. 9, which shows the flow of secret values through Servo. In the *Processing of keyboard events* component, secret values are labeled with the user's static secrecy label; our modifications use CARAPACE's static secrecy label `StatSecLabel::A` (Fig. 2a) to represent the user's static secrecy label, $\{user_{sec}\}$. In the *Handling of text input* and *Marshaling of text input* components, secret values are labeled with the dynamic label corresponding to the domain that generated the web form, e.g., $\{icrc_{sec}\}$.

To limit the scope of changes, we decided to trust Servo's networking component, which sends raw data to a web server, effectively making it part of the trusted computing base (TCB) as Fig. 9 shows. To further limit our changes, we decided to trust Servo's page-rendering components, which

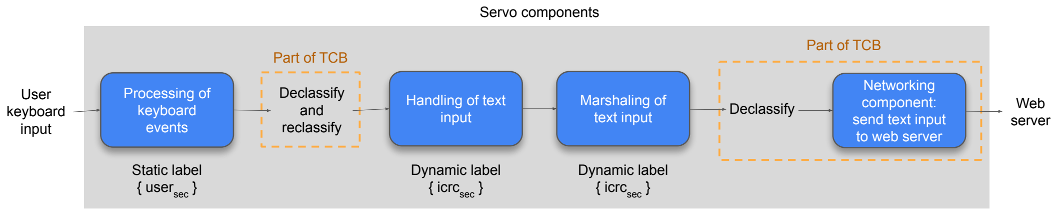

Servo components
User keyboard input
Processing of keyboard events
Static label { user$_{sec}$ }
Part of TCB
Declassify and reclassify
Handling of text input
Dynamic label { icrc$_{sec}$ }
Marshaling of text input
Dynamic label { icrc$_{sec}$ }
Part of TCB
Declassify
Networking component: send text input to web server
Web server

Fig. 9. An illustration of the flow of secret input data through Servo to a web server. The diagram excludes non-secret data and components through which no secret data flows.

display web pages including form data entered by the user (not shown in the figure). There were also several other places in the code—certain overloaded operator uses, `if let` expressions, and trusted third-party library calls—where we chose to bypass the secure blocks' side-effect-free checks using a special trusted operation provided by Carapace.

We also decided that supporting the propagation of secure data through Servo's JavaScript engine would be outside our scope. Otherwise, both the the JavaScript engine and the JavaScript API (Servo's implementation of calls from JavaScript programs into Servo) would have required pervasive changes. Specifically, a JavaScript program querying the content in an `HTMLInputElement` or `HTMLTextAreaElement`, or identifying the key inside a `KeyboardEvent`, would need to propagate secret values into the JavaScript engine, which cannot be supported by Carapace because the JavaScript engine used by Servo is written in C++. Instead of treating the JavaScript engine as a trusted component, which is arguably dangerous because it could leak secret data to arbitrary JavaScript code supplied by an untrusted server, we disabled use of the aforementioned JavaScript API.

Aside from the networking component, rendering components, and JavaScript engine, we did not trust or disable any other Servo component.

Processing of user keyboard events. Conceptually, when keyboard input is delivered to Servo from the OS, the data should be labeled with the user's secrecy label. Since our prototype does not label data coming from system calls, we modified Servo to wrap keyboard input data in `SecureValues`.

These modifications were complicated by the fact that keyboard data is initially created by a library called `keyboard_types` that is external to Servo. It provides keyboard data in the form of a `KeyboardEvent`, which in turn contains instances of other types—`Code`, `Key`, `KeyState`, `Location`, and `Modifiers`—all of which are defined in the `keyboard_types` library. Carapace requires that custom types be annotated as `InvisibleSideEffectFree` using the `#[derive(InvisibleSideEffectFree)]` annotation (§ 5.4), but Rust disallows annotating types defined in an external library. To address this issue, we added a "wrapper" type to Servo for each type whose instances are contained in `KeyboardEvent`: `CodeWrapper`, `KeyWrapper`, etc. We used `unsafe` code to implement `InvisibleSideEffectFree` for each wrapper type.[7] These changes allowed us to create a new keyboard event type containing secure values including `SecureValue<CodeWrapper, StatSecLabel::A, StatIntLabel::All>` and `SecureValue<KeyWrapper, StatSecLabel::A, StatIntLabel::All>`.

