Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions

Kinan Dak Albab Artem Agvanian Allen Aby Corinn Tiffany
Alexander Portland Sarah Ridley Malte Schwarzkopf
Brown University

Abstract

Web applications are governed by privacy policies, but developers lack practical abstractions to ensure that their code actually abides by these policies. This leads to frequent oversights, bugs, and costly privacy violations.

Sesame is a practical framework for end-to-end privacy policy enforcement. Sesame wraps data in policy containers that associate data with policies that govern its use. Policy containers force developers to use *privacy regions* when operating on the data, and Sesame combines sandboxing and a novel static analysis to prevent privacy regions from leaking data. Sesame enforces a policy check before externalizing data, and it supports custom I/O via reviewed, signed code.

Experience with four web applications shows that Sesame's automated guarantees cover 95% of application code, with the remaining 5% needing manual review. Sesame achieves this with reasonable application developer effort and imposes 3–10% performance overhead (10–55% with sandboxes).

CCS Concepts: • **Security and privacy → Information flow control**.

Keywords: privacy enforcement; information flow control; privacy policies; Rust; static analysis; sandbox.

ACM Reference Format:
Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, Malte Schwarzkopf. 2024. Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3694715.3695984

1 Introduction

Modern web applications are subject to both self-imposed privacy policies and those required for compliance with privacy laws (e.g., GDPR [17], HIPAA [42], FERPA [18]). Inadvertent breaches of these policies can lead to significant penalties [7, 31–33]. For example, Instagram was fined €405M for accidentally disclosing children's email addresses and phone numbers [16]. Although Instagram had the correct internal policy and enforced it manually, developers overlooked an edge case where children had business accounts with public details. Similarly, a Google+ bug gave 438 third-party apps access to data from 500k Google+ accounts [20]; and LinkedIn accidentally leaked user data via its "auto-fill" feature [6].

These problems are difficult to avoid because developers today lack *practical* frameworks to ensure that their code abides by their privacy policies. It's easy for a developer to misremember which policy applies to data, or to forget to apply appropriate checks throughout the application code. To reduce this burden, developers need small and clear regions of privacy-critical code on which to focus their attention, and automatic guarantees for the remaining code.

Existing systems that seek to provide policy compliance guarantees face the challenge of enforcing complex and application-specific policies over an entire codebase. Classic approaches to compile-time enforcement require developers to aid the analysis, e.g., by reasoning about security labels or writing proofs [26, 27]. Dynamic approaches, by contrast, often require custom runtimes and either suffer from high overhead [46, 47] or limit application functionality [44].

Sesame is a new framework for writing web applications that guarantees *by construction* that the majority of application code upholds policies attached to data, and clearly highlights the remaining code that developers must audit. Sesame embraces key taint-tracking techniques from prior work on Information Flow Control (IFC) systems, but makes different tradeoffs to provide practical abstractions for developers. Sesame's key idea is to break the application into smaller, independent *privacy regions* that operate on sensitive data and "glue code" that connects these regions. This breakdown is possible because the Rust type system provides automated guarantees for the glue code, and it enables Sesame to apply a new hybrid approach to reason about privacy regions. Sesame checks privacy regions with a policy-independent static analysis, uses selective dynamic enforcement when static analysis fails, and taps into existing software engineering processes like code review as a fallback.

Sesame needs to prevent code outside privacy regions from accessing sensitive data, and must track the association between data and policy. Sesame addresses this need with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695984

policy containers. A policy container is a wrapper type that protects the wrapped data and associates a policy object with it. Sesame relies on Rust's encapsulation of private members to restrict access to the sensitive data, and on Rust's static type system to propagate the policy taint, even as the policy container moves through data structures and glue code.

Of course, application business logic eventually needs to compute on the sensitive data. To do so, the developer uses a Sesame privacy region, realized as a closure that has access to the raw data. Sesame unwraps data in policy containers passed to the privacy region and re-wraps any returned data. This allows Sesame to distill the problem of arbitrary policy enforcement into enforcing a fixed, policy-independent *leakage-freedom* property over privacy regions. In particular, a privacy region must not—directly or implicitly—leak sensitive data into captured or global variables, via system calls, or through native or unsafe code. Sesame applies static analysis to privacy regions to detect such leakage.

Because no existing static analysis for Rust covers this leakage-freedom property, Sesame contributes SCRUTINIZER, a new static analyzer that soundly rejects leaking privacy regions. Rust's mutability, ownership, and lifetime information enable SCRUTINIZER's analysis, which is difficult to do precisely in other languages. Verified leakage-free regions that pass SCRUTINIZER run as-is and without runtime overhead.

SCRUTINIZER's static analysis is sound but incomplete, so it may reject non-leaking privacy regions that call into complex-to-analyze code or libraries with native code. Sesame executes such regions in a sandbox that prevents leaks.

Naturally, applications eventually do intentionally externalize data, e.g., via database queries, HTTP RPCs, or emails. Sesame accommodates this via trusted Sesame-enabled libraries, which invoke policy checks before releasing data from policy containers. For custom I/O via arbitrary unsupported libraries, Sesame provides *critical regions*. A code reviewer must manually review and sign each critical region. Sesame's design makes critical regions infrequent (≤5% of code), slim (≈16 LoC), and clear to the reviewer; and Sesame enforces new reviews when the underlying code changes.

Sesame targets honest developers who make unintentional mistakes and assumes that sanctions deter developers from malicious behavior. Thus, timing and side-channel attacks are out of scope. The Rust compiler, SCRUTINIZER's static analysis, the sandbox, and Sesame-provided libraries are trusted, and Sesame's guarantees for critical regions rely on good-faith and attentive code review. Developers should use Sesame-provided libraries to interact with external entities, such as an HTTP client or a database, to avoid frequent critical regions. Such mandatory use of specific libraries is already common practice in organizations today.

We evaluated our Sesame prototype [1] with four Rust web applications. Our experience suggests that Sesame can express a wide variety of policies and requires limited developer effort, and that effort is focused on critical regions that need careful attention. In our case studies, Sesame's automated guarantees cover 95% of application code, including the vast majority of privacy regions. Sesame's policy checks add 3–10% runtime overhead in the common case, while sandboxed regions have higher overhead (10–55%).

In summary, this paper makes four contributions:

1. Sesame, a practical framework that enforces policy compliance over data by construction, by breaking the application into glue code and privacy regions.
2. A new approach that composes Rust's guarantees with hybrid static/dynamic enforcement of a single, policy-independent leakage-freedom property.
3. A novel static analysis that checks Sesame's leakage-freedom property over privacy regions.
4. Dynamic enforcement of Sesame's guarantees using sandboxing where static analysis fails, and a design that focuses developer attention on critical regions that require human review.

2 Background and Related Work

Sesame builds on a rich literature on enforcing policies over data. While inspirational, these prior approaches have failed to gain traction because they are impractical or costly.

Information flow control (IFC) enforces end-to-end security policies in programs. IFC systems may rely on enforcement via runtime labels [25, 47–49], compile-time enforcement via type systems [27, 41] and static analysis [13], or on hybrid approaches [4, 10, 29, 36]. A common failure point of IFC systems is that they are challenging for developers to use [15]: some require use of custom languages unfamiliar to web developers [4, 27, 29, 35], and others require developers to propagate complex security labels [48, 49]. Static approaches can only express limited policies that developers must encode [26, 39, 40], and dynamic approaches often come with steep performance costs [46, 47].

Compile-time policy enforcement statically guarantees policy compliance. Application developers define their policies in a single, reviewable location, and associate them with data. The compiler then checks that the program's execution cannot violate policies. Policies can range from classic IFC non-interference (e.g., enforced through Rust types in Cocoon [26]) to more complex policies over SQL databases and their schemas (e.g., in Ur/Flow [8], or via refinement types in Storm [27]). The latter approaches can express data-dependent policies, but only if they are exclusively defined via relationships over the database schema. They also require writing applications in specialized languages, such as Ur/Web [9] or LiquidHaskell [27], which limits adoption.

Dynamic policy tracking attaches policies to data, and tracks these associations with a modified language runtime. The runtime maintains a policy taint for every variable, propagating and combining the policy taints as the program runs. Developers write policies declaratively over a data model [46] or attach them dynamically to data [47], and the

```rust
fn submit_homework(request: Request) {
  let uid = authenticate(request);
  // Find the email of the authenticated user
  let email = DB.lookup(uid);
  // Get the homework answer from the request
  let answer = request.answer;
  DB.insert(answer);

  // Format email body
  let body = format!("submitted {}", answer);







  // Email access control implicit
  email::send(email, body);

}
```

(a) Without Sesame.

```rust
fn submit_homework(request: SesameRequest) {
  let uid = authenticate(request);
  // email: PCon<String, ...>
  let email = SesameDB.lookup(uid);
  // answer: PCon<String, AnswerAccessPolicy (§4.1)>
  let answer = request.answer;
  SesameDB.insert(answer);
  // body: PCon<String, AnswerAccessPolicy>
  let body = sesame::privacy_region(answer,            // §6.1
    |raw_answer: String| format!("submitted {}", raw_answer)
  );
  // Sesame checks that email meets AnswerAccessPolicy.
  let context = Context { email };                     // §4.2
  sesame::critical_region(body, context,               // §6.3
    #[signed(Kinan Dak Albab <babman@brown.edu>, 7459f3da..)]
    |raw_body: String, context: Context::Out| {
      // Reviewer checks context.email used as recipient
      email::send(context.email, raw_body);
    });
}
```

(b) With Sesame, privacy regions highlighted.

