1 Problem and Motivation

Modern web applications are subject to privacy policies required for compliance with privacy laws (e.g., GDPR, HIPAA, FERPA), set by an organization internally or accepted by users in the terms of service. Inadvertent breaches of self-imposed policies or laws lead to significant penalties [4, 14–16]. For example, Twitter used phone numbers for advertising, even though users had consented only to their use for account security, incurring a $150m fine [6].

Developers today lack practical frameworks to ensure that their code abides by the privacy policies, which leads to undetected violations. Remembering which policy applies to data and performing the appropriate checks throughout the application code is easy to get wrong or forget. Developers need small and clear regions of privacy-critical code to focus their attention on and automatic guarantees for the remaining codebase.

2 Background and Related Work

Information flow control (IFC) enforces end-to-end security policies in programs. IFC systems may rely on runtime enforcement via runtime labels [10, 22–24], compile-time enforcement via type systems [11, 19] or static analysis [8], or on hybrid approaches [2, 5, 12]. A common failure point of these systems is that they are challenging to use for developers [9]. Some require using complex or custom languages unfamiliar to web developers [2, 11, 12]. They frequently require significant application changes and rely on developers managing and passing complex labels [23, 24]. Dynamic approaches often come with a steep performance cost [22], while static approaches can only express limited policies that developers must laboriously encode using first order logic [17] or dependent types [18]. Some IFC systems like Riverbed [20] avoid the usability and performance pitfalls, but impose limitations on application functionality, like separating user data into separate universes.

Dynamic policy tracking in Resin [22] attaches policy objects to data, and tracks them across a program in a modified language runtime. Resin’s runtime tracking imposes substantial overhead, and requires using slower interpreted languages, but minimizes developer effort.

Resin’s techniques are a key inspiration for Alohomora. Alohomora’s privacy containers perform a similar role to taint tracking in Resin’s: they associate data with policies, track and combine policies as data is used in the application, and ensure applications cannot modify policies or reveal data without checking them. However, Alohomora targets a statically-typed language and avoids performing taint tracking for every expression. Alohomora only manages taints at the boundaries of privacy regions, allowing selective operation on raw, untainted data within these regions. Alohomora achieves this by relying on infrequent, manually-reviewed critical regions and a more relaxed threat model.

Compile-time policy enforcement guarantees policy compliance statically in systems like Storm [11]. Application developers define their policies in a single, reviewable location, and associate them to data at the granularity of database columns. They express this association in Storm’s objection-relational model and write static policies in a Prolog-like relational logic. Storm uses refinement types to check policies at compile time, but requires developers to write applications in LiquidHaskell, limiting adoption. Alohomora targets a widely-used programming language (Rust), and its policies execute arbitrary Rust code at runtime. This means Alohomora can express richer policies that rely on dynamic information about the data or application.

Other approaches include cryptographic enforcement systems like Zeph [3], which have strong threat models, but incur prohibitive performance overheads and only support restricted classes of computations. PrivGuard [21] focuses on ensuring that programs submitted by analysts, and executed by data curators, meet baseline privacy policies mandated by privacy regulations (e.g., GDPR). Like Alohomora, PrivGuard has a relaxed threat model that assumes analysts are honest but fallible, and combines static analysis with runtime mechanism (TEEs). However, Alohomora focuses on enforcing a broader set of privacy policies in web applications.

3 Uniqueness

Alohomora ensures by construction that developers abide by privacy policies when they operate on sensitive data. Alohomora embraces key taint-tracking techniques from prior work on Information Flow Control (IFC) systems, but makes different tradeoffs to provide practical abstractions for developers. Alohomora aims to keep developer effort low; it builds on the Rust type system and supplements it with static analysis to achieve automatic guarantees for most code. For the remaining code, Alohomora taps into dynamic protection mechanisms via sandboxing, and into existing software development processes like code review.

3.1 Alohomora’s Approach

Making Alohomora’s approach work requires addressing several challenges. First, Alohomora must let developers flexibly express application-dependent policies and reliably enforce them. Developers must be able to express these policies, and Alohomora must ensure they remain attached to data as it flows through the application and ultimately get enforced.
Second, Alohomora must impose low developer overhead to be practical. Developers should only have to think about policies and Alohomora’s enforcement mechanisms when they actually perform privacy-sensitive operations, like externalizing, manipulating, or combining sensitive data. Third, Alohomora and its enforcement mechanisms must be fast and add as little performance overhead as possible.

Alohomora addresses these challenges with two key ideas: policy containers and privacy regions.