Handling of text input. In the middle of the flow from keyboard input to web server output is Servo's handling of text input (Fig. 9). Servo provides numerous functions to allow storing, processing, and querying text input. We modified these functions to use secure values and secure blocks.

In this part of Servo, a ubiquitous Servo-defined type is `DOMString`, which represents text input. The code often uses *deref coercion* (automatic invocation of a custom dereference operation by the

[7]We could not use `#[derive(InvisibleSideEffectFree)]` on the wrapper types, since that would still require the externally defined types to implement `InvisibleSideEffectFree`.

compiler) to access `DOMString`'s internal string data. However, deref coercion is syntactically invisible to Carapace's procedural macros and is thus disallowed by Carapace—specifically, it is not possible to implement `InvisibleSideEffectFree` for `DOMString` (§ 5.4). We thus eliminated `DOMString`'s custom dereference operation and changed accesses to be explicit.

Marshaling of text input and transmission to a web server. To submit form data to a web server, Servo marshals values of each form element into a container type appropriate for the submission method.[8] These values include both secure and ordinary values because the IFC policy is only enforced for text fields. To allow the container to hold both types of values without requiring all of its values to be wrapped in `SecureValue`, we added a `MaybeSecure` enum to Carapace-retrofitted Servo. Each `MaybeSecure` instance represents either an ordinary (unwrapped) or `SecureValue`-wrapped value, allowing the container to hold a heterogeneous mix of value types at run time.

Servo then sends the marshaled data to Servo's networking component, which transmits the data to the appropriate web server. Since we elected to trust the networking component, our modified code declassifies the text input data using the server's secrecy label before sending the data to the networking component. The result is fully declassified (i.e., empty secrecy label) and able to be sent to the networking component only if the data's secrecy label (e.g., $\{icrc_{sec}\}$) is a subset of the server's secrecy label (e.g., $\{icrc_{sec}\}$ or $\{evil_{sec}\}$). Otherwise, the trusted secure block panics.

10.2 Evaluation

To implement the IFC policy as described above, we added 3,029 lines and removed 636 lines from Servo's code base. These changes are spread across 37 modified and added files. These changes include converting code to use `SecureValue`-wrapped types and secure blocks. In total we added 101 *untrusted* secure blocks across Servo.

Trusted secure blocks are a measure of how much code must be audited and trusted. In total our modifications include adding 75 trusted secure blocks to Servo, totaling 196 lines of code. Of these, 41 trusted secure blocks are to send data to the trusted networking and page-rendering components. Another 29 trusted secure blocks declassify the length of the input data, and the remaining 5 blocks declassify boolean values derived from keyboard data, such as whether a key press is the Alt key.

Effectiveness. To test whether retrofitted Servo enforces the target security policy, we conducted two experiments. Each experiment uses two tests: a "good" test that does not violate the target security policy, and a "bad" test that violates the target security policy.

For the first experiment, we crafted proof-of-concept good and bad tests. In both tests, the user (of Servo) inputs data into a simple form on a static HTML document served from a local HTTP server. When the user submits, the good test's page sends the user data to the local server, while the bad test's page sends the data to a remote server. In unmodified Servo, both tests executed successfully, submitting data to either the local or remote server and displaying a response from it. In the Carapace version of Servo, the good test also executed successfully. However, in the bad test, Carapace's run-time checking detected an illegal declassification prior to sending data to the network, leading to a panic.[9] Thus Carapace correctly prevented the violation of the policy.

In the second experiment, the good and bad tests access TheOldNet (https://theoldnet.com), a JavaScript-free website that uses the Internet Archive to browse old web pages. For the good test, when TheOldNet loaded, we clicked on the first text entry box, which allows one to input a website URL to go to. We deleted the text currently in that text entry box and entered weather.gov without

[8] `vec[u8]` for `GET` requests, or a generic iterator over key–value pairs for `POST` requests.

[9] In our experiments, Servo panicked and terminated *non-silently*, despite Carapace calling `std::panic::always_abort()` (§ 5.3). We suspect Servo may be circumventing Rust's panic handling, e.g., by registering a trap handler with a system call (which would require `unsafe` code).

Table 4. Tests from Servo's Web Platform Tests that execute at least 5,000 unwrap operations.