Figure 1. An HTTP endpoint for submitting homework answers, implemented (a) without Sesame and (b) with Sesame. Using Sesame, the developer must use privacy regions to operate on data in PCons. Sesame verifies the first region via static analysis (green), but the region that sends an email requires a critical region (yellow), which a code reviewer must sign.

runtime checks the associated policy before application sinks externalize tainted data. Runtime tracking imposes high performance overheads (e.g., 33% [47] to 75% [46]) and requires using interpreted languages, but minimizes developer effort.

Sesame targets end-to-end enforcement of flexible, data-dependent policies for applications written in a widely used, mainstream programming language (Rust). This means that Sesame can express richer policies that rely on dynamic information about the data or application, and makes it easy to adopt. Sesame achieves this without a custom taint-tracking runtime, in the presence of third-party libraries, with limited extra burden for developers, and at low overhead.

3 Sesame Overview

Sesame is a framework for web application development. To illustrate how developers use Sesame, consider how a developer might write an HTTP endpoint for students to submit their answers in a homework submission application (Figure 1a, without Sesame). The code authenticates the user (line 2), retrieves their email address from the database (line 4), inserts the student's homework answer into the database (lines 6–7), and sends the student a confirmation email via a third-party library (lines 10 and 18).

This endpoint handles two types of user data: the student's email address and their submitted answer. Each type might be governed by a different policy. For example, a policy for the answer might allow only the student themselves, TAs, and the instructor to view the submission.

We now look at how the developer uses Sesame to implement this endpoint with compliance guarantees (Figure 1b). The developer invokes Sesame-enabled libraries, as mandated by their organization, to look up the email address in the database (line 4) and to access the answer in the HTTP request (lines 1, 6). As these libraries are Sesame-enabled, they return data wrapped in a *Policy Container* (PCon; see §5). They also accept PCons as input, e.g., to insert the PCon-wrapped answer directly into the database (line 7). A PCon keeps the underlying data private and inaccessible to the application and associates it with a policy: e.g., the answer is protected by an `AnswerAccessPolicy`. §4.1 explains how developers write policies and associate them with data.

Now, the developer needs to construct the email body. This is application-specific functionality unavailable in a Sesame-enabled library, and it operates on the answer data. However, accessing the answer directly causes a compiler error, as the answer is a private member in a PCon. Instead, the developer must use a privacy region. They invoke Sesame's `privacy_region` API, passing the PCon along with a closure to execute on the raw answer (lines 9–11). Sesame's static analysis (§6.1) verifies that the closure is leakage-free, so it runs as-is on the raw data. Sesame wraps the output of the closure in a PCon with an identical policy to the input.

Finally, the developer emails the body they constructed to the student using a third-party email library. The contents of `body` are inside a PCon, so the developer could again invoke `privacy_region` with a closure that sends the email.

However, this closure intentionally leaks data via email, so it is clearly not leakage-free. The developer informs Sesame of this by instead using a *critical region (CR)* (lines 14–19). Before executing a critical region, Sesame checks the associated policy relative to a developer-provided *context*. In our example, this context contains the recipient's email address, and Sesame checks that `AnswerAccessPolicy` allows sending the answer to that email address. A code reviewer, e.g., a team lead or a policy engineer, must now manually review the critical region and ensure that it sends the email to the address in `context` that passed Sesame's policy check. §4.2 describes Sesame's policy check and the review workflow.

In general, reviewers must verify that the CR uses the context and acts in ways that are consistent with it, and then sign the region. During a release build, Sesame validates the signature (§6.3). If the CR's code or any of its dependencies change, validation fails. If the developer failed to see the need for a CR on line 14 and used `privacy_region` instead, Sesame's static analysis would reject this privacy region. The developer then would have to decide whether they believe the rejected region to be leakage-free. If so, they can run it in a sandbox (§6.2); if not, they use a CR as shown.

Developers may implement policies concurrently with the application, or use placeholder policies first and then add the policy logic later; §8 discusses porting existing applications.

4 Design

Sesame's policy enforcement is concerned with the application's sources and sinks, which correspond to where data enters and leaves the application. When the application reads from a source, Sesame attaches a policy to the data. Sesame's design then ensures that policy remains attached to the data, including derived data, as it flows through the application (i.e., taint-tracking). Sesame only allows data to leave the application at a sink if its associated policy check succeeds.

Sources. Sesame provides built-in support for common sources such as HTTP requests and SQL databases. Sesame reads data from those sources, places it in a policy container (PCon), attaches the appropriate policies to it, and returns it to the application. For example, Figure 2 shows an endpoint for students to view their homework answers. `answer_id` comes from a Sesame HTTP source (line 4), and `rows` comes from a Sesame database source (lines 7–10), so the application receives data in PCons. Attempts to read raw data from these sources will cause a compile-time type error.

Sources not in Sesame-enabled libraries provide raw data, which developers must manually place in policy containers and associate with policies. Organizational rules are responsible for ensuring that developers do this correctly: e.g., requiring developers to use a custom I/O module that associates policies to the data, rather than reading files directly.

Sinks. Similarly, Sesame has built-in support for common sinks, such as rendering HTML templates (line 15), or passing data to the DB for reads (line 9) or writes. Sesame receives policy containers from the application, checks the associated policies, and sends the data to the sink if the policy permits. Releasing data to custom sinks unsupported by Sesame requires critical regions and code review. For these sinks, developers explicitly provide Sesame a *context* that describes the intended use. Sesame checks that the policy associated with data accepts the context before running the critical region that touches the custom sink. Code reviewers must review and sign CRs to ensure that they match the context.

```
1  #[sesame::get("/view/<answer_id>", auth="student")]
2  pub fn view_answer(
3    student: Student, // authenticates via cookie
4    answer_id: PCon<u8, NoPolicy>,
5    context: Context) -> sesame::HTMLTemplate {
6    // rows : Vec<sesame::PConRow>
7    let rows = SesameDB.query(
8      "SELECT * FROM answers WHERE id = ?
9       AND author = ?", (answer_id, student.email),
10      context);
11   // answer: PCon<String, AnswerAccessPolicy>
12   let answer = rows[0]["answer"];
13   // Render the answer. Sesame automatically checks
14   // associate AnswerAccessPolicy here.
15   sesame::render("answer.html", answer, context)
16 }
```

Figure 2. An endpoint for students to view their answer uses `answer_id` from the HTTP request to look up the answer in the DB. Sesame only reveals the PCon-wrapped answer if the policy allows the signed-in student to view it.

4.1 Policies

Developers express each policy type as a Rust struct and implement Sesame's Policy trait for this struct. This trait requires providing a check function that Sesame invokes before revealing data associated with that policy at a sink. The check function may use the context as well as metadata stored inside the policy itself, and can execute arbitrary code (e.g., query a database). If the policy check fails, Sesame reports an error; otherwise, it releases the data to the sink.

Figure 3 shows a policy for student homework answers. The policy allows revealing an answer only to its author, the instructor, or students who are discussion leaders for the lecture (identified via a database query).

Associating Policies with Data. Application developers must associate policies with the data read from application sources. Developers must explicitly associate insensitive data that is *intentionally* not covered by any policy with NoPolicy (e.g., Figure 2, line 4). For sources with a structured schema, application developers specify the policy associations decoratively for that schema: e.g., Figure 3 associates `AnswerAccessPolicy` with the `answer` column in

```rust
#[db_policy(table = "answers", columns = ["answer"])]
struct AnswerAccessPolicy {author: String, lecture: u64}
impl Policy for AnswerAccessPolicy {
  fn check(&self, context: Context::Out) -> bool {
    let email: String = context.email;
    email == self.author || is_instructor(email)
      || is_discussion_leader(email, self.lecture)
  }
  fn join(self, other: Self) -> Self { ... }
}
impl DBPolicy for AnswerAccessPolicy {
  fn from_row(row: &MySQLRow) {
    AnswerAccessPolicy {
      author: row.get("author"),
      lecture: row.get("lecture_id"),
    }
  }
}
```

Figure 3. The CHECK function (lines 4–8) of `AnswerAccessPolicy` only allows sending an answer to an email, from `context.email` (line 5), if it matches the answer's author, the instructor, or a discussion leader. Line 1 binds the policy to the `answer` column; lines 12–16 instantiate it.

table `answers` (line 1). Applications declare the associated policies when they read data from unstructured sources, such as a cookie or GET parameter. Developers must implement constructors to create instances of a policy for each type of source to which they attach the policy. Since Sesame trusts policy code, it passes raw data to policy constructors.

Policy Conjunction. Sesame combines policies when combining PCons, e.g., when executing a privacy region over a vector of PCons. If the policies have different types, Sesame generically combines them by *stacking* them. The stacked policy stores all source policies, and checks all of them in its CHECK function. If all the policies have the same type, Sesame combines them using a policy-specific JOIN, if policy developers implement one (Figure 3, line 9). Joining and stacking must be semantically equivalent, but joining may result in more compact policies that are faster to check.