A policy container is a wrapper type that statically restricts access to the contained data and associates the data with a policy. At runtime, the application can only externalize data from a policy container if its corresponding policy check passes. Static typing combined with Alohomora’s suite of static analyses ensure that programs cannot remove the policy container’s policy taint within safe Rust.

Developers express each policy type as a Rust struct, and then implement Alohomora’s Policy trait for it. The policy trait requires providing a check function that Alohomora invokes before revealing data at a sink. The check function may use metadata stored inside the policy, and can execute arbitrary code. If the policy check fails, Alohomora reports an error; otherwise, it releases the data to the sink. Application developers must associate policies with the data read from different application sources. For sources with a structured schema, application developers specify the policy associations decoratively for that schema. Applications declare the associated policies when they read data from unstructured sources, such as a cookie or GET parameter.

If the program needs to compute on or transform data in a policy container, Alohomora requires developers to use a privacy region. These regions are small, clearly-delineated regions of code that can access the raw data, but are free of side-effects, such as writing to global memory or captured variables, or externalizing data via files or the network. Alohomora enforces guarantees for privacy regions via static analysis and dynamic sandboxing. If a privacy region can be verified by static analysis (e.g., type conversion), Alohomora runs the transformation as-is, with no runtime overhead. If static analysis is unable to verify the safety of a transformation (e.g., training a machine learning model), Alohomora spawns a lightweight sandbox to prevent data from leaking and runs the privacy region in the sandbox.

Practical applications, of course, sometimes do have intentional side-effects, such as database queries, HTTP RPCs, or sending emails. Alohomora accommodates this functionality via trusted Alohomora-enabled libraries, which invoke policy checks before releasing data from policy containers, and via critical regions. Critical regions contain code that sends data to arbitrary sinks (e.g., third party libraries, custom I/O). These regions are manually reviewed and signed by developers, and Alohomora’s design strives to make them infrequent, slim, and clear to the reviewer.

Alohomora has a relaxed threat model compared to many IFC systems: Alohomora targets honest developers who make unintentional mistakes and assumes that fines deter developers from being malicious. The Rust compiler, Alohomora’s static analysis, the sandbox, and Alohomora-provided libraries are trusted, and Alohomora’s guarantees for critical regions rely on good-faith and attentive code review. Furthermore, timing and side-channel attacks are out of scope. Finally, developers should use Alohomora-provided libraries to interact with entities external to the application, such as an HTTP client or a database, to avoid frequent critical regions. This is reasonable, as mandating the use of specific libraries is common practice in organizations today.

3.2 Using Alohomora

Alohomora is a framework for web application development. To illustrate how developers use Alohomora, consider the developer of a homework submission system who adds a new HTTP endpoint for students to submit their answers. The application stores the answer in a database, and then sends the student an email confirming submission.

Figure 1 shows how a developer might implement this endpoint in Rust without using Alohomora.

This endpoint handles two types of user data: the user’s email address and the student’s submitted answer. The answer’s policy allows only the student themselves, TAs, and the instructor to view the submission, and the email address is personally-identifiable information (PII). We now look at how the developer uses Alohomora to implement this endpoint with compliance guarantees (Figure 2). The developer invokes Alohomora-enabled libraries, mandated by their organization, to look up the email address in the database (line 4), to access the answer in the HTTP request (lines 1, 6), and then to insert the policy container-wrapped answer text directly into the DB (line 7). Being Alohomora-enabled, these libraries return data wrapped in a Policy Container (PCon), and can accept Policy Containers as input. A PCon keeps the underlying data private and inaccessible to the application, and associates it with a runtime policy of a given type. For example, the answer text is protected by an AnswerAccessPolicy.

Now, the developer needs to construct the email body. This is application-specific functionality not available in an Alohomora-enabled library, and it operates on the answer data. However, accessing the answer directly will cause a compiler error, as the answer is wrapped in a Policy Container. Instead, the developer must use a Privacy Region. They invoke Alohomora’s privacy_region API, passing the Policy Container along with a closure to execute on it (lines 10–12). If Alohomora deems the closure to be safe, it invokes it on the underlying raw data. Alohomora also wraps the output of the closure in a Policy Container with an identical policy to the input, and returns it.
Alohomora’s static analysis determines that the first privacy region is safe, which makes it a pure region. But Alohomora rejects the second privacy region, as it leaks sensitive data. If the developer believes a rejected region to be safe, they can run it in a sandbox. However, this region intentionally reveals data. The developer informs Alohomora by escalating it to a critical region (CR). Before executing a CR, Alohomora checks the associated policy relative to developer-provided context information. In our example, this context is the target email address provided as an argument to critical_region, which determines who receives the data. Alohomora checks that AnswerAccessPolicy allows this email address receive the answer. If the check passes, Alohomora executes the region.

