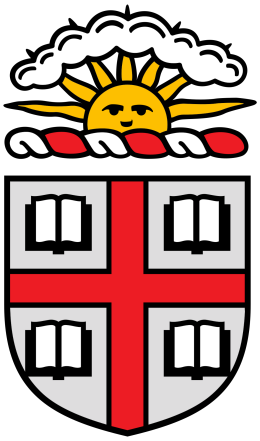
Paralanguage: A Privacy Policy Specification Language

Carolyn Zech

Advisor: Malte Schwarzkopf
Reader: Shriram Krishnamurthi



Department of Computer Science
Brown University
Providence, RI
May 2024

Contents

Abstract

Software providers handling sensitive data must verify that their applications conform to a wide breadth of privacy policies. For example, to comply with the GDPR, applications must delete a user's personal data on request.

Paralegal is a static analyzer that helps developers verify the compliance of their Rust applications. Developers write custom privacy policies for their application, which Paralegal checks against the implementation. We imagine that developers use Paralegal as part of their normal code review processes, allowing them to catch bugs before deployment. Paralegal must therefore be *practical*: it should be fast, robust to source code changes, and straightforward to use.

Writing a Paralegal policy, however, is not straightforward. Policies are Rust programs which reason about data flow between particular source code entities (e.g., arguments or types). As such, policies are closely tied to the application's implementation, which makes them difficult to write and inaccessible to all those but the application's developers.

Paralanguage is a new privacy policy specification language that seeks to reduce the developer effort required to apply Paralegal to applications. It can specify a wide range of privacy policies in an intuitive, controlled natural language format. Developers compile their Paralanguage policies to Paralegal Rust policies, which they can then run against their applications to verify compliance.

We evaluate Paralanguage on seven real-world Rust applications and find that it can express a wide variety of privacy policies, it incurs low developer effort, and that it is executes quickly enough for use in a production setting.

Chapter 1

Introduction

Privacy bugs in applications harm users (e.g., by leaking their sensitive data) and put software providers at risk for financial liability. In May of 2023 alone, Ireland fined Meta €1.2 billion for GDPR violations, while the United States fined Amazon $25 million for failing to honor data deletion requests under the Children's Online Privacy Protection Act [15, 18]. To ensure compliance, organizations must relate dense legal text to their low-level application source code. Currently, organizations use manual audits to verify compliance, which are expensive, time-consuming, and infrequent [19, 20].

Paralegal is a static analyzer for Rust applications that developers leverage to find such bugs in their programs before deployment. Paralegal evaluates an application by generating its program dependence graph (PDG), where nodes are program entities (arguments, types, etc.) and edges are control or data flow dependencies. Paralegal evaluates the PDG against the policy and outputs whether the application is compliant.

Developers can receive immediate feedback on the impact of their changes by running Paralegal continually throughout the development process, perhaps as part of a CI pipeline. We envision that Paralegal could replace (or at least reduce the frequency of) manual compliance audits.

1.1 The Problem

Paralegal policies are Rust programs that execute low-level graph queries over an application's PDG. To write these programs, developers must first connect high-level concepts (e.g., "all sensitive data is encrypted") to low-level implementation details. They must then understand how Paralegal represents these details with a PDG, and finally translate their high level policy into a series of assertions about paths in this graph. This design limits the available pool of policy writers to those with an in-depth understanding of Rust programming, application semantics, and

Paralegal's PDG representation. Since policies are programs, they have the same problems as the applications they evaluate: they are buggy, laborious to write, and comprehensible only to programmers familiar with the application.

Ideally, policy writers would specify only the high-level concepts behind their policy, which Paralegal would translate automatically into an application-specific query. We realize this goal with Paralanguage, a DSL for privacy policies over PDGs. Paralanguage is a Controlled Natural Language (CNL) interface; users write policies in natural language, but are restricted to a defined grammar [4]. Natural language offers the intuitive quality we seek, while the grammar allows us to avoid the ambiguity of unconstrained English. We pair Paralanguage with a compiler that translates policies to Paralegal graph queries in Rust. Developers then invoke Paralegal on their application, which generates its PDG, evaluates it against the compiled Paralanguage policy, and outputs whether the application is compliant.

1.2 Goals

The design of Paralanguage must address four challenges.

First, it must be accessible yet unambiguous. Paralanguage should avoid requiring policy writers to reason about low-level source code entities, such as particular functions or arguments. Such a design would tie the policy to the application's implementation, effectively restricting policy writing to the application's developers. It would also make the policies more difficult to write and brittle to source code changes. Paralanguage should instead be accessible to a range of technical stakeholders. Ideally, a compliance engineer–someone who knows how to program, but who is not necessarily familiar with the particular implementation details–would write policies in a code-agnostic way.

Natural language provides an intuitive structure for specifying these policies. However, unconstrained natural language can be ambiguous. For example, the sentence "`A` or `B` and `C`" is vague; which operator should take higher precedence? While such ambiguity may be acceptable (or even desirable) in legal text, it is unacceptable for a policy that is checkable by an automated tool. Paralanguage must reject ambiguous policies without becoming so onerous to use that it requires the same expertise and time investment as writing a graph query program.

Second, Paralanguage should be expressive. A trivial way of eliminating ambiguity would be to define a highly restricted natural language interface that allows for only a handful of policy formulations. This structure, however, would be insufficiently expressive to meet the needs of complex applications, which must abide by a wide range of privacy requirements. Instead, Paralanguage should be flexible

enough to apply across policy and application domains.