	Statistics				Run times	
	Secure		Label		(± 95% confidence intervals)	
Benchmark	blocks	(trusted)	comps.	Unwraps	Original	W/Carapace
exec-command-with-text-editor.tentative.html	5,680	(2,104)	22,280	5,464	2,140 ± 15 ms	2,138 ± 14 ms
form-validation-validity-patternMismatch.html	5,561	(2,421)	21,546	5,561	2,780 ± 81 ms	2,735 ± 74 ms
form-validation-validity-tooLong.html	5,931	(2,573)	22,930	5,757	1,769 ± 10 ms	1,770 ± 10 ms
form-validation-validity-tooShort.html	5,947	(2,581)	23,010	5,781	1,780 ± 11 ms	1,781 ± 10 ms
input-untrusted-key-event.html	12,799	(1,101)	39,286	6,990	2,115 ± 29 ms	2,106 ± 27 ms
baseline-alignment-and-overflow.tentative.html	9,697	(4,654)	38,252	9,607	3,189 ± 68 ms	3,128 ± 26 ms
focus-dynamic-type-change-on-blur.html	11,384	(5,631)	45,374	11,463	2,246 ± 7 ms	2,248 ± 7 ms
type-change-state.html	12,731	(5,326)	48,260	12,901	1,933 ± 12 ms	1,900 ± 10 ms

changing the year we wanted to view the website from, which was 1996, before clicking the Go button to the right. Then, we scrolled to the bottom of weather.gov, and clicked the link titled The National Weather Service Home Page. In the page that opened, we clicked the SEARCH button, and in the text entry box that appeared on the loaded page, we entered the text "tornado" before pressing Enter. This test functioned without error in both unmodified and modified Servo.

For the bad test, when TheOldNet loaded, we deleted the text in the text entry box for Frog Find searching. We entered "weather" and clicked the Search Frog Find button. In unmodified Servo, this loaded a search page of various websites. In the Carapace version of Servo, this immediately generated a panic. Disallowing this behavior is correct because it violates the security policy: Data entered into a form flows from one domain (https://theoldnet.com) to another (http://frogfind.com).

Performance and run-time metrics. To measure the performance impact of Carapace on Servo, we compared the performance of unmodified Servo with the Carapace version of Servo using Servo's existing test suite, specifically its *Web Platform Tests*. Servo has about 30,000 such tests.[10]

We ran all of these tests with Carapace-modified Servo and found that only a few of them exercised Carapace's functionality substantially. In particular, the vast majority of the tests executed fewer than 20 unwrap operations. (We use unwrap operations as a proxy for Carapace functionality, since applications can only access secure values using unwrap operations.) However, eight of the tests stood out by executing more than 5,000 unwrap operations each. We ran each of these eight tests 1,000 times using both unmodified and Carapace-modified Servo to measure performance, and used separate runs to collect statistics. The runs used the same experimental methodology as the microbenchmarks evaluation (§7). Table 4 shows the results. The middle columns show how many unwrap operations, secure blocks, and label comparisons were executed.

The last two columns report run-time performance. For all tests except type-change-state.html, there is no significant run-time overhead added by Carapace (i.e., the confidence intervals are small and overlapping). This result is unsurprising considering the rate at which Servo performs Carapace operations: The tests last multiple seconds and perform only thousands of Carapace operations. We expect that many real applications would similarly perform Carapace operations sparsely. For type-change-state.html, Carapace-modified Servo has *lower* run time than unmodified Servo. This (repeatable) result may be an artifact of compiler or microarchitecture optimizations.

Three of these tests had subtests that only modified Servo failed. These failures occurred because Carapace does not allow secret values to be printed as strings, leading to unexpected output. The failures always occurred at the end of each subtest and thus did not affect behavior otherwise.

[10]About 4,000 additional tests are consistently skipped by Servo's testing infrastructure, so our results do not include them.

Evaluation of full test suite. We also ran all ~30,000 WPT tests and found our changes have limited impact on functionality, as reported in auxiliary material uploaded to the ACM Digital Library [10].

11 Related Work

This section discusses prior work not covered elsewhere in the paper (mainly in §2.3 and §5.3).

Following Cocoon [35], Carapace handles calls to Rust Standard Libraries by only allowing calls to functions identified as side effect free (§5.4). Hedin et al. consider the problem of how to model the effects of library calls, which can be modeled in a "shallow" way as in Carapace, or in a "deep" way by applying IFC analysis to the functions' implementations [28].