4.2 Context and Policy Checks

Sesame invokes policy checks when: *(i)* a policy container reaches a Sesame-enabled sink; and *(ii)* before running a critical region on the data in a PCon.

Like in prior systems, Sesame policy checks happen relative to a *context.* Contexts contain summary information about the associated source or sink. They are immutable objects, either provided by Sesame or created by the developers themselves. The content and API of a context is application-specific: e.g., the homework submission example identifies users by emails, but other application contexts might contain user IDs or OAuth tokens. Contexts can store sensitive information in PCons, which Sesame replaces with raw data when invoking CHECK and critical regions (i.e., CONTEXT::OUT; see §5 SESAMETYPE).

Contexts created by Sesame are trusted, e.g., they correctly identify the authenticated user. Developers may pass them to built-in Sesame sinks, which use them for policy checks without any developer intervention (Figure 2, line 15). By contrast, Sesame only allows custom contexts with custom sinks, and relies on manual review to ensure the sink's behavior is consistent with the context.

Example. Figure 1b contains a critical region to send emails (a custom sink). The application creates a custom, untrusted context that indicates the *intended* recipient of the email (line 13). The application then invokes a critical region on `body` with that context (line 14). Since `body` is associated with `AnswerAccessPolicy` (line 8), Sesame first invokes that policy's CHECK function (Figure 3, lines 4–8) with the provided context, and executes the region only if that check succeeds. CHECK ensures that the `email` provided in the context belongs to the author, the instructor, or a discussion leader. A reviewer must ensure that the critical region indeed sends an email to the intended address (Figure 1b, line 18).

A buggy implementation might violate this policy in two ways. The application could provide Sesame with a context that contains an unauthorized email (Figure 4a). This causes the policy check to fail, so Sesame never executes the critical region. Or the application may provide a context with an authorized email when the code in the critical region sends an email to a different address (Figure 4b). This is an example of a custom context that misrepresents the sink. Here, the policy check would succeed, but a careful reviewer refuses to sign the critical region, because it does not email `context.email`, and Sesame errors on a release build.

Importantly, the reviewer merely verifies that the critical region uses `context.email`, which passed a policy check, and therefore Sesame guarantees it is authorized.

4.3 Guarantees and Threat Model

Sesame views policy definitions as ground truths that specify desired application behavior. This includes each policy's CHECK and JOIN functions, constructors, and associations with data at sources. Sesame also relies on the soundness of the Rust type system, its compiler, and the correct implementation of Sesame components, all of which are trusted. Subject to the assumptions below, Sesame guarantees that data can only be revealed at a sink whose context passes the policy check associated with the data's origin.

Proper Usage. Organizations must mandate the proper use of Sesame, e.g., via code review, linters, or other best practices. Specifically, developers must *(i)* use Sesame's built-in sources when applicable, *(ii)* correctly wrap data from

```rust
1 let context = Context { email: "someone@else.com" };
2 sesame::critical_region(body, context,
3   // Sesame invokes policy check with context prior
4   // to calling the closure. Policy check fails.
5   #[signed(..., 7459f3da..)]
6   |raw_body: String, context: Context::Out| {
7     email::send(context.email, raw_body);
8   });
```

(a) A signed critical region correctly uses its context, but the email in the context is unauthorized and Sesame's policy check fails.

```rust
1 let context = Context { email: answer.author };
2 sesame::critical_region(body, context,
3  // Policy check would pass on this recipient, but the
4  // region uses an email address not from the context.
5  // Reviewer refuses to sign region.
6  |raw_body: String, context: Context::Out| {
7    email::send("someone@else.com", raw_body);
8  });
```

(b) Sesame's policy check passes on an authorized email, but the critical region mismatches its context, so the reviewer rejects it.

Figure 4. Buggy alternative implementations for Figure 1. Sesame rejects both via policy checks (a) or code review (b).

custom sources in policy containers, and *(iii)* compile their applications with Sesame's toolchain.

Accurate Review. Sesame assumes that reviewers carefully vet critical regions and only sign them after ensuring that the regions are consistent with their specified contexts.

Unsafe Rust. Rust guarantees encapsulation for applications that operate solely within safe Rust. However, applications often, directly or in dependencies, invoke unsafe code for which Rust provides no encapsulation guarantees. Sesame implements additional protections to ensure data in PCons remains inaccessible even to unsafe Rust code (§5), assuming that such code is buggy but not outright malicious. For example, Sesame protects against a logging library that uses unsafe code to byte-wise dump provided objects, but not unsafe code that dumps the entire memory of the application process (in line with other IFC systems [27, 29, 47]).

Implicit Leakage. Sesame protects against direct leakage and implicit data-dependent control flow. Application code cannot perform control flow on data wrapped in policy containers without using privacy regions, which have mechanisms for mitigating data-dependent control flow (§6). This also ensures that observable data-dependent interactions (e.g., with a database) only occur following a successful policy check. The one exception is certain Fold APIs that Sesame provides for ergonomic reasons, which can leak information about the shape of some data types. Developers can disable them for data with restrictive policies (§5). Timing and micro-architectural side channels are out of scope.

5 Policy Containers

Sesame's policy containers are a generic data type, PCon<T, P>, that wraps two private members: data of type T, and a policy object of type P. The data wrapped by a PCon is private and can only be accessed or manipulated by Sesame and never by application code, except through a privacy region. This guarantees application code cannot leak such data without going through Sesame's checks. Using policy containers also guarantees that a policy associated with data at a source remains associated with that data, including derived data, throughout the application.

PCons are regular Rust data types. Application code can pass or return PCons to and from functions. It can store PCons inside vectors and other collections, and it can define structs that have PCon fields. While PCons simplify policy tracking and checking, they prevent application code from directly computing over the wrapped data. Therefore, a core challenge in Sesame is to provide application developers with tools and abstractions to allow them to operate on data inside PCons. PCons provide builtin functions for common operations, such as type conversions (e.g., from a PCon<u32, P> to a PCon<String, P>). Developers must use privacy regions to perform more complex tasks on data in PCons.

PCon Layout. Rust guarantees that safe Rust code cannot access the private members of a PCon. However, unsafe Rust code can circumvent these protections by using casts or accessing the bytes of the policy container directly. Library code sometimes does this for legitimate reasons, e.g., a data structure that uses `memcpy` for efficient resizing, or a logging library that dumps the bytes of arbitrary objects. To ensure that such libraries cannot accidentally leak the data wrapped by PCons, Sesame stores the data on the heap and references it within the PCon using an obfuscated pointer. Sesame XORs this pointer with a random global secret. This prevents unsafe application or library code from accidentally leaking the data, but cannot protect against actively malicious code that intentionally undoes the obfuscation or scans memory contents, both of which are out of scope. Finally, this layout has performance implications: operations on the PCon require an additional XOR and pointer dereference, and vectors and collections of PCons have worse cache locality than their plain counterparts. These overheads affect wrapping and unwrapping data at privacy region entry and exit, but not the body of a privacy region, which accesses the data directly. This indirection adds a 1.7–2.1× overhead in microbenchmarks, but is negligible for real workloads (§9).

SesameType. Sesame handles types with arbitrary nested PCons, such as Vec<PCon<T, P>> or a custom struct with

API Level	Supports	Guarantees	Root of Trust	Application Developer Effort
Built-in	Common primitives	Static (taint tracking) Runtime (policy checks)	Rust + Sesame	Use PCon<T, P> instead of T with compiler guidance
Verified Region (VR)	Statically-verified leakage-free closures	Static analysis (sound but incomplete)	Rust + Sesame	Check that Sesame accepts closure as leakage-free
Sandbox Region (SR)	Leakage-free closures that cannot be verified statically	Runtime (sandbox)	+ RLBox [30]	Engineering setup
Critical Region (CR)	Arbitrary closures including sinks	Code signing	+ reviewers	Authorized developers review and sign closure

Figure 5. Sesame's API levels. Built-in libraries and verified regions require a minimal root of trust. Sandboxed regions support more complex code at the cost of runtime overhead. Critical regions support arbitrary code, but rely on code review.

nested PCon fields, by relying on the SesameType trait. For each type X that implements it, this trait defines the corresponding out-type $X::Out$. $X::Out$ mirrors the structure of X but replaces every nested PCon<T, P> with the corresponding T. The trait also defines private conversion functions between the two types, which only Sesame can invoke. Sesame implements this trait for various monads, tuples, and collections that may contain PCons, e.g., Vec<*X*> and Option<*X*> with out-types Vec<*X*::*Out*> and Option<*X*::*Out*>. Because this trait deals with raw data, Sesame disallows applications from manually implementing it for their custom types (enforced via custom lints, §7). Instead, Sesame provides a macro that applications can use to derive trusted implementations of this trait and its out-type for their custom types.