Third, Paralanguage should support sufficiently precise queries to express common privacy policies, without overwhelming the compliance engineer with the specifics of Paralegal's PDG representation. Paralegal's PDG representation represents program structure with a specific level of precision. With the existing graph query interface, compliance engineers can leverage this precision to make highly specific queries (e.g., count the exact number edges between two nodes in a graph). However, privacy policies often do not require this level of precision. If Paralanguage were to require reasoning about the PDG specifics, it would not be accessible– compliance engineers would require expertise to choose from a wide array of query patterns over a complex, unfamiliar data structure. However, if policies are insufficiently precise, they are more likely to produce false positives or false negatives when run against application code.

Fourth, Paralanguage policies, once compiled, should be similarly efficient to graph queries written in Rust. This goal is necessary so that Paralegal runs at interactive timescales when checking Paralanguage policies.

We evaluate Paralanguage against seven third-party Rust applications. We find that Paralanguage can express a wide range of privacy policies for these applications, that these policies correctly identify bugs, and that they incur acceptable run time overhead ($\approx$0-12%) compared to hand-optimized Rust graph queries.

Paralegal is the product of collaboration with Justus Adam, Livia Zhu, Sreshtaa Rajesh, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf. My contribution specifically is the Paralanguage policy language and the compilation to Rust policies.

Chapter 2

Background

Paralanguage expresses policies over Paralegal's PDG. First, we provide some background on PDGs in general, then on Paralegal's design.

2.1 PDGs

A program dependence graph (PDG) [6] represents a program's *dependencies*. The nodes are, generally speaking, any variables in the program, including function arguments and return values. The edges represent dependencies between nodes. If the PDG contains an edge $x \to y$, that indicates that y depends on x. There are two kinds of dependencies: data dependencies and control dependencies. A data dependency $x \xrightarrow{\text{data}} y$ indicates that x directly affects y, e.g., $y = x$, because data flows from x into y. A control dependency $x \xrightarrow{\text{control}} y$ indicates that x indirectly affects y, e.g., `if (x == 1) y = 1 else y = 2`. We say that x has *transitive* influence on y if there exists some path from x to y through the PDG, either through a direct edge from x to y or through intermediate nodes (e.g., $x \xrightarrow{\text{data}} z \xrightarrow{\text{control}} y$). x has transitive *data* influence on y if there is a path from x to y comprised entirely of data edges. x has transitive *control* influence on y if there is a path from x to y comprised entirely of control edges.

2.2 Paralegal

Paralegal generates PDGs for Rust applications. Users express their polices as assertions over their application's PDG. For example, an assertion may state that x must have transitive data influence on y. Paralegal provides a Rust API, called the *Graph Query API*, for common query patterns. For instance, compliance engineers use the `flows_to(x, y, EdgeSelection)` primitive to query for transitive influence, providing `EdgeSelection::Both`, `EdgeSelection::Data`, or `EdgeSelection::Control` for transitive influence, transitive data influence, or transitive control influence, respectively.

If a compliance engineer wanted to write a policy about some privacy-critical operation (e.g., data sinks), they could manually inspect the PDG to identify all relevant nodes, but that would be tedious and error-prone. Instead, they use Paralegal's *markers* abstraction to annotate their source code with custom labels. When Paralegal constructs the application's PDG, it attaches markers to the appropriate nodes. Developers apply markers to source code through inline annotations, e.g.:

```
#[paralegal::marker(sink, argument = recipient)]
fn send_email(recipient, sender, content) {... }
```

which applies a marker `sink` to the `recipient` argument of the `send_email` function. The PDG will contain one `recipient` node for each invocation of `send_email`, with `sink` applied to each of them.

Compliance engineers use Paralegal's marker abstraction to write their privacy policies. In this example, rather than explicitly enumerate the dependencies for each `recipient` node, the compliance engineer can instead specify that *all* nodes marked `sink` have some dependency. Markers make policies clearer because they allow compliance engineers to reason about concepts like `sink` or `user data`, rather than source-code level entities like function arguments.

After writing a policy and applying its markers to source code, the developer leverages their application knowledge to mark certain functions as *entrypoints* for Paralegal's analysis. Paralegal constructs PDGs for each entrypoint. Entrypoints are, generally speaking, the "privacy relevant" functions in the application. In a web application, for example, they are the user-facing endpoints where data enters and exits the application. The developer then runs Paralegal against their application, which generates the application's *marked PDGs*, i.e., its PDGs with its markers applied to the appropriate graph nodes. Paralegal checks the policy against the marked PDG and outputs whether the application is compliant.

Chapter 3

Overview

Compliance engineers and developers collaborate to apply Paralegal to their applications and catch privacy bugs before deployment. We demonstrate Paralanguage's role in this process through an example.

Freedit is a text-based social media application, similar to Reddit or Twitter [9]. The application stores a user's viewing history, but deletes it after three days [8]. To verify that the source code upholds the policy, a compliance engineer must first ensure that any viewing data is always stored alongside the current date. Otherwise, the application would have no way to check later whether the data is expired. To encode this policy in Paralanguage, the compliance engineer first considers which operations or data are relevant to the policy. They determine that the relevant concepts for this policy are *(i)* the viewing data, *(ii)* any database store operations, and *(iii)* the current date. They define `views`, `store`, and `time` markers to represent each of these ideas. The compliance engineer must have some technical background to establish these concepts–they must, for example, understand what a database is, and the kind of data that it would store. However, they do not need to understand how the source code implements this functionality, or the details of the database client library that the app uses.

Next, the compliance engineer writes the policy in terms of these markers. They know that a developer will eventually apply their markers to one or more source code entities. Their policy must therefore reason about these marked object(s). To do so, the compliance engineer introduces *variables* into their policy. A variable represents a single source code entity with a given marker. In this example, the compliance engineer introduces their variables as follows:

```
For each "view" marked views:
  For each "database store" marked store:
```

Here, and in the following text, variables are in purple and markers are in teal.