Alohomora requires a code reviewer, such as a team lead or a policy engineer, to manually review and sign the CR code. Reviewers need to verify that (i) the body of the CR is allowed under the organization’s policies; and (ii) that the body correctly uses the context information. Alohomora validates the signature given the current CR code and the reviewer’s public key. If the code in the CR changes, including any of its dependencies, the signature becomes invalid.

3.3 Privacy Regions

Alohomora provides different ways for applications to operate on data locked inside PCons. Figure 3 summarizes them: “built-in” describes Alohomora-enabled libraries, while the other three API levels correspond to privacy regions. Privacy regions allow applications to execute code on the raw data protected by PCons. Alohomora’s static analysis helps the developer determine what type of privacy region to use. At a high level, Alohomora’s static analysis checks that a closure cannot leak input data or derived data, e.g., by writing to a file, or modifying a global or captured variable.

3.3.1 Static Analysis and Pure Regions

Alohomora’s static analyzer, Scrutinizier, checks whether a closure passed to a privacy region is safe to execute. Scrutinizier is sound but incomplete: it never accepts unsafe privacy regions, but may conservatively reject safe ones. Scrutinizier searches the application code for instances of Alohomora’s privacy_region call. We refer to the closure passed to the privacy region as the top-level function. Scrutinizier 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. Scrutinizier 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 filesystem), or via mutating captured variables that other parts of the application can observe (e.g., global variables). Scrutinizier 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 Alohomora wraps the return in a PCon.

Scrutinizier uses Flowistry [7] to compute an approximation of the function’s information flow, including implicit flows and flows via helper functions call chains. Flowistry is
<table>
<thead>
<tr>
<th>API Level</th>
<th>Supports</th>
<th>Guarantees</th>
<th>Root of Trust</th>
<th>Application Developer Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in</td>
<td>Common primitives</td>
<td>Static (taint tracking)</td>
<td>Rust + Alohoma</td>
<td>Use PCon&lt; T, P &gt; instead of T with compiler guidance</td>
</tr>
<tr>
<td>Pure Region (PR)</td>
<td>Statically-proven</td>
<td>Static analysis (sound but</td>
<td>Rust + Alohoma</td>
<td>Check that Alohoma accepts closure as safe</td>
</tr>
<tr>
<td></td>
<td>safe closures</td>
<td>incomplete)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sandbox Region (SR)</td>
<td>Safe closures</td>
<td>Runtime (sandbox)</td>
<td></td>
<td>Engineering setup</td>
</tr>
<tr>
<td></td>
<td>that cannot be</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>validated statically</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical Region (CR)</td>
<td>Arbitrary closures</td>
<td>Code signing</td>
<td></td>
<td>Authorized developers review and sign closure</td>
</tr>
<tr>
<td></td>
<td>including sinks</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Alohoma’s API levels. Built-in libraries and pure regions require a minimal root of trust. Sandbox regions support more complex code at the cost of runtime overhead. Critical regions are most expressive, but rely on code review.

Alohomora’s sandbox regions use RLBox [13], a lightweight sandbox used in Firefox to execute untrusted external libraries. RLBox relies on software-based fault isolation (SFI), which uses inline dynamic checks to restrict memory reads and writes to a dedicated memory region allocated at sandbox creation time. This isolates the memory that the sandboxed region uses from the rest of the application. In addition, RLBox forbids system calls. This interface ensures that sandboxed regions cannot externalize data via I/O.

3.3.3 Critical Regions. If Scrutinizer rejects a privacy region and executing the code in a sandbox is unwieldy or it involves an intentional side-effect, developers may choose to use a critical region (CR). CRs should be small and infrequent, and are manually reviewed for privacy violations.

Alohomora requires every CR to be signed by a separate developer, such as a team lead or a privacy engineer. This mirrors existing industry practices that require approval by authorized reviewers for merging code. Alohomora intends CRs to be concise, single-purpose, and self-contained. Reviewers should ensure the CR code is consistent with the context it receives, as Alohomora’s policy check ensures that any associated policies accept that context.

Alohomora also requires a signature for the application’s dependencies—specifically, Rust’s Cargo.lock, which pins the versions of dependencies the application uses and includes checksums. This signature can be reused across CRs. By using a different signature for the dependencies, Alohomora allows approved dependency updates without having to re-sign all CRs in the application.

4 Results

4.1 Application Case Studies

We used Alohomora with four web applications that we either wrote with Alohomora in mind, or ported to Alohomora.