Fold. For better ergonomics, Sesame provides a fold API, which allows applications to move PCons in and out of data structures. Suppose the application has d: PCon<X, P> where X is some application struct containing $\{a\colon T1,\ b\colon T2\}$. Applications can use fold(d).a to retrieve the field a: PCon<T1, P> ("folding in"). Alternatively, applications can fold PCons "out", e.g., to transform Vec<PCon<T, P>> to a PCon<Vec<T>, P> whose policy is the conjunction of all the input policies. Folding out is always safe, but folding in may leak information about the shape of the underlying data, e.g., the length of a vector or whether an `Option` is None or Some. If undesirable, policy developers can annotate a policy with NoFoldIn to prohibit folding in on its associated data.

6 Privacy Regions

Sesame provides different ways for applications to operate on data wrapped by PCons. Figure 5 summarizes them: "built-in" describes Sesame-enabled libraries, while the other three API levels correspond to privacy regions. Privacy regions allow application code to execute on the raw data. Sesame's static analysis helps the developer determine what type of privacy region to use. At a high level, the analysis checks that a closure cannot leak input or derived data, e.g., by writing to a file or modifying a global or captured variable.

Depending on the static analysis outcome, different types of privacy regions are appropriate:

1. If the static analysis passes, the privacy region is a *verified region (VR)* and runs as-is (§6.1).
2. If the static analysis fails, but the developer expects the region to be leakage-free, they can choose to use a *sandboxed region (SR)*, which runs the code in a constrained environment that prevents leakage (§6.2).
3. If the developer expects a failing region to have intentional side effects (e.g., because it interacts with a custom sink), they use a *critical region (CR)*, which requires manual code review (§6.3).

In general, the `privacy_region` APIs accept a SesameType X and a closure $F\colon X::Out \to Y$ as arguments. Sesame executes the closure over the input after replacing PCons with their underlying data. Sesame wraps the result of the closure in a PCon, and associates it with the conjunction of all the policies in the input. Critical regions are the exception; they can return data with a different policy or no policy at all.

6.1 Static Analysis and Verified Regions

Sesame's static analyzer, Scrutinizer, checks whether a closure passed to a privacy region could leak some of its arguments outside the region. Scrutinizer is sound but incomplete: it never accepts leaky privacy regions, but may conservatively reject leakage-free ones.

Scrutinizer searches the application code for instances of Sesame's `privacy_region` call. We refer to the closure passed to the privacy region as the *top-level function*. Scrutinizer considers each argument to the top-level function to be *sensitive*. Top-level functions may also capture external variables from their environment, but captured variables are not sensitive. Scrutinizer accepts a function only if it concludes that the function cannot leak any of its arguments or any data derived from them. This includes leakage via external side effects (e.g., printing to stdout, changing the file system) or via mutating captured variables that other parts of the application can observe (e.g., global variables). Scrutinizer checks the top-level function and its callees, direct or indirect, including those in external libraries. Top-level functions can return data derived from their arguments since Sesame wraps the return value in a PCon.

Information Flow. Scrutinizer computes a sound over-approximation of the information flow of the arguments and captured variables in the top-level function all the way through call chains to helper and library functions. Scrutinizer uses Flowistry [12] to find flows from a sensitive variable to any aliases within a single function body. We extend Scrutinizer with additional analysis to track sensitive variables as they are passed to and returned from other functions. Scrutinizer uses dataflow analysis to soundly resolve dynamic dispatch with good accuracy. Thus, Scrutinizer identifies all aliases or variables derived from sensitive variables, either directly or implicitly via control flow. Scrutinizer considers all such variables to be sensitive as well.

Analysis. Scrutinizer checks that the information flow of sensitive variables is contained entirely within the analyzed code. Within Sesame's threat model, information can flow outside a function in three ways:

1. via mutably captured variables or variables derived from such captures;
2. via any unsafe mutation primitives applied to captured variables and variables derived from captures, regardless of their mutability;
3. via functions with unknown bodies that Scrutinizer cannot conservatively approximate, such as native code or unresolved generics, unresolved dynamic dispatch, and unresolved function pointers.

Scrutinizer catches all three cases. Mitigating the first case ensures that the function cannot mutate external variables in ways that depend on (and thus leak) sensitive arguments. Mitigating the second case covers all forms of interior mutability in Rust, which ultimately rely on unsafe mutation (via transmute or raw pointer dereferences). Mitigating the third case ensures that Scrutinizer rejects functions that leak sensitive data via external side effects, such as writing to files or sockets, as they must invoke native code.

Allow list. Scrutinizer allow-lists some trusted functions, including certain Rust intrinsics and low-level functions for string formatting and panics. We manually confirmed that these functions are leakage-free.

Scrutinizer also allow-lists functions in standard library collections (e.g., `Vec::push`) that take the `Self` parameter as a mutable reference. This is sound, as invoking such a function on a captured collection would require a mutable reference to it, which can only be acquired by mutably capturing it (violating case 1) or via an unsafe conversion from an immutable capture to a mutable one (violating case 2). Since Scrutinizer rejects such code, these functions can only be called on local variables, which external code cannot observe. The only remaining risk is the allow-listed functions directly leaking arguments (e.g., by writing to a file), which standard library collections do not. This assumption makes `std::collections` part of Sesame's TCB, an acceptable risk in practice.

Details. Scrutinizer first collects Rust's MIR representation of all available function bodies via `rustc` dataflow analysis framework, including all possible variants for dynamic dispatch. Second, Scrutinizer ensures all captures are immutable and then uses Flowistry to track information flow through the collected call tree. If it encounters any violations of cases 1–3, Scrutinizer rejects the privacy region. Appendix A describes the analysis in more detail.

Discussion. Scrutinizer is sound but incomplete for three reasons. First, Scrutinizer conservatively rejects functions if it fails to resolve their information flow in its entirety. For example, a leakage-free function will be rejected if it contains dynamic dispatch that Scrutinizer cannot resolve. Second, Scrutinizer checks for stronger (but easier to detect) variants of the three cases above. For example, Scrutinizer rejects *all* functions that capture via mutable reference, even if they never mutate them based on sensitive values. Third, Flowistry itself over-approximates information flow [12].

Scrutinizer uses Flowistry to propagate sensitivity labels within a function body, but it contributes new dataflow and type analyses that *(i)* propagate labels across functions, *(ii)* handle unsafe code, generics, and dynamic dispatch, and *(iii)* detect mutation, including in implicit, data-dependent ways.

6.2 Sandboxes

Developers may choose to run a region that Scrutinizer rejects as a sandboxed region, which relies on runtime protections to enforce that the region never leaks sensitive data.

Sesame's sandboxed regions use RLBox [30], a lightweight sandbox used in Firefox to execute untrusted native libraries. RLBox relies on software-based fault isolation (SFI), which uses inline dynamic checks to restrict memory reads and writes to a memory region allocated at sandbox creation time. This isolates the sandboxed region's memory from the rest of the application. In addition, RLBox forbids system calls, so sandboxed regions cannot externalize data via I/O.

Extending RLBox. RLBox is designed to isolate untrusted libraries from a trusted host application (Firefox). Hence, RLBox allows the application to access the sandbox's outputs and lets the sandbox print to stdout and stderr for debugging. In Sesame, the application is untrusted and the sandbox must not leak any sensitive information to it. We thus align RLBox with Sesame's requirements by: *(i)* modifying the RLBox runtime to forbid printing, and *(ii)* building infrastructure around sandbox invocations to compute the conjunction of all policies associated with the sandbox inputs and wrapping the sandbox output in a PCon with the conjoined policy.

Optimizations. RLBox allocates the entire sandbox memory on sandbox creation, which makes creating and destroying sandboxes expensive. Firefox overcomes this by reusing the same sandbox for invocations in the same trust domain (i.e., the same library and HTTP origin). Sesame sandboxes

process data with different policies, making such reuse unsafe: earlier invocations with stronger policies could affect sandbox state that influences later invocations with weaker policies. Instead, Sesame uses a pool of pre-allocated sandboxes, and wipes the sandbox memory after each use to ensure isolation across invocations. This involves zeroing out the sandbox stack and heap, and restoring global data and metadata to their initial state from a checkpoint.

Because of sandbox memory isolation, Sesame must copy all input data into the sandbox memory, and vice-versa for its outputs. However, the same datatype may have an incompatible size and memory layout across the application and sandbox, as RLBox runs sandboxes in 32-bit WASM and offsets its address space for isolation. For example, a vector type in the application may store three 64-bit fields: the pointer to the underlying buffer, a length, and a capacity, while the vector type in the sandbox stores all three in 32-bit variables with an offset pointer. Sesame provides a SandboxCopy trait for quickly deep-copying Rust objects to/from sandbox memory, while altering their memory layout and offsetting any pointers (i.e., "pointer swizzling"). Sesame implements this trait for primitives, strings, and vectors, and provides developers with macros to derive the trait for their custom types. For types that do not implement this trait, Sesame falls back on serializing and deserializing data.

6.3 Critical Regions

Developers must use a *critical region (CR)* to send data to custom sinks. They may also use a CR to execute a leakage-free region that Scrutinizer conservatively rejected, and in rare cases where that region is incompatible with sandboxing or the runtime overheads of sandboxing prove undesirable. Code reviewers manually review CRs for unintentional leakage and for correct use of the region's context.

Review Process. CRs should be concise, single-purpose, and self-contained. Reviewers should reject CRs that are unfocused or overly complex and request that authors simplify them, similar to regular code review.