The compliance engineer must now establish that a given `"database store"` actually stores this `"view"`– if it stores some other, unrelated data, the policy does not apply. To do so, they introduce the first *predicate* into their policy:

```
For each "view" marked views:
  For each "database store" marked store:
    If "view" goes to "database store" then:
```

Predicates are expressions that relate two objects to each other–either two variables, or a variable and a marker.

Note that the compliance engineer need not specify *how* or *when* the `"view"` reaches the `"database store"`. The `"view"` could undergo any number of data transformations first–e.g., it could be inserted as a field of a type containing other user data, and that type is then stored. The compliance engineer does not worry about these implementation details; they must only understand that the `"view"` reaches the `"database store"` in some form, at some point.

The compliance engineer then defines the consequent of the conditional:

```
For each "view" marked views:
  For each "database store" marked store:
    If "view" goes to "database store" then:
      There is a "date" marked time where:
        "date" goes to "database store"
```

The consequent ensures that if the view is stored, that some date is also stored.

Finally, the compliance engineer establishes the *scope* of their policy, For instance, a GDPR data deletion policy would assert that somewhere in the application, all of a user's data is deleted. In this case, the compliance engineer determines that the entire application should uphold this policy, so they select the "Everywhere" scope, producing the final policy:

```
Scope: Everywhere

Policy:
For each "view" marked views:
  For each "database store" marked store:
    If "view" goes to "database store" then:
      There is a "date" marked time where:
        "date" goes to "database store"
```

The compliance engineer then sends this policy to a developer, who leverages their implementation knowledge to apply the markers to the appropriate source code entities. Here, they might attach the `views` marker on all function arguments of type `PageView`, the `store` marker on an argument to a database library insertion

function, and the `time` marker on the return value of a function that gets the current date. The developer runs Paralanguage's compiler, which outputs a Rust program that contains an equivalent graph query over a marked PDG and boilerplate code to invoke Paralegal. Paralegal generates the application's marked PDG, evaluates it against the policy, and outputs whether the policy passed or failed.

Chapter 4

Design

A Paralanguage policy has three sections–its body, its scope, and (optionally) its definitions:

- The body contains the predicates that define the policy.
- The scope determines which analysis entry points must satisfy the body.
- The definitions allow compliance engineers to make their bodies simpler and more efficient.

Paralanguage's full grammar is in Appendix A.

4.1 Scope

A compliance engineer has three options for the scope of their policy:

1. *Everywhere* indicates that every entrypoint must satisfy the body.
2. *Somewhere* indicates that at least one entrypoint must satisfy the body.
3. *In* `name` indicates that the entrypoint with name `name` must satisfy the body.

The appropriate scope is policy-dependent. For instance, if an application should always encrypt sensitive data before storage, the compliance engineer should select an "Everywhere" scope. For a GDPR data deletion policy, however, an "Everywhere" scope would not make sense, since that would require every entrypoint storing user data to also delete it. Instead, such a policy could use a "Somewhere" scope to mandate that at least one entrypoint deletes a user's data. Compliance engineers may wish to be more specific and ensure that a particular entrypoint obeys a policy, in which case they should use the *In* `name` scope. This scope achieves greater precision, but is also tied to the source code; if the name of the entrypoint changes, the policy must change as well.

4.2 Body

The policy body has three components: quantifiers, predicates, and conjunctions/disjunctions of them.

4.2.1 Quantifiers

Quantifiers allow compliance engineers to iterate over a collection of marked items, reasoning about one item at a time. Paralanguage provides two quantifiers: `For each` and `There is`. There are four ways to use these quantifiers, depending on whether the variable is defined or not:

(i) `For each "x" marked m:`

(ii) `There is a "x" marked m where:`

(iii) `For each "x":`

(iv) `There is a "x" where:`

(i) and (ii) iterate over all items marked `m`, while (iii) and (iv) iterate over a *defined* collection of items. Definitions allow compliance engineers to gather a collection of items ahead of time, filtering on a more complicated set of conditions than markers alone (Section 4.3).

Both types of quantifiers iterate over a collection of objects, evaluating the body in the context of the current object `"x"`. A `For each` loop succeeds if the body is true for all `"x"`. A `There is` loop succeeds if the body is true for at least one `"x"`.

Vacuity: If there are no `"x"`s, the body of a `For each` loop is vacuously true. Paralanguage allows for vacuity for `For each` loops because it evaluates policies on a per-entrypoint basis, and some entrypoints may not have certain markers. For example, take the (abbreviated) Freedit policy from §3:

```
Scope: Everywhere

Policy:
For each "view" marked views:
    [...]
```

If `For each` quantifiers enforced non-vacuity, this policy would fail on every entrypoint that does not handle `views`. To avoid these failures, the compliance engineer would have to use the *In* `name` scope to include only the entrypoints that should contain `views`. However, this solution would defeat Paralanguage's goal of being independent from source code. Paralanguage allows `For each` loops to be vacuous so that compliance engineers can use the "Everywhere" scope over the *In* `name` scope by default.

If a compliance engineer wants to a write a stricter policy with a vacuity check, they can leverage the `There is` quantifier, as in Figure 4.1. They should then change their scope to use the *In* `name` scope, so that Paralegal only evaluates the policy against the entrypoints that should contain `views`.

```
Policy:
There is a "view" marked views where:
  There is a "database store" marked store where:
    "view" goes to "database store"
and
For each "view" marked views:
  [...]
```

Figure 4.1: The Freedit policy from §3 with a vacuity check that enforces that at least one `"view"` is stored.