Prior work shows how to enforce fine-grained IFC for diverse targets. SecWasm provides IFC for WebAssembly [9] . BMJVM proposes IFC for a bare-metal JVM running directly on a hypervisor [40].

In contrast to fine-grained IFC, coarse-grained IFC tracks information flow at the granularity of processes, pipes, and other OS entities. Heule et al. show how to achieve coarse-grained IFC at the language level and demonstrate how to prove noninterference without relying on language features [29]. Vassena et al. show a general transformation between fine- and coarse-grained IFC [60]. Laminar melds fine-grained, language-level IFC and coarse-grained, OS-level IFC [47, 49].

IFC for JavaScript. A key issue with enforcing IFC for programs in dynamic languages like JavaScript is handling implicit flows. It is hard for analysis to determine which writes to public variables are dependent on secret values. Prior work shows that dynamic IFC can handle implicit flows soundly by terminating the program when a publicly visible effect may depend on a secret value [54]. Flow- and value-sensitive static analysis can enable more expressive IFC policies (i.e., less likely to terminate) [12, 51]. An iterative testing-based approach can identify places where the program terminates and prevent the same termination from occurring in future runs [13]. (Modeling common JavaScript operations such as those in the Document Object Model (DOM) interface can help with modeling explicit and implicit flows accurately [52].)

To detect implicit flows, *secure multi-execution* executes two versions of the program—one affected by secret values and one unaffected by secret values—and compares them [24]. Austin and Flanagan show how to support efficient secure multi-execution by optimizing for the common case when values do not differ between secret and public executions [5, 6]. Jaskelioff et al. show how to use secure multi-execution in a functional language, Haskell [30].

Prior work on IFC for dynamic languages has considered various enforcement mechanisms. Most work enforces IFC in the client-side JavaScript interpreter. Alternatively, enforcement can be performed in a just-in-time-compiled JavaScript VM to improve performance [31], or using server-side source-to-source translation on JavaScript programs to increase flexibility [8, 39].

As §5.3 described, Carapace addresses the challenges of identifying and tracking implicit flows by requiring accesses to secure data to be placed in uniformly labeled, lexically scoped blocks.

12 Conclusion

Carapace provides fine-grained, hybrid static–dynamic IFC for off-the-shelf Rust and its compiler. Carapace can provide comprehensive IFC policies in complex applications, as evidenced by retrofitting three real applications. This work advances the state of the art in software security by demonstrating that off-the-shelf support for comprehensive IFC policies is feasible.

Data-Availability Statement

An artifact containing Carapace, the microbenchmarks, the three retrofitted applications, and scripts to reproduce our evaluation is publicly available [11].

Acknowledgments

We thank Chujun Geng, Zhiqiang Lin, and Max Taylor for many helpful discussions; and the anonymous reviewers and Victor Chen for valuable feedback on the paper. This work is supported by NSF grants CSR-2106117 and SaTC-2348754.

References

[1] 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

[2] Mufeez Amjad. 2024. Avail. https://github.com/mufeez-amjad/avail

[3] 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

[4] Aslan Askarov and Andrei Sabelfeld. 2009. Catch me if you can: permissive yet secure error handling. 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, 45–57. https://doi.org/10.1145/1554339.1554346

[5] 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

[6] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/2103656.2103677

[7] 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), 94y99. https://doi.org/10.1145/3139645.3139660

[8] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. 2012. Secure Multi-Execution through Static Program Transformation. In *Formal Techniques for Distributed Systems*, Holger Giese and Grigore Rosu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–202.

[9] Iulia Bastys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. 2022. SecWasm: Information Flow Control for WebAssembly. In *Static Analysis*, Gagandeep Singh and Caterina Urban (Eds.). Springer Nature Switzerland, Cham, 74–103.

[10] Vincent Beardsley, Chris Xiong, Ada Lamba, and Michael D. Bond. 2025. Carapace – ACM Digital Library. https://doi.org/10.1145/3720427.

[11] Vincent Beardsley, Chris Xiong, Ada Lamba, and Michael D. Bond. 2025. *Carapace artifact*. https://doi.org/10.5281/zenodo.14915697

[12] Luciano Bello, Daniel Hedin, and Andrei Sabelfeld. 2015. Value Sensitivity and Observable Abstract Values for Information Flow Control. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 63–78.