Sesame executes a CR only after a successful policy check on the input data, given the provided context (§4.2). Reviewers thus do not need to reason about the semantics of the associated policies (e.g., which emails are allowed). Instead, they reason about the code of the CR and the semantics of the context object it receives (e.g., does the CR send an email to the address specified in the context). They also need to ensure the CR has no unintentional leaks, e.g., via logging.

Reviewing a CR includes reasoning about library code it calls into. Large companies have existing procedures for approving and updating dependencies; for example, Google curates approved dependencies and versions that developers are allowed to use [21]. A reviewer of a CR that invokes an approved dependency need only check that the usage of the dependency is consistent with its documented API, and relies on the curation process to ensure the sanity of the docs. Open source projects or smaller companies may not have the capability to perform such detailed vetting. In such cases, reviewers must look up each dependency they encounter in a CR, and decide how strictly they review it based on reputation, bug reports, or other criteria.

Sesame requires reviewing and re-signing a CR when its code or dependencies change. Since Rust locks the dependency versions in an application's `Cargo.lock` file, dependency updates are explicit, intentional, and generally rare.

Signatures. Reviewers indicate to Sesame that they approve a CR by signing it. Signatures serve two purposes: *(i)* they verify to Sesame that the CR has indeed been approved by an authorized reviewer, and *(ii)* that the CR and its dependencies are unchanged since the review. During review, Sesame produces a hash of the CR. Reviewers sign that hash and attach their signature to the CR. During each subsequent release build, Sesame reproduces a hash for that CR using the same procedure, and checks that the signature attached to the CR is a valid signature for that hash. The hash differs if the CR changed since review, including changes to helper and library functions. This in turn invalidates the signature, which prompts Sesame to reject the CR and require a reviewer to re-vet and re-sign it.

Hashing. Sesame builds a call graph for the CR similar to §6.1, but stops at calls to external dependencies. Sesame concatenates the source code of all functions in the call graph into a normalized string (e.g., without comments and extraneous white spaces). Sesame then records the external dependencies the region calls into and traverses the `Cargo.lock` file to find the exact versions of these dependencies and any transitive dependencies. Sesame augments the CR string with the dependency information, and hashes it. Changes to the application portion of the CR, or to direct or transitive dependencies, result in a different hash. Updates to dependencies or application code unrelated to the CR do not affect the hash and avoid superfluous review.

Ergonomics. Sesame omits signature checks in debug mode, which allows developers to implement and test their CRs without review. Then, authors request signatures from reviewers, who must review these regions, e.g., as part of a pull request. This mirrors existing industry practices that require approval by authorized reviewers for merging code.

Our prototype uses GitHub as a key provider and for identity management. Sesame pulls public keys for reviewers from GitHub to validate the signature of each CR during release builds. A reviewer's privileges can be revoked, and Sesame can either invalidate their signatures immediately, or preserve existing signatures while disallowing future signatures if privilege revocation and signatures are timestamped.

7 Implementation

We implement a Sesame prototype in 12k LoC in Rust, including 4.2k LoC for Scrutinizer, 2.1k for Sesame's web framework, and 0.5k for Sesame's MySQL library. The web framework and MySQL library mirror the APIs of Rocket.rs [38] and mysql [3], modified to accept PCons at sinks and generate PCons at sources. Sesame also has partial support for SeaORM [43]. All this code is trusted, as is RLBox (for SRs).

RLBox is primarily designed for sandboxing C++ functions; Sesame generates the necessary wrappers and bindings to use it with Rust. Our prototype uses RLBox with WebAssembly (WASM), and thus does not support code and libraries incompatible with WASM in sandboxed regions.

Sesame provides mock versions of its built-in sources and sinks for end-to-end application tests. These versions strip policy containers from application outputs, and allow testing code to create synthetic contexts to test policy check functions. Sesame uses Rust's conditional compilation to ensure these features are only available in tests.

Finally, Sesame includes linting rules it checks when compiling in release mode. These forbid developers from manually implementing SesameType on their custom types, and instead force them to use Sesame's `[#derive]` macros to generate automatic and correct implementations.

Our prototype's hashing of CRs is sensitive to some syntactic changes to code that have no semantic effect (e.g., renaming a variable), which invalidate signatures. Better stable code hashing techniques could improve precision [14].

8 Application Case Studies

We applied Sesame to four web applications: *(i)* WebSubmit [37] and *(ii)* Portfolio [24] are pre-existing applications we ported to Sesame; *(iii)* Voltron is an application from Storm [27] that lets group of students collaboratively edit a piece of code with instructor oversight; and *(iv)* YouChat is a simple chat application for individuals and groups. The original versions of WebSubmit and Portfolio have 1.3k and 5.1k LoC. We built a Rust version of Voltron with comparable functionality to the original LiquidHaskell application. For Voltron and YouChat, we first built idiomatic Rust implementations without Sesame in mind (0.5k and 0.8k LoC) and then ported these implementations to Sesame. This section describes the applications' policies and our process porting them to Sesame, while §9.1 quantifies developer effort.

Policies. We added policies for access control, purpose limitation, user consent, and aggregates to the applications.

YouChat has a single access control policy: users can only view messages that they sent or received, or messages from groups they are members of.

For **Voltron**, we implemented all of the policies from Storm [27]: *(i)* only admins can enroll new instructors; *(ii)* students can only be enrolled into a class by that class instructor; and *(iii)* code buffers can only be read or modified by students in the corresponding group or by the class instructor. The last policy turns into two Sesame policies that cover reads and writes. We also added two additional policies: *(i)* Firebase authentication headers from HTTP requests may only be used for read database queries; and *(ii)* endpoints may only use the email address of the authenticated user.

WebSubmit is a homework submission system similar to our example in §3. Prior to porting to Sesame, we extended WebSubmit with a machine learning model over students' grades (training and inference), and with aggregate statistics for university managers and employers, as well as a user consent choice to release the latter. WebSubmit has six policies: *(i)* a student's answer is only accessible to the student, instructor, and discussion leaders for the corresponding lecture; *(ii)* an individual grade is only accessible to the student and instructor; *(iii)* a student's average grade and email are only released to employers if the student consents; *(iv)* a student's data can only be used for model training if the student consents; *(v)* university administrators cannot aggregate over protected demographic data; and *(vi)* aggregate grade data released must contain grades from at least k different students (k-anonymity).

Portfolio is a high school admissions system deployed in the Czech Republic [24]. Candidates create accounts, input personal information, and upload PDF documents for admission review; Portfolio encrypts the stored data at rest. We add two policies to Portfolio, which cover the most sensitive data it handles: *(i)* sensitive candidate data, in either plain or ciphertext form, is accessible only to the candidate themselves and to school administrators reviewing their application; and *(ii)* private keys are never revealed outside of the DB, except in cookies to their respective owners.

Porting Experience. To port these applications to Sesame, we first implemented the policies, then associated policies with data at sources, and adapted application code to use PCons instead of raw data. Finally, we used Sesame to check verified regions, and reviewed and signed any CRs.

Swapping Sesame-provided libraries for the web framework (Rocket) and database connector (SeaORM in Portfolio, MySQL in others) made the applications fail to compile, as the libraries now provide PCons in, e.g., HTTP request data, but the application expects them to be raw types. To fix these compiler errors, we replaced these raw types with PCons with an associated policy: for structured data, this was a quick update to database schemas; for unstructured data, we had to change code that obtains it from Sesame libraries.

These changes sufficed for simple endpoints (e.g., Figure 2), but endpoints that compute on data in PCons still had compiler errors. We fixed these errors by introducing privacy regions, akin to Figure 1b. Rust made this process of lifting code into privacy regions easy, as idiomatic Rust already encourages closures (e.g., in iterator chains).

With the application building and tests passing in debug mode, we compiled in release mode to invoke Scrutinizer and Sesame's lints. Scrutinizer accepted most regions as verified regions; for the remainder, we found the distinction between sandboxed regions and CRs obvious (e.g., hashing a password vs. sending an email). Scrutinizer rejected six regions that use an encryption library that relies on async Rust, even though they are in fact leakage-free. We attempted to make them sandboxed regions, but the library is incompatible with WebAssembly, so we had to turn these regions to CRs and review them manually. Replacing the library with a compatible alternative could avoid these CRs.

Anti-Patterns. We found two problematic code patterns. First, because Scrutinizer currently lacks support for async Rust, it rejects regions that contain `await`. To overcome this, we perform operations inside the privacy region without `await` and return a PCon that wraps a future. PCon has an API to `await` a wrapped future outside of privacy regions; this is safe because the result remains wrapped in a PCon.

Second, some endpoints in Portfolio and WebSubmit early-return, e.g., on failed form validations. But early-return checks inside privacy region closures cannot return from the surrounding function. Instead, we return a Result<T> from these privacy regions, which Sesame wraps in a PCon with the appropriate policy. Sesame's Fold API (§5) lets the application fold it into a Result<PCon<T, P>>. This allows early-return when the Result is an error, but creates a channel for implicit leakage. Policies can disable this if unacceptable, forcing the remaining code to operate on the Result monad instead and defer the early return.