Variables: Observe that the loop bodies reason about the quantifier variables. If instead, Paralanguage eschewed quantifiers and wrote this policy as:

```
If a "view" marked views goes to a "database store" marked store:
  There is a "date" marked time where:
    "date" goes to a "database store" marked store
```

Paralanguage would not enforce that `"view"` and `"date"` go to the *same* `"database store"`, since the second instance of `"database store"` is again defined over all locations marked `store`. Quantifiers allow for the correct version of the policy by allowing compliance engineers to refer to the same object multiple times.

4.2.2 Predicates

Predicates are between two objects: either two variables and a variable and a marker. The full list of available predicates is in Figure 4.2. Paralanguage also supports the negation of each of these predicates, e.g. `"a" does not go to "b"`.

Paralanguage Predicate	Obligation on PDG
`"a" influences "b"`	`"a"` has transitive influence on `"b"`.
`"a" goes to "b"`	`"a"` has transitive data flow influence on `"b"`.
`"a" affects whether "b" happens`	`"a"` has transitive control flow influence on `"b"`.
`"a" goes to "b" only via "c"`	On every path from `"a"` to `"b"`, `"a"` passes through `"c"`.
`"a" goes to "b"'s operation`	`"a"` goes to the call site associated with `"b"` (e.g., if `"b"` is an argument to a call site, then `"a"` goes to any of that call site's arguments).
`"a" is marked m`	`"a"` is marked `m`.

Figure 4.2: Paralanguage's predicates and the obligations they enforce on Paralegal's marked PDG.

4.2.3 Numbered Clauses

In our examples thus far, we have used indentation to nest quantifiers. However, such a design is error-prone–with just one accidental indentation, a compliance engineer would write an entirely different policy than what they intended. For instance, take the policies in Figure 4.3, which differ only in indentation but have different meanings. Policy 4.3(a) will fail if there is no `"x"`. Policy 4.3(b), however, can still pass if `"y"` goes to `"z"`.

Rather than allow a stray indent to change the meaning of the policy, Paralanguage instead enforces that compliance engineers explicitly specify the scope of each statement. They do so with *numbered clauses*, inspired by similar notation in legal text. Policies 4.3(c) and Policies 4.3(d) are equivalent to Policy 4.3(a) and Policy 4.3(b), respectively, but they use Paralanguage numbered clauses to make the scope of each statement explicit.

A numbered clause is followed by an quantifier or a predicate. To introduce an additional numbered clause at the same level, compliance engineers use `and` or `or` operators, which indicate an unambiguous connection between the clauses. Compliance engineers are not permitted to mix operators (`and`s and `or`s) on the same numbered clause level, since the operator precedence in such cases would be ambiguous.

```
There is a "x" where:
  "x" goes to "y"
  or
  "y" goes to "z"
```

(a)

```
There is a "x" where:
  "x" goes to "y"
or
"y" goes to "z"
```

(b)

```
1. There is a "x" where:
  A. "x" goes to "y"
  or
  B. "y" goes to "z"
```

(c)

```
1. There is a "x" where:
  A. "x" goes to "y"
or
2. "y" goes to "z"
```

(d)

Figure 4.3: Policies with identical syntax but different scopes. (a) and (b) use indentation to indicate scope, while (c) and (d) make the scope explicit through numbered clauses. (These policies are partial; we elide quantifiers for brevity.)

4.2.4 Other Details

Conditionals: Privacy policies often impose *conditional* obligations, e.g., *if* an application stores sensitive data, *then* it must obtain a user's consent. Paralanguage supports conditionals through its `If p, then: q` construct, where `p` is a predicate and `q` is either a predicate, quantifier loop, or another conditional. Compliance engineers leverage Paralanguage's numbered clauses to specify when `q` begins and ends. For example, in Figure 4.4, the antecedent `"x" goes to "y"` begins with numbered clause indicator `a.`. Thus, the consequent `q` is every statement between `then:` and the next statement on the same "clause level" as the antecedent, which is indicated with a `b.`.

```
1. For each "x":
  A. For each "y":
    a. If "x" goes to "y", then:
      i) For each "z":
        A) "z" does not go to "y"
        and
        B) [...]
      or
      ii) [...]
    b. [...]
```

Figure 4.4: A Paralanguage policy that uses a conditional (bolded).

Types: Paralanguage intentionally abstracts away details of the PDG to make policies easier to write. Namely, Paralanguage avoids reasoning about particular source code entities (e.g., arguments, functions). The one exception to this rule is datatypes. To understand why this exception is necessary, consider the following data deletion policy:

```
1. For each "sensitive" type marked user_data:
  A. There is a "source" that produces "sensitive" where:
    a. There is a "deleter" marked deletes where:
      i) "source" goes to "deleter"
```

This policy enforces that for each user data type, some data of that type is deleted. Consider how a compliance engineer would write this policy without the `type` keyword. They could write that `There is a "source" marked user_data` that is deleted, but only checks that at least one `user_data` type is deleted, not that

all of them are. They could approximate the `type` keyword by giving each type a unique marker and using `There is` quantifiers for each of them, but such a policy would be tedious to write and easy to get wrong.