[13] Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. 2012. Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing. In *Computer Security – ESORICS 2012*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.

[14] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. 2010. Unifying Facets of Information Integrity. In *Information Systems Security*, Somesh Jha and Anish Mathuria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 48–65.

[15] Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 463–475. https://doi.org/10.1109/ACSAC.2007.37

[16] 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

[17] Tianyu Chen and Jeremy G. Siek. 2024. Quest Complete: The Holy Grail of Gradual Security. *Proc. ACM Program. Lang.* 8, PLDI, Article 212 (jun 2024). https://doi.org/10.1145/3656442

[18] 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

[19] Chrono developers. 2024. Chrono: Timezone-aware date and time handling. https://crates.io/crates/chrono

[20] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for javascript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 50–62. https://doi.org/10.1145/1542476.1542483

[21] David D. Clark and David R. Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*. 184–184. https://doi.org/10.1109/SP.1987.10001

[22] 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

[23] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. 2024. Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 709–725. https://doi.org/10.1145/3694715.3695984

[24] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *2010 IEEE Symposium on Security and Privacy*. 109–124. https://doi.org/10.1109/SP.2010.15

[25] 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.

[26] 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.

[27] 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

[28] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. 2017. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *Principles of Security and Trust*, Matteo Maffei and Mark Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–70.

[29] Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. 2015. IFC Inside: Retrofitting Languages with Dynamic Information Flow Control. In *Principles of Security and Trust*, Riccardo Focardi and Andrew Myers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 11–31.

[30] Mauro Jaskelioff and Alejandro Russo. 2012. Secure Multi-execution in Haskell. In *Perspectives of Systems Informatics*, Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 170–178.

[31] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Information flow tracking meets just-in-time compilation. *ACM Trans. Archit. Code Optim.* 10, 4, Article 38 (dec 2013). https://doi.org/10.1145/2541228.2555295

[32] Dave King, Boniface Hicks, Michael W. Hicks, and Trent Jaeger. 2008. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *ICISS* (Seoul, South Korea). Springer-Verlag, Berlin, Heidelberg.

[33] 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

[34] 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

[35] 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 (apr 2024). https://doi.org/10.1145/3649817

[36] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. 2024. Implementation of Cocoon [35]. https://github.com/PLaSSticity/Cocoon-implementation

[37] 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

[38] Linux Foundation. 2023. Servo. https://servo.org

[39] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. 2010. On-the-fly Inlining of Dynamic Security Monitors. In *Security and Privacy – Silver Linings in the Cloud*, Kai Rannenberg, Vijay Varadharajan, and Christian Weber (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–186.

[40] Karthikeyan Manivannan, Christian Wimmer, and Michael Franz. 2010. Decentralized information flow control on a bare-metal JVM. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (Oak Ridge, Tennessee, USA) *(CSIIRW '10)*. Association for Computing Machinery, New York, NY, USA, Article 64. https://doi.org/10.1145/1852666.1852738

[41] Peng Li Yun Mao and Steve Zdancewic. 2003. Information Integrity Policies. In *Workshop on Formal Aspects in Security & Trust (FAST)*.

[42] 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

[43] 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

[44] 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

[45] 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.

[46] 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

[47] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. 2014. Practical Fine-Grained Information Flow Control Using Laminar. *ACM Trans. Program. Lang. Syst.* 37, 1, Article 4 (nov 2014). https://doi.org/10.1145/2638548

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

[49] 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

[50] 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

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

[52] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. 2009. Tracking Information Flow in Dynamic Tree Structures. In *Computer Security – ESORICS 2009*, Michael Backes and Peng Ning (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 86–103.

[53] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[54] Andrei Sabelfeld and Alejandro Russo. 2010. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In *Perspectives of Systems Informatics*, Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 352–365.

[55] 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.

[56] 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

[57] Softbear Games. 2024. Mk48.io. https://github.com/SoftbearStudios/mk48

[58] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2034675.2034688

[59] Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. 2016. Flexible Manipulation of Labeled Values for Information-Flow Control Libraries. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 538–557.

[60] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From fine- to coarse-grained dynamic information flow control and back. *Proc. ACM Program. Lang.* 3, POPL, Article 76 (jan 2019). https://doi.org/10.1145/3290389

[61] 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.

[62] 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

[63] 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

[64] 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 2024-10-16; accepted 2025-02-18