9 Evaluation

We evaluate Sesame with four applications: YouChat, Voltron, Portfolio, and WebSubmit. We ask three questions:

1. What developer effort does using Sesame require? (§9.1)
2. What is the impact of using Sesame on end-to-end application performance? (§9.2)
3. What is the impact of key Sesame components on its correctness and performance? (§9.3)

Our benchmark machine has a Xeon E3-1230v5 CPU (3.4GHz) and 64 GiB RAM. We use Ubuntu 20.04, Rust nightly-2023-10-06 for sandboxes and nightly-2023-04-12 for static analysis.

9.1 Developer Effort

We evaluate the developer effort required to implement applications with Sesame or to port them as described in §8.

Implementing Policies. A critical component of writing an application with Sesame is implementing policies. Ideally, the size of these policies and the effort required to implement them should depend on the complexity of the policies themselves, rather than application size.

App.	Policy count	App LoC	...of which policy	...of which check
YouChat	1	1.1k	118	38
Voltron	6	1.2k	425	121
Portfolio	2	6.7k	305	85
WebSubmit	6	2.2k	373	72

Figure 6. Policy code size scales with the complexity and number of policies, rather than the size of the application.

We thus measured the size, in LoC, of the policies in our four applications. Policy code includes the policy structs, constructors, the `Policy` trait implementations, and the check functions. The check function typically dominates the developer effort for policy authoring.

Figure 6 shows the results. Policy complexity varies based on the diversity of application user roles and purposes of data use. For example, Voltron (1.2k LoC total) is a small application, but it contains a complex hierarchy of user roles, so its policy code is comparatively large (425 LoC). By contrast, Portfolio is the largest application (6.7k LoC), but it has fewer, simpler policies, as it collects broad data (academic history, letters, demographics) but for the same purpose (viewing by admissions officials). This indicates that effort for writing Sesame policies reflects the complexity of the application's data-use semantics, rather than application size.

We compare the three policies Storm [27] implements for Voltron with the equivalent policies in Sesame. In Sesame, the check functions for these policies consist of 88 Rust LoC, compared to 37 and 17 LiquidHaskell LoC for policy and "non-trivial type annotations" in Storm. This suggests that writing policies for Sesame requires comparable effort to existing work. In addition, Sesame can express more diverse policies, such as k-anonymity, that depend on runtime state.

Using Policy Containers. Writing a Sesame application requires developers to use PCons to associate data with policies, to change types to PCons where necessary (e.g., in function signatures), and to add privacy regions. To quantify this effort, we consider our experience porting Portfolio to Sesame. Porting an existing application is more work than writing a new one that already anticipates these abstractions.

Porting Portfolio took 30 person-hours. We changed types in five ORM and four JSON payload structs to associate policies with structured data. For unstructured data, e.g., cookies and GET parameters, we declared the policy type on each read. We spent the majority of porting time adjusting function signatures to use PCons. This is largely mechanic and guided by compiler type errors that indicate where changes are needed. Compiler errors related to PCons also pointed us to application logic that requires raw data, and we lifted this logic into privacy region closures. Across our applications, this required no structural changes (e.g., moving code between functions or changing control flow). Sesame requires

App.	Region	# of regions	Total % of App	Size (LoC)
YouChat	VR	3	<1%	1–5
Voltron	VR	3	<1%	1–2
	CR	2	<1%	3–7
Portfolio	VR	43	1.2%	1–8
	SR	6	<1%	1–5
	CR	20	1.4%	1–27
WebSubmit	VR	17	2.0%	1–9
	SR	2	1.0%	4–19
	CR	2	1.3%	8–22

Figure 7. Counts and sizes of each Sesame privacy region used. This accounts for the size of the top-level region closure, but not helper functions that require no porting effort.

App.	LoC	# CRs	Burden %	Avg Burden
YouChat	1.1k	0	–	–
Voltron	1.2k	2	<1%	5 LoC
Portfolio	6.7k	20	5.0%	16.8 LoC
WebSubmit	2.2k	2	1.5%	16.5 LoC

Figure 8. Critical region count in applications. Burden % indicates the fraction of code that needs review; average burden is the average size of in-crate code for CRs.

a few dozen privacy regions for Portfolio, and smaller applications require fewer (Figure 7). Overall, we had to lift 1–4.3% of application code into region closures.

Critical Region Review. We now consider the review effort for critical regions. A good result for Sesame would show that critical regions are slim and infrequent.

Figure 8 shows the number of critical regions and their review burden in terms of in-crate code to audit. In all applications, critical regions are small and shallow: the review burden in Portfolio makes up 5% of application code including the CR closures and all their in-crate helpers. Portfolio has 20 CRs with an average review burden of 17 LoC each.

CRs often invoke external dependencies in addition to in-crate code, so the review burden extends to auditing these dependencies. Reviewers may leverage organizations' existing supply chain audit protocols for approving dependencies and updates to reduce review burden.

These results suggest that Sesame focuses reviewer attention on infrequent, small, and contained code regions.

9.2 Application Performance

Next, we evaluate Sesame's impact on end-to-end application performance using WebSubmit and Portfolio. PCon encapsulation adds overhead due to pointer indirection and the additional memory footprint of policies; runtime policy checks and the use of sandboxes may also incur overhead. We compare *(i)* the baseline application, and *(ii)* application

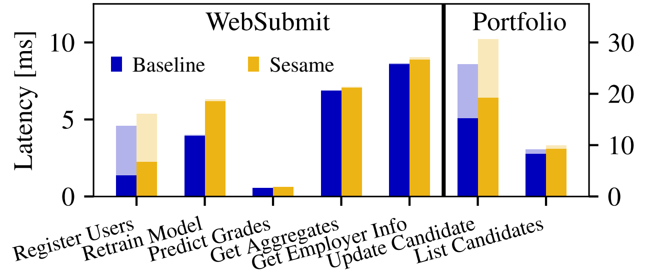


Figure 9. Sesame has reasonable performance overheads for WebSubmit and Portfolio (solid: median; shaded 95th %-ile).

with Sesame. We use a database with 100 students and 100 homework questions for WebSubmit and an admission cohort of 1k candidates (one application each) for Portfolio. We measure endpoint latency for HTTP endpoints that use privacy regions and/or require policy checks. A good result for Sesame would show comparable latencies with and without Sesame and acceptable overhead for sandboxed operations.

Figure 9 shows the results. WebSubmit performs API key hashing during user registration ("Register Users") and ML model training ("Retrain Model") in sandboxed regions. These endpoints with sandboxed regions see 10% and 55% overhead, most of which is due to the cost of copying data into the sandbox. User registration (10% overhead) copies a single record into the sandbox (tail latency is due to DB disk I/O); retraining the ML model transfers the grades of all consenting users to the sandbox (55% overhead). "Get Aggregates", which computes statistics over the whole class, and "Get Employer Info", which computes average grades for release to employers, have a 1–3% overhead. Both endpoints combine many students' data, which can have different policies that cannot be folded together into a single policy, so Sesame must perform repeated policy checks. "Predict Grades" has a 10% overhead, albeit with low absolute latency as grade inference operates on an in-memory model without I/O.

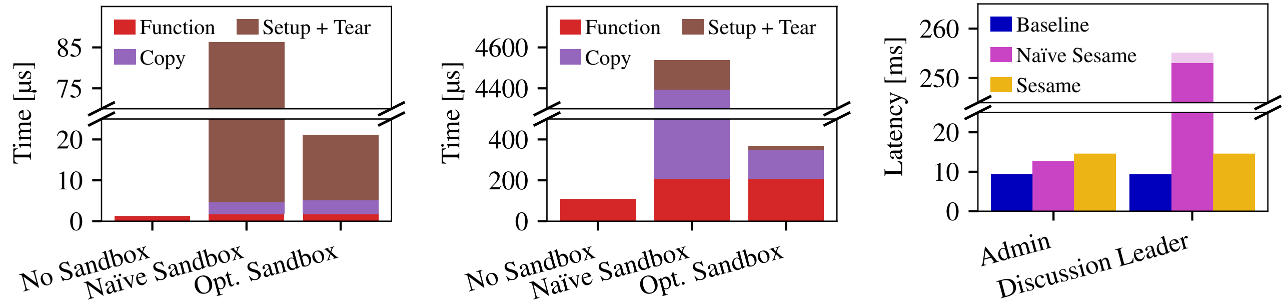
Portfolio's "Update Candidate" writes candidate demographics to the DB on disk. It performs JSON serialization in a sandboxed region, resulting in a 25% overhead. "List Candidates" has admins retrieve a paginated list of 20 applications. Sesame uses Fold to combine the candidates' policies into a single policy check, resulting in a 10% overhead.

These results indicate that PCons and policy checks impose low overheads, while the cost of sandboxing scales with the size of the data copied into the sandbox.

9.3 Drill-Down Experiments

We now evaluate key components of Sesame—Scrutinizer, sandboxes, and policy composition—in isolation.

Scrutinizer. Sesame's guarantees depend on sound rejection of leaking regions. Simultaneously, if Scrutinizer



(a) Reuse optimizations speed up a sandbox running a cheap hash function by 4×.

(b) Copy optimizations speed up a sandbox running ML training by 11×.

(c) Policy composition avoids repeated policy checks (median; 95th%-ile shaded).