4.3 Definitions

In the Freedit example from §3, the compliance engineer's final policy was:

```
1. For each "view" marked views:
  A. For each "database store" marked store:
    a. If "view" goes to "database store" then:
      i) There is a "date" marked time where:
        A) "date" goes to "database store"
```

This policy contains five levels of nesting. As policies get more complex, many levels of nesting can make policies harder to understand. To address this issue, Paralanguage allows compliance engineers to create *definitions*. A definition declares a variable ahead of time which refers to all items that meet a certain condition. Observe that the Freedit policy does not enforce any obligations on a `"database store"` unless it stores a `"view"`. Rather than iterate through *all* database stores, a compliance engineer can collect only the relevant database stores up front, then write their policy in terms of those. In this case, the Freedit compliance engineer would create the following definition:

```
1. "view store" is each "store" marked db_store where:
  A. There is a "view" marked views where:
    a. "view" goes to "store"
```

and revise their policy to:

```
1. For each "view store":
  A. There is a "date" marked time where:
    a. "date" goes to "view store"
```

Note that the latter policy is also more efficient because it avoids the double `For each` loop of the original. While we do not expect efficiency to be a compliance engineer's primary concern, the definitions feature may be a useful way to improve performance if developers find the original policy to be too slow in practice.

Since Paralegal constructs per-entrypoint PDGs, Paralanguage policies, once compiled, are evaluated against one entrypoint at a time. The scope of the policy dictates which entrypoint(s) must uphold the policy. Since policy bodies are entrypoint-specific, definitions are entrypoint-specific by default as well.

However, a compliance engineer may want to gather all marked items meeting a certain condition *across entrypoints*. For instance, consider a policy that states

that for each `sensitive` type that the application `store`s, that type is also `delete`d. It is unlikely that a single entrypoint would both store the sensitive data *and* delete it. Instead, the compliance engineer could declare a definition to gather the sensitive types that are stored across the application, then write a policy that states that some entrypoint must delete those types. They do so by appending "anywhere in the application" to their definition declaration (Figure 4.5).

```
Scope: Somewhere

Definitions:
1. "stored sensitive" is each "sensitive" type marked sensitive where,
anywhere in the application:
  A. There is "database store" marked store where:
    a. "sensitive" goes to "database store"

Policy:
1. For each "stored sensitive":
  A. There is a "deleter" marked deletes where:
    a. "stored sensitive" goes to "deleter"
```

Figure 4.5: A data deletion policy which uses a definition to gather all sensitive types from each of the application's entrypoints.

Chapter 5

Implementation

5.1 Compiler

The Paralanguage compiler translates policies into graph queries over Paralegal's PDG, which developers run against their application. The compiler first parses the policy into an abstract syntax tree (AST). It then traverses this AST to verify that the policy is properly scoped. It errors if any variables are used in a predicate without first being introduced by an quantifier or definition. It also prohibits duplicate introductions of the same variable in the same scope.

Once the compiler verifies that the policy is properly scoped, it performs a second pass over the AST to compile it to Rust code. This compilation happens via template expansion. The compiler identifies the relevant template for each node in the AST, substituting in the policy's variables and markers.

`For each` and `There is` loops compile to iterators over nodes with the given marker in the entrypoint's PDG. For example, the statement `For each "view" marked views` compiles to an iterator over all nodes marked `views`. The `and`, `or`, and `if` constructs correspond directly to Rust builtin operators.

Predicates compile to expressions using Paralegal's Graph Query API. For example, `"x" goes to "y"` compiles to `flows_to(x, y, EdgeSelection::Data)`.

The compiler outputs a Rust program containing the policy. This program contains boilerplate code to invoke Paralegal against the policy, which the developer edits to provide application-specific information, such as the application directory and optional Paralegal configuration flags. Once the developer is ready to check their policy against their application, they compile and run this program.

We implemented a prototype of the Paralanguage compiler in 2,370 lines of Rust.

5.2 Limitations

This prototype has some limitations which could be overcome with further engineering effort. We detail those below.

5.2.1 Direct Dependencies

Paralanguage predicates reason about *transitive* dependencies between PDG nodes. There is no way for compliance engineers to enforce a *direct* edge between two nodes. In most cases, transitive dependencies are the correct level of abstraction. For example, given the Rust code `sink(*sensitive)`, one might reasonably conclude that `sensitive` goes to `sink`. However, since `sensitive` is dereferenced, the PDG contains an intermediate node between `sensitive` and `sink`, so there is no direct data flow edge between `sensitive` and `sink`. However, it is possible that a developer would want to forbid any intermediate data transformations to sensitive data before it reaches some `sink` (perhaps in a security-critical setting). Paralanguage cannot express this policy, while Paralegal's Graph Query API can.

5.2.2 Code Improvements

The Paralanguage compiler currently lacks optimization passes, so it misses opportunities for efficiency improvements. For example, if a policy contains two quantifiers over the same data, the Paralanguage compiled policy will traverse the PDG twice, when it could have just done one traversal and stored the result for the subsequent iteration.

The Graph Query API also has a sophisticated error message framework, which allows developers to print which source code locations are causing policy failure. Paralanguage's policies currently only print whether the policy was successful, but the compiler's templates could be revised to support this error message framework.

5.2.3 Other Limitations

Our Paralanguage prototype supports five levels of numbered clauses, but more could be added with additional engineering effort. The prototype also only supports iterating over defined variables and variables marked `m`. If Paralanguage allowed filtering on predicates, e.g. `For each "x" that goes to "y"`, its policies would be more concise.

Chapter 6

Evaluation

We evaluate Paralanguage against seven third-party Rust applications to answer four questions:

1. Can Paralanguage's grammar express real-world privacy policies? (§6.1)
2. How do Paralanguage's abstractions impact policy precision and correctness?(§6.2)
3. Are Paralanguage's policies efficient enough for practical use? (§6.3)
4. What is the effort required to encode policies in Paralanguage? (§6.4)

We tried to pick popular applications spanning different policy domains. We summarize the applications in Figure 6.1.

