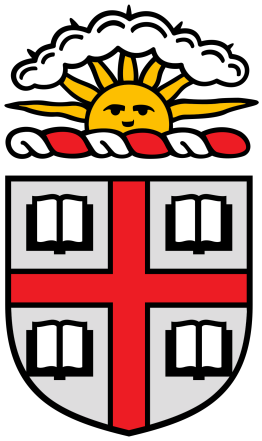
Enhancing the Expressiveness and Efficiency of Privacy Linting

Livia Zhu

Advisor: Malte Schwarzkopf
Reader: Shriram Krishnamurthi



Department of Computer Science
Brown University
Providence, RI
December 2023

Contents

Acknowledgments

Thank you to my research advisor, concentration advisor, professor, two-time boss, life mentor, and friend, Malte. Malte's kindness, passion for systems, and unwavering faith in me have changed my life in indescribable ways over the past three years.

Thank you to my reader, Shriram, for his support; I am so glad to have had his incredible teaching as my first introduction to CS at Brown.

Thank you to the Paralegal team who have been such a joy to work with. Justus, for his patience, guidance, brilliance, and Compliments Jar. Sreshtaa, for her passion and friendship, and for sticking with me through 3 research projects over 2 years. Carolyn, for her friendship, diligence, and advice. Ben, for his friendship, generosity, and humor.

Thank you to my parents, my sister, the rest of the ETOS group, and all of the friends I have made at Brown. Everything I've accomplished is because of you.

Abstract

A privacy linter is a new static program analysis tool that helps developers check that software conforms to privacy policies. Developers use the linter to catch privacy violations during the development process before they can impact end users. Our prototype privacy linter, Paralegal, is based on a dependency analysis that captures the information needed for common privacy policies and leverages the Rust type system to enhance the precision of the analysis.

This thesis presents work to improve the expressiveness and efficiency of Paralegal. The initial Paralegal prototype lacked the expressiveness to support checking essential privacy policies like encryption-at-rest and access control, and the mechanism it used to check policies was inefficient. Both of these issues severely hindered the adoptability of the system.

This work is based on two key insights. First, adding history-preservation and distinguishing data- and control-flow dependencies in Paralegal's program representation – creating a Program Dependence Graph (PDG) – is crucial to check key privacy policies. Second, the original model-checker-based backend was slow and a study of real-world policies can be expressed as graph queries over the PDG. These graph queries can then be efficiently executed in Rust. Based on these insights, we modified Paralegal's analysis to add history-preservation and distinguish control dependencies, and defined and implemented the Rust Policy API, a library for writing policies in Paralegal.

We evaluate Paralegal's enhanced expressiveness and efficiency on a variety of Rust applications and find that enhanced expressiveness is needed in six out of seven applications, and that using a graph query backend increases the speed of policy checking by $23 - 86\times$ across three applications.

1 Introduction

Protecting user privacy has been solidified as a top priority for companies, particularly with the introduction and enforcement of laws like the GDPR and CCPA, under which companies have been fined upwards of $1 billion for privacy violations [19].

Privacy policies that companies would like to enforce might include, for example, users' right to access and delete their data under the GDPR, time-limited data retention, or the confidentiality of sensitive data. However, these policies are hard to get right in complex applications for even well-intentioned developers. Today, organizations must rely on manual audits by privacy experts or external consultants to check if their code respects privacy policies. Naturally, manual audits are laborious, error-prone, and unlikely to happen frequently, and attempting compliance with the GDPR has cost companies billions of dollars [26, 27].

A *privacy linter* is a new tool that warns developers of potential privacy problems they introduce *during development*. We imagine that engineers regularly run the privacy linter on their code base during the software development process as part of the CI (continuous integration) process or an IDE plugin. The tool flags a range of privacy issues so that developers can address them before deployment. Privacy linters must emphasize utility, which depends on a balance of four qualities. The privacy linter must:

1. require little user effort in writing policies, adding code annotations, or modifying code;
2. accurately detect privacy incompliance, including predictably detecting real violations and reducing the number of false positives on supported code;
3. have the expressiveness to capture a wide range of desirable privacy policies; and
4. efficiently check complex policies for production-level code bases.

This thesis focuses on the last two qualities. When this work began, our prototype privacy linter, Paralegal, was limited in its expressivity and unable to articulate important privacy policies such as enforcing authorization checks and encryption-at-rest. Furthermore, it was slow and struggled with larger code cases, making it untenable in an interactive setting like an IDE plugin.

1.1 The Problem

At the beginning of this work, Paralegal used two key systems that are necessary to understand as context for this thesis: 1) Flowistry [5] for data-flow analysis, and 2) the SAT-backed Forge modeling language and checker to verify policies.

Dependency analysis with Flowistry. Flowistry, a dependency analysis tool, finds explicit and implicit data dependencies between instructions in Rust code, outputting a dependency relation [5]. For every pair of instructions, a dependency relation shows one of two things: an

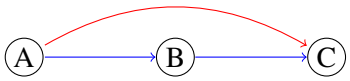


Figure 1: The transitive dependency graph between A, B, and C.

instruction either influences the other instruction, or it doesn't. In graph terms, the relation is the transitive closure of the program's dependencies. Consider Figure 1, which shows the transitive closure of dependency relationships between