Figure 10. Drill-down experiments: Sesame benefits from sandbox reuse reducing setup/teardown overhead (a) and from direct copies avoiding serialization of data passed into the sandbox (b); policy composition reduces check overheads (c). Functions run 2× slower in the sandbox because of the WASI runtime [11] and inserted dynamic checks, e.g., on pointer dereferencing.

	Privacy Regions			
Application	*Leak-free*	*... of those, accepted*	**Functions Analyzed**	**Time**
YouChat	3	3	823	2.31s
Voltron	3	3	11	0.80s
Portfolio	55	43	774,624	711.61s
WebSubmit	19	17	332,326	500.52s

Figure 11. Scrutinizer accepts the majority of leakage-free regions, avoiding sandboxing overheads or the burden of manual review. Scrutinizer rejected all leaking regions.

accepts genuinely leakage-free regions, this avoids sandboxing overhead and manual review. We evaluate Scrutinizer on 98 privacy regions across our four applications: 80 that we manually verified as leakage-free, and 18 that we know to be leaking. In a good result, Scrutinizer would accept most leakage-free regions and reject all leaking regions.

Figure 11 shows the results of running Scrutinizer over regions we know to be leakage-free. Scrutinizer successfully verified 66 of 80 regions, but conservatively rejected two regions in WebSubmit and twelve in Portfolio. Six of the rejected regions use async code not currently supported by Scrutinizer. The remaining eight regions perform cryptographic hashing, encryption, ML training, and CSV serialization via external libraries that dereference raw pointers for performance. With extra engineering effort, Scrutinizer should be able to verify some of these as leakage-free. Scrutinizer correctly rejects all 18 leaking regions.

We also evaluated Scrutinizer on methods from standard library containers, which extensively use unsafe code for performance. Scrutinizer rejected all leaking methods, and rejected two out of 57 leakage-free methods.

Sandboxes. To measure the cost of sandboxing, we consider the sandboxed regions in WebSubmit: “Register Users” and “Retrain Model”. The former computes a hash over a short string and the latter fits a linear regression model to thousands of rows. We compare the runtime of executing the two privacy regions *(i)* without a sandbox (the baseline); *(ii)* in a sandbox without any optimizations (“Naïve Sandbox”); and *(iii)* in a sandbox with reuse and copy optimizations (§6.2).

The results are in Figures 10a and 10b. Compared to the baseline, the naïve sandbox adds substantial overhead: sandbox setup and teardown dominate the (fast) hashing sandbox’s runtime (Figure 10a), while data copying via serialization dominates for ML training, which operates over more data (Figure 10b). With optimizations, the overhead of sandboxes decreases substantially. Reusing sandboxes after erasing their memory improves the hashing sandbox’s runtime by 4×. Copying data and swizzling pointers reduces data copy time by 29× and overall runtime by 11× compared to an naïve, serialization-based ML training sandbox.

In both cases, the actual code of the region itself (“Function”) takes roughly twice as long as without sandboxing, in line with overheads reported by RLBox [30].

Policy Composition. We now measure the performance impact of policy composition using fold. We use two endpoints from WebSubmit that display homework answers to course staff and discussion leaders, for a setup with 100k answers (100 students, 100 lectures). Releasing homework answers requires evaluating the `AnswerAccessPolicy`: answers are shared only with authors, admins, or discussion leaders. The list of admins is part of the application’s in-memory configuration, while discussion leaders must be retrieved with a database query. In the discussion leader case, each policy check requires a database query, and joining policies that have the same discussion leaders reduces this policy check to a single database query. The experiment measures the impact of this policy join, and its overhead for the admin case, where the policy check is inexpensive. Figure 10c shows the results: the admin endpoint without policy composition incurs a 1.4× performance overhead compared

to a policy-free baseline, while policy composition incurs a 1.6× overhead. For the discussion leaders endpoint, retrieving answers without policy composition has a 27× overhead, while the same operation with policy composition incurs a 1.5× overhead. This experiment indicates that policy composition is a worthwhile abstraction: while it incurs some additional overhead when policy check execution is cheap, it cuts the cost of policy checks when execution is expensive.

10 Related Work

Sesame draws inspiration from Resin's techniques for attaching flexible policies to data [47]. Akin to taint tracking in Resin's runtime, PCons associate data with policies, track and combine policies as data flows through the application, and ensure applications cannot modify policies or reveal data without policy checks. Sesame avoids expression-granularity taint tracking, and instead manages taints at the boundaries of privacy regions, which operate on untainted data, secured by static analysis, sandboxing, and rare critical regions.

Cocoon [26] is a static type-based IFC system for Rust that avoids compiler modifications and runtime overheads. Cocoon centers on *secret blocks* that operate directly on data and serve a similar purpose to verified regions. While Sesame relies on Scrutinizer to verify privacy regions, Cocoon ensures that blocks are leakage-free via trait bounds and rewriting with procedural macros. This approach is more conservative than Scrutinizer's: it e.g., disallows all Rust unsafe code or types with interior mutability, and requires porting any dependencies invoked in a block to Cocoon. Scrutinizer, by contrast, analyzes unmodified libraries. Finally, Cocoon enforces non-interference via static secrecy labels, while Sesame enforces arbitrary dynamic policies.

RuleKeeper [19], like Sesame, combines static analysis with runtime policy enforcement. It protects against policy violations at HTTP endpoints and database queries, but not against accidental leaks or custom sinks (e.g., logging, file I/O). RuleKeeper's static analysis of JavaScript source code is unsound and can miss policy violations. Sesame achieves soundness by analyzing Rust code for leakage-freedom, rather than for more complex policy compliance. While both RuleKeeper and Scrutinizer can have false positives, Sesame can apply sandboxing to code that fails static analysis, while RuleKeeper must default to manual review.

Laminar [36] is a decentralized information flow control (DIFC) system that supports "security regions" within which code can access raw data. Laminar uses a modified Java VM and a kernel module to manage labels and capabilities that developers assign to regions. Sesame's privacy regions enforce leakage-freedom via static analysis or sandboxes.

Harpocrates [34] associates policies with data and enforces context-dependent checks when calling into potentially side-effecting code, detected via static analysis. Harpocrates leverages Scala features to avoid type signature changes and to automate context capturing and propagation of policies to derived data. Sesame requires more developer work, but combines static analysis, sandboxing, and code review for end-to-end policy enforcement from explicit sources to sinks.

Riverbed [44] separates users with different policies into separate application deployments ("universes") that cannot interact, and leverages a taint-tracking language runtime. Sesame lets users with different policies share a deployment, but ensures their data interacts only in permissible ways.

Other policy enforcement systems differ from Sesame in their threat model, scope, or mechanism used. PrivGuard [45], like Sesame, uses static analysis and runtime mechanisms to enforce privacy policies, but targets a narrower set of Pandas-like data analytics programs. Ryoan [23] has a strong threat model: it trusts neither the application nor the underlying cloud platform, and leverages sandboxes, hardware enclaves, and IFC to protect sensitive data. Sesame trades a weaker threat model for general application support and lower overheads. Zeph [5] relies on cryptography to restrict computations over sensitive data, but only supports restricted classes of computations (e.g., streaming sums), while Sesame supports general computation. Finally, classic role-based access control (RBAC), as well as systems like Daisy [22] and Qapla [28] enforce access control policies in a database-centric environment. These approaches cannot protect against application bugs; Sesame focuses on application code and supports policies beyond access control, such as k-anonymity. K9db [2] handles users' right to access/delete their data within a relational database; an orthogonal concern to application bugs, which K9db cannot reason about.

11 Discussion and Future Work

Sesame could be extended with an optional DSL for policies that compiles to Rust. This could make expressing common policies easier and more succinct, enable automated reasoning about them, and maintain expressiveness.

Sesame's guarantees for unsafe Rust code rely in part on pointer obfuscation (§5). Sesame could instead apply a static analysis that detects unsafe code that breaks encapsulation. We believe this is feasible and has utility beyond Sesame.

Finally, leveraging information from a database that understands data ownership, such as K9db [2], in Sesame's policies is an interesting avenue for future work.

12 Conclusion

Sesame is a framework to help developers enforce application-specific privacy policies over user data across an application. Privacy regions allow developers to operate over policy-protected data by leveraging runtime policy checks, static analysis, sandboxing, and human code review.

We show that Sesame requires modest developer effort, incurs acceptable overheads, and achieves practical compliance. Sesame is available as open-source software:

https://github.com/brownsys/sesame.

Acknowledgments

We thank Deniz Altınbüken, Alexandre Meïer Doukhan, Shriram Krishnamurthi, Akshay Narayan, Deepti Raghavan, Samyukta Yagati, Carolyn Zech, and the ETOS and Systems groups at Brown for their helpful feedback on drafts of this paper. Feedback from the anonymous reviewers and our shepherd, Brad Karp, helped greatly improve the paper. We are also grateful to Sinan Pehlivanoglu, whose master's thesis on enforcing privacy polices using compiler techniques [34] inspired the policy APIs and use of static analysis in Sesame; and to Sreshtaa Rajesh and Livia Zhu, whose CSCI 2390 project on static IFC in Rust ("Beaver") spawned early ideas on how to realize policy-protected data in Rust.