6.1 Expressivity

We found that Paralanguage could express all of the policies that we defined for these applications. In cases where the policy was inherently dynamic, we defined static approximations. For example, a GDPR data deletion policy would state that some entrypoint deletes *all* of a user's data. Paralegal cannot verify that the application actually deletes all of the user's data, since the exact contents of that data are only known at runtime. However, it can ensure that for each type marked

Application	**Type**	**LoC**	**Policies**
Atomic [2] (v0.34.2)	Graph DB	9.6k	Access Control
Contile [3] (v1.11.0)	Advertising	4.9k	Purpose Limitation
Freedit [9] (v0.6.0-rc.3)	Social	6.6k	Data Retention/Expiration
Hyperswitch [11] (v0.2.0)	Payments	198.9k	Credential Security, Limited Data Collection
Lemmy [12] (v0.16.6)	Social	31.4k	Access Control
Plume [14] (v0.7.2)	Blogging	21.4k	Data Deletion
WebSubmit [16] (v1.0)	Homework	1.6k	Data Deletion, Access Control

Figure 6.1: Case study applications with code size and policies.

`user_data`, there is some data of that type that goes to a `"deleter"`. This policy is expressive enough to find bugs where applications forget to delete a given type of user data, but cannot catch bugs where an application only deletes *some* of the data of a given type.

While we were able to express all of the policies for these applications, Paralanguage has limitations that prevent it from expressing every privacy policy. For example, our Paralanguage prototype cannot express policies that rely on direct dependencies or more than five levels of nested expressions (Sections 5.2.1 and 5.2.3).

6.2 Precision

Paralanguage policies support a lower degree of precision than native Rust polices (Chapter 4 and section 5.2.1) in exchange for greater accessibility. We evaluate to what extent this loss of precision affects the accuracy of Paralanguage policies.

For each application, we wrote Graph Query API policies and equivalent Paralanguage policies. The Graph Query API policies leveraged functionality that is out of scope for Paralanguage (e.g., reasoning about the direct siblings of a marked node). We ran these policies on compliant and incompliant versions of the applications. We found that both sets of policies were correct for every application, i.e., passed for the compliant versions and failed for the incompliant ones. This result is a promising indicator that Paralanguage's reduced precision is acceptable in practice.

6.3 Efficiency

For each application, we compare the total execution time of its Graph Query API policies and its Paralanguage policies. To avoid the variance of a single run unduly affecting the result, we average the results over 10 runs each. We would expect Paralanguage policies to be slower on average because a human developer can optimize their graph queries, while our prototype compiler outputs unoptimized code (§5.2.2).

Figures 6.2 and 6.3 compare the Paralanguage and Graph Query API policy execution times. We found that Paralanguage policies are 2-12% slower than their Graph Query API counterparts. The one exception is WebSubmit, which was 0.5% faster than the Rust API policies. However, the WebSubmit policies run quickly (≈30ms), so in any given run, a difference of a few milliseconds causes this percentage to vary widely.

These results demonstrate that Paralanguage policies incur an acceptable overhead compared to hand-optimized graph queries.

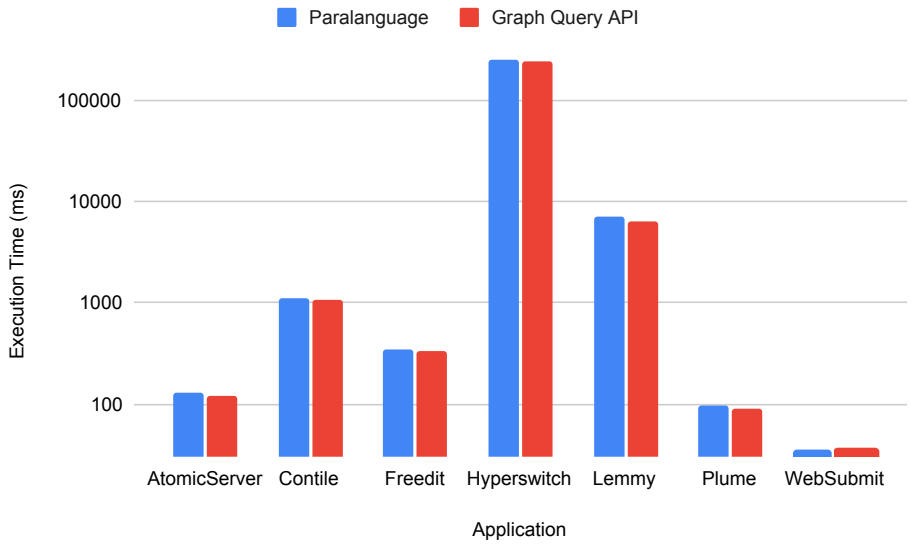


Figure 6.2: Paralanguage and Graph Query policy execution times, on a logarithmic scale, averaged across 10 runs. The Paralanguage execution times are similar to those of hand-optimized queries.

Application	**Paralanguage Time (ms)**	**Graph Query API Time (ms)**	**Paralanguage Percent Slower**
AtomicServer	127.92	118.8	7.68
Contile	1109.14	1064.43	4.2
Freedit	347.57	339.85	2.27
Hyperswitch	251044.37	241890.42	3.78
Lemmy	7026.23	6277.3	11.93
Plume	95.63	88.86	7.62
WebSubmit	29.89	30.07	-0.6

Figure 6.3: Comparison of Paralanguage and Graph Query API policy execution times. Paralanguage policies are, on average, marginally slower than the Graph Query API policies.

6.4 Effort

We evaluate the effort required to encode policies in Paralanguage. Ideally, we would have conducted a user study with users unfamiliar with the system. However, due to time limitations, we instead summarize our experience (as the authors of Paralegal) writing policies.