YouChat. YouChat is a simple online messaging application we implemented from scratch using Alohomora. YouChat users send messages to individuals or a group.

Voltron. Voltron is an application in which groups of students collaboratively edit a piece of code with instructor oversight, originally described in the Storm paper [11]. Storm’s implementation is written in LiquidHaskell, so we wrote a version of Voltron in Rust.
### 4.2 Developer Effort

Using Alohomora adds three steps to application development and porting. Developers must: (i) implement policies and associate with data, (ii) write code that uses PCons instead of raw types for sensitive data, and (iii) invoke Alohomora to determine the types of each privacy region. We now evaluate the developer effort required by these steps.

**Implementing Policies.** Figure 4 shows the size of the policy code relative to application size. This includes the policy structs, their constructors, and the Policy trait implementation. This code includes a lot of Rust boilerplate. The real crux of this code is the implementation of the CHECK functions, which we also report in the table ("CH"). The bodies of the CHECK functions are small compared to the total policy code and the size of the application.

**Using PCons.** Alohomora-enabled libraries offer nearly identical APIs to commonly used Rust crates. Because our libraries mandate PCons in their APIs, the Rust compiler/IDE guided us in changing the types read in sources to PCons during porting. We mechanically grouped adjacent operations that must operate on raw data in (potential) pure regions. Figure 4 shows that our applications needed no more than a few dozens of these regions.

**Reviewer Effort.** Figure 5 shows the number and size of critical regions that require manual review and signing. For Portfolio, our largest application, these regions make up 5% of application code. The critical regions are also small and shallow: the average lines of code that must reviewed per region, including all callees within the application code, is 24.2 for Portfolio and lower for other applications. These results suggest that Alohomora focuses reviewers’ attention on infrequent, small, and contained code regions.

### 4.3 Application Performance

We analyze performance of Alohomora with WebSubmit, comparing to a baseline without Alohomora. WebSubmit uses sandbox regions for hashing API keys during registration, and for ML model training. Endpoints that retrieve aggregate data for managers and average grades for employers deal with many users’ data, which may have different policies. We load the database with data for a medium-sized university course (100 students, 100 homework questions) and measure endpoint latency. A good result for Alohomora would show comparable latencies in endpoints with policy checks and acceptable overhead for sandboxing.

Figure 6 shows the results. Endpoints with repeated policy checks ("Get Aggregates", "Get Employer Info") are comparable to the baseline. Sandboxed endpoints ("Register Users", "Retrain Model") incur 1.1–2.3× performance overhead, mostly due to serialization cost. Grade prediction incurs 1.2× overhead, but its absolute latency is very low, as the machine learning model is stored in-memory and there is no I/O.

---

**Figure 4.** Applications LoC with and without Alohomora, the total policy code and check function implementations LoC ("CH"), and counts of each privacy regions used.

<table>
<thead>
<tr>
<th>App</th>
<th>LoC w/o</th>
<th>LoC w</th>
<th>Policy CH</th>
<th>Region Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouChat</td>
<td>800</td>
<td>1.2k</td>
<td>150</td>
<td>30</td>
</tr>
<tr>
<td>Voltron</td>
<td>500</td>
<td>1.1k</td>
<td>144</td>
<td>30</td>
</tr>
<tr>
<td>Portfolio</td>
<td>6.4k</td>
<td>6.7k</td>
<td>415</td>
<td>70</td>
</tr>
<tr>
<td>WebSubmit</td>
<td>1.3k</td>
<td>2.1k</td>
<td>84</td>
<td>20</td>
</tr>
</tbody>
</table>

**Figure 5.** Critical region count in applications. % of Total shows the proportion of critical code to application size. Average reviewer burden shows the average size of in-crate code for each critical region in LoC.

<table>
<thead>
<tr>
<th>Application</th>
<th>LoC</th>
<th># CRs</th>
<th>% of Total</th>
<th>Avg Burden</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouChat</td>
<td>1.2k</td>
<td>0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Voltron</td>
<td>1.1k</td>
<td>2</td>
<td>1.40%</td>
<td>8.0 LoC</td>
</tr>
<tr>
<td>Portfolio</td>
<td>6.7k</td>
<td>20</td>
<td>5.33%</td>
<td>24.2 LoC</td>
</tr>
<tr>
<td>WebSubmit</td>
<td>2.1k</td>
<td>2</td>
<td>1.36%</td>
<td>15.5 LoC</td>
</tr>
</tbody>
</table>

**Figure 6.** Alohomora incurs acceptable performance overheads on WebSubmit (solid: median; shaded 95th %-ile).
References