This research was supported by NSF awards CNS-2045170 and DGE-2335625, by a Google Research Scholar Award, a Microsoft Grant for Customer Experience Innovation, an Amazon Research Award, and a gift from VMware.

References

[1] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. *Sesame*. Sept. 2024. URL: https://github.com/brownsys/sesame (visited on 09/14/2024).

[2] Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Artem Agvanian, Leonhard F. Spiegelberg, and Malte Schwarzkopf. "K9db: Privacy-Compliant Storage For Web Applications By Construction". In: *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, Massachusetts, USA, July 2023, pages 99–116.

[3] blackbeam. *mysql*. URL: https://docs.rs/mysql_common/latest/mysql_common/ (visited on 04/19/2024).

[4] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing static and dynamic typing for information-flow control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Vancouver, British Columbia, Canada, Aug. 2015, pages 289–301.

[5] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. "Zeph: Cryptographic Enforcement of End-to-End Data Privacy". In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 387–404.

[6] Jack Cable. *LinkedIn AutoFill Exposed Visitor Name, Email to Third-Party Websites*. Apr. 2018. URL: https://lightningsecurity.io/blog/linkedin/ (visited on 09/14/2024).

[7] California Attorney General. *Privacy Enforcement Actions*. URL: https://oag.ca.gov/privacy/privacy-enforcement-actions (visited on 07/31/2023).

[8] Adam Chlipala. "Static checking of dynamically-varying security policies in database-backed applications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 105–118.

[9] Adam Chlipala. "Ur: statically-typed metaprogramming with type-level record computation". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario, Canada, 2010, pages 122–133.

[10] Stephen Chong, K. Vikram, and Andrew C. Myers. "SIF: enforcing confidentiality and integrity in web applications". In: *Proceedings of 16th USENIX Security Symposium*. Boston, Massachusetts, USA, Aug. 2007.

[11] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. Mar. 2019. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (visited on 08/12/2024).

[12] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. San Diego, California, USA, 2022, pages 1–14.

[13] Dorothy E. Denning and Peter J. Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* 20.7 (1977), pages 504–513.

[14] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler, and Daniel Lohmann. "cHash: Detection of Redundant Compilations via AST Hashing". In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*. Santa Clara, California, USA, July 2017, pages 527–538.

[15] Petros Efstathopoulos and Eddie Kohler. "Manageable Fine-Grained Information Flow". In: *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Glasgow, Scotland, UK, Apr. 2008, pages 301–313.

[16] European Data Protection Board. *Binding Decision 2/2022 on the dispute arisen on the draft decision of the Irish Supervisory Authority regarding Meta Platforms Ireland Limited (Instagram) under Article 65(1)(a) GDPR*. July 2022. URL: https://www.edpb.europa.eu/system/files/2022-09/edpb_bindingdecision_20222_ie_sa_instagramchildusers_en.pdf (visited on 09/14/2024).

[17] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and

repealing Directive 95/46/EC (General Data Protection Regulation)". In: *Official Journal of the European Union* L119 (May 2016), pages 1–88.

[18] *Family Educational Rights and Privacy Act*. United States Code of Laws, 20 U.S.C. § 1232g. Aug. 1974.

[19] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. "RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks". In: *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*. San Francisco, California, USA, May 2023, pages 2817–2834.

[20] Google, Inc. *Project Strobe: Protecting your data, improving our third-party APIs, and sunsetting consumer Google+*. Oct. 2018. URL: https://blog.google/technology/safety-security/project-strobe/ (visited on 09/14/2024).

[21] Google, Inc. *Google Open Source: Third-Party*. Mar. 2023. URL: https://opensource.google/documentation/reference/thirdparty (visited on 09/16/2024).

[22] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. "Information-Flow Control for Database-Backed Applications". In: *Proceedings of the 4th 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. Stockholm, Sweden, June 2019, pages 79–94.

[23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. "Ryoan: A distributed sandbox for untrusted computation on secret data". In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), pages 1–32.

[24] Vojtěch Jungmann and Sebastian Pravda. *Portfolio*. 2022. URL: https://github.com/admisio/Portfolio (visited on 04/12/2024).

[25] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information flow control for standard OS abstractions". In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, 2007, pages 321–334.

[26] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. "Cocoon: Static Information Flow Control in Rust". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024).

[27] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. "STORM: Refinement Types for Secure Web Applications". In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, July 2021, pages 441–459.

[28] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. "Qapla: Policy compliance for database-backed systems". In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, British Columbia, USA, Aug. 2017, pages 1463–1479.

[29] Andrew C. Myers and Barbara Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000), pages 410–442.

[30] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting fine grain isolation in the Firefox renderer". In: *Proceedings of the 29th USENIX Security Symposium*. Virtual Event, Aug. 2020, pages 699–716.

[31] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-008*. URL: https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-008 (visited on 07/31/2023).

[32] NOYB: European Center for Digital Rights. *GDPRHub: CNIL SAN-2020-018, Nestor SAS*. URL: https://gdprhub.eu/index.php?title=CNIL_-_SAN-2020-018 (visited on 07/31/2023).

[33] NOYB: European Center for Digital Rights. *GDPRHub: GPDDP 9485681, Vodafone Italia*. URL: https://gdprhub.eu/index.php?title=Garante_per_la_protezione_dei_dati_personali_-_9485681 (visited on 07/31/2023).

[34] Sinan Pehlivanoglu and Malte Schwarzkopf. *Harpocrates: A Statically Typed Privacy Conscious Programming Framework*. May 2022. arXiv: 2411.06317 [cs.CR].

[35] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. "Liquid information flow control". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (Aug. 2020).

[36] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. "Laminar: practical fine-grained decentralized information flow control". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland, 2009, pages 63–74.

[37] Malte Schwarzkopf. *websubmit-rs: a simple class submission system*. URL: https://github.com/ms705/websubmit-rs (visited on 06/03/2024).

[38] SergioBenitez. *rocket*. URL: https://docs.rs/rocket/latest/rocket/ (visited on 04/19/2024).

[39] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. "Logical attestation: an authorization architecture for trustworthy computing". In: *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 2011, pages 249–264.

[40] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. "Secure distributed programming with value-dependent types". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pages 266–278.

[41] Nikhil Swamy, Brian J Corcoran, and Michael Hicks. "Fable: A language for enforcing user-defined security policies". In: *Procedings of the 29th IEEE Symposium on Security and Privacy (S&P)*. Oakland, California, USA, May 2008, pages 369–383.

[42] *The Health Insurance Portability and Accountability Act of 1996*. United States Public Law 104-191. Aug. 1996.

[43] Chris Tsang and Chan Billy. *SeaORM: An async & dynamic ORM for Rust*. URL: https://crates.io/crates/sea-orm (visited on 09/17/2024).

[44] Frank Wang, Ronny Ko, and James Mickens. "Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services". In: *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Feb. 2019, pages 615–630.

[45] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. "PrivGuard: Privacy regulation compliance made easier". In: *Proceedings of the 31st USENIX Security Symposium*. Boston, Massachusetts, USA, Aug. 2022, pages 3753–3770.

[46] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, dynamic information flow for database-backed applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, California, USA, 2016, pages 631–647.

[47] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Improving Application Security with Data Flow Assertions". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pages 291–304.

[48] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, Washington, USA, 2006, pages 263–278.

[49] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. "Securing Distributed Systems with Information Flow Control". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, California, USA, Dec. 2008.

A Appendix

This appendix contains supplementary material that has not been peer-reviewed.

Scrutinizer Code. Scrutinizer is open-source software at https://github.com/brownsys/scrutinizer. Our experiments in the paper used the version tagged `sosp24`.

Scrutinizer Analysis Details. Scrutinizer follows a two-stage approach to check the properties in §6.1.

First, Scrutinizer builds a call tree of all functions and code that may be executed by the top-level function under analysis. Scrutinizer uses Rust's dataflow analysis framework to traverse function bodies recursively in execution order. This discovers all possible function bodies that the top-level function could call, and organizes them into a call tree. When it encounters dynamic dispatch, Scrutinizer attempts to construct a superset of all concrete functions the dynamic dispatch may resolve to, and analyzes all of them. If Scrutinizer cannot construct such a set, it rejects the function. Scrutinizer keeps track of functions it visited to avoid unnecessary recomputation. This stage finishes when Scrutinizer discover no more new function calls.

Second, Scrutinizer begins the analysis stage. Scrutinizer rejects a top-level function if it captures any variables with a mutable reference, since such sensitive data could leak into such variables. For top-level functions that pass this check, Scrutinizer labels the arguments to the function as "sensitive". It then traverses every statement in the call tree while simultaneously propagating the "sensitive" label to aliases and derived variables using Flowistry [12]. This ensures that Scrutinizer keeps track of sensitive arguments as they are passed, aliased, and derived from throughout the call tree. If Scrutinizer encounters a function call into native or otherwise unresolvable code that sensitive variables flow into, it rejects. If Scrutinizer encounters a function call to an allow-listed function, or with arguments that lack the sensitive label, it skips it. Otherwise, Scrutinizer analyzes the function's body. If Scrutinizer encounters an unsafe mutability mechanism, such as a raw mutable pointer dereference, it rejects. If Scrutinizer finishes analyzing the call tree without rejecting, it accepts the top-level function.