Policy Strictness: We find that the effort required to encode policies in Paralanguage is proportional to the strictness of the policy. For example, take the AtomicServer application, a graph-based database [2]. A previous version of the application failed to verify that the user was permitted to modify a database resource before applying the update [1]. We encode this policy in Paralanguage by enforcing that if a database resource is modified, the application checks which users are permitted to modify the resource, and that check affects whether the update happens. This policy fails on the buggy version of the application and passes on the fixed version.

Prior to this work, we wrote a version of this policy in the Graph Query API. This policy was stricter: it also included the implementation-specific notion of *commits*. Commits are records of modifications to a resource (analogous to Git commits). Our Graph Query API policy enforced that a commit had transitive data influence on the modified resource. However, this assertion is not necessary to catch the permission check bug.

Throughout this work, we found multiple other instances where our Paralanguage policies were simpler than the Graph API policies we wrote earlier. Since the Graph Query API allows developers access to the full expressive power of the PDG, we often found ourselves referencing application source code to ensure that we selected the right primitives. As a result, we unwittingly wrote policies that were more closely tied to the source code than actually necessary to express the policy we sought to check. When we wrote the Paralanguage policies, we had a much smaller subset of predicates available. We found that by abstracting away the full power of the PDG representation, Paralanguage encouraged us to think about the minimum set of concepts necessary to express our policies, so our policies did not contain these unnecessary checks.

If a compliance engineer does want to check in their Paralanguage policy that the modified resource is indeed affiliated with a commit, they will need to expend more effort. They may need to consult with the developer to understand the implementation-specific notion of commits and how they interact with resources. The developer would also need to apply (and maintain) more markers to the application.

Numbered Clauses: We found that for some of the policies, our first attempt to express them required more levels of numbered clauses than Paralegal supports (Section 5.2.3). We refactored the policies to use definitions, which allowed us to express the policies with fewer nested expressions. We found that this process made our final policies easier to read, even if it took us longer to write them.

Chapter 7

Related Work

A privacy policy specification language should be **accessible**, **expressive**, and **precise**. Existing systems meet at most two of these goals.

Program-based specifications allow developers to write precise and expressive policies. In Resin [23], developers specify dynamic dataflow assertions in their application code. Ponder [5] also encodes policies through code; developers specify assertions that must hold or code that should run if a given predicate is satisfied. Other interfaces, such as IFC [13] or regular expression-based rules [10], allow for precise reasoning over programs without writing code, but still require a strong technical background.

Legalese [17] is a privacy policy specification language that, like Paralanguage, is structured around allowing or disallowing information flows over a data dependency graph. Legalese can only express whether one node does or does not flow to another–it cannot express control flow or the order of nodes in a dataflow path, and therefore has more limited expressivity than Paralanguage. While Legalese is meant to be more accessible than program-based specifications, user surveys have found that non-programmers struggle to define or interpret Legalese policies [17, 22]. Paralanguage borrows the concept of numbered clauses from the law in part to make policies more accessible to nontechnical stakeholders (e.g., lawyers) familiar with that structure.

Riverbed [21] and RuleKeeper [7] offer more accessible policy specifications. However, they are very limited to particular domains–Riverbed encodes four binary allow/disallow attributes, while RuleKeeper only supports a particular subset of GDPR provisions. Paralanguage is more flexible; as long as the compliance engineer can express their policy in terms of data and control dependencies, they can encode it in Paralanguage.

Chapter 8

Conclusion

Paralanguage is a privacy policy specification language that seeks to be accessible, expressive, precise, and performant. Compliance engineers can express high-level policies about flows between marked entities in an application, without requiring in-depth knowledge of the underlying PDG or application source code.

Our evaluation demonstrates that Paralanguage is expressive enough to represent a variety of real-world privacy policies, that its compiler outputs correct, precise, and performant policies.

Bibliography

[1] *AtomicServer (Initial Bug Fix): Prevent Unauthorized Commits*. URL: https://github.com/atomicdata-dev/atomic-server/commit/46a503a (visited on 04/19/2024).

[2] *AtomicServer: a lightweight, yet powerful CMS / Graph Database*. URL: https://github.com/atomicdata-dev/atomic-server.

[3] *Contile: The back-end server for the Mozilla Tile Service*. URL: https://github.com/mozilla-services/contile (visited on 04/19/2024).

[4] *Controlled Natural Language*. URL: http://www.sigcnl.org/ (visited on 02/01/2023).

[5] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. "The Ponder Policy Specification Language". In: *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. POLICY '01. Berlin, Heidelberg: Springer-Verlag, 2001, pages 18–38.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pages 319–349.

[7] Mafalda Ferreira, Tiago Brito, José Santos, and Nuno Santos. "RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks". In: May 2023, pages 2817–2834.

[8] *Freddit: Pageview data retention*. URL: https://github.com/freedit-org/freedit/blob/f5905db9ea3c8630d61c80143d5f2553ee654b15/src/controller/user.rs#L1096 (visited on 04/19/2024).

[9] *Freedit: The safest and lightest forum, powered by rust.* URL: https://github.com/freedit-org/freedit (visited on 04/19/2024).

[10] Karuna Grewal, P. Brighten Godfrey, and Justin Hsu. "Expressive Policies For MicroService Networks". In: HotNets '23 (2023).

[11] *Hyperswitch: An open source payments switch written in Rust to make payments fast, reliable and affordable*. URL: https://github.com/juspay/hyperswitch.

[12] *Lemmy: A link aggregator and forum for the fediverse*. URL: https://github.com/LemmyNet/lemmy.

[13] Andrew C. Myers and Barbara Liskov. "Protecting Privacy Using the Decentralized Label Model". In: *ACM Trans. Softw. Eng. Methodol.* 9.4 (Oct. 2000), pages 410–442.

[14] *Plume: A federated blogging engine based on ActivityPub*. URL: https://github.com/Plume-org/Plume.

[15] Adam Satariano. *Meta Fined $1.3 Billion for Violating E.U. Data Privacy Rules*. May 2023. URL: https://www.nytimes.com/2023/05/22/business/meta-facebook-eu-privacy-fine.html?unlocked_article_code=1.nk0.KFcs.zTTKmSZwMUhz&smid=url-share (visited on 04/27/2024).

[16] Malte Schwarzkopf. *Websubmit-rs: A Simple Class Submission System*. Oct. 2021. URL: https://github.com/ms705/websubmit-rs.

[17] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. "Bootstrapping Privacy Compliance in Big Data Systems". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pages 327–342.

[18] Natasha Singer. *Amazon to Pay $25 Million to Settle Children's Privacy Charges*. May 2023. URL: https://www.nytimes.com/2023/05/31/technology/amazon-25-million-childrens-privacy.html?unlocked_article_code=1.nk0.fGdA.IFRM2SYu7oGe&smid=url-share (visited on 04/27/2024).

[19] Oliver Smith. *The GDPR Racket: Who's Making Money From This $9bn Business Shakedown*. URL: https://www.forbes.com/sites/oliversmith/2018/05/02/the-gdpr-racket-whos-making-money-from-this-9bn-business-shakedown/ (visited on 02/01/2023).

[20] *The Cost of Continuous Compliance*. Feb. 2020. URL: https://www.datagrail.io/resources/reports/gdpr-ccpa-cost-report/ (visited on 02/01/2023).

[21] Frank Wang, Ronny Ko, and James Mickens. “Riverbed: Enforcing User-Defined Privacy Constraints in Distributed Web Services”. In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI’19. Boston, MA, USA: USENIX Association, 2019, pages 615–629.

[22] 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: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pages 3753–3770.

[23] 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 ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pages 291–304.

Appendix A

Grammar

⟨*paralegal policy*⟩ ::= `Scope:` ⟨*scope*⟩ [`Definitions:` ⟨*definitions*⟩] `Policy:` ⟨*exprs*⟩

⟨*definitions*⟩ ::= ⟨*definition*⟩ ⟨*definitions*⟩ | ⟨*definition*⟩

⟨*definition*⟩ ::= ⟨*numbered clause*⟩ ⟨*variable*⟩ `is each` ⟨*variable_intro*⟩ `where:` (⟨*exprs*⟩ | ⟨*body*⟩)

⟨*scope*⟩ ::= `Everywhere:` | `Somewhere:` | `In` <name>:

⟨*exprs*⟩ ::= ⟨*clause*⟩ ⟨*operator*⟩ ⟨*exprs*⟩ | ⟨*clause*⟩ | ⟨*only via predicates*⟩

⟨*numbered clause*⟩ ::= ⟨*number*⟩. | ⟨*number*⟩) | ⟨*letter*⟩. | ⟨*letter*⟩)

⟨*operator*⟩ ::= `and` | `or`

⟨*clause*⟩ ::= ⟨*clause intro*⟩ ⟨*clause body*⟩

⟨*clause intro*⟩ ::= ⟨*for each*⟩ | ⟨*there is*⟩

⟨*for each*⟩ ::= ⟨*numbered clause*⟩ `For each` ⟨*variable intro*⟩ `:`

⟨*there is*⟩ ::= ⟨*numbered clause*⟩ `There is a` ⟨*variable intro*⟩ `where:`

⟨*variable intro*⟩ ::= ⟨*variable*⟩ `input`
- | ⟨*variable*⟩ `item`
- | ⟨*variable*⟩
- | ⟨*variable*⟩ `marked` ⟨*marker*⟩
- | ⟨*variable*⟩ `type marked` ⟨*marker*⟩
- | ⟨*variable*⟩ `that produces` ⟨*variable*⟩

⟨*clause body*⟩ ::= (⟨*clause*⟩ | ⟨*body*⟩) ⟨*operator*⟩ ⟨*clause body*⟩
| ⟨*clause*⟩
| ⟨*body*⟩

⟨*body*⟩ ::= ⟨*numbered clause*⟩ ⟨*predicate*⟩ ⟨*operator*⟩ ⟨*body*⟩
| ⟨*conditional*⟩
| ⟨*numbered clause*⟩ ⟨*predicate*⟩

⟨*conditional*⟩ ::= ⟨*numbered clause*⟩ `If` ⟨*predicate*⟩ `then:` ⟨*clause body*⟩

⟨*only via predicates*⟩ ::= ⟨*only via predicate*⟩ | ⟨*only via predicate*⟩ ⟨*operator*⟩ ⟨*only via predicates*⟩

⟨*only via predicate*⟩ ::= `Each` ⟨*variable intro*⟩ `goes to a` (⟨*variable*⟩ | ⟨*variable*⟩ `marked` ⟨*marker*⟩) `only via a` (⟨*variable*⟩ | ⟨*variable with marker*⟩) `marked` ⟨*marker*⟩

⟨*predicate*⟩ ::= ⟨*variable*⟩ `does not influence` ⟨*variable*⟩
| ⟨*variable*⟩ `influences` ⟨*variable*⟩
| ⟨*variable*⟩ `goes to` ⟨*variable*⟩
| ⟨*variable*⟩ `does not go to` ⟨*variable*⟩
| ⟨*variable*⟩ `affects whether` ⟨*variable*⟩ `happens`
| ⟨*variable*⟩ `does not affect whether` ⟨*variable*⟩ `happens`
| ⟨*variable*⟩ `is marked` ⟨*marker*⟩
| ⟨*variable*⟩ `is not marked` ⟨*marker*⟩
| ⟨*variable*⟩ `goes to` ⟨*variable*⟩'s `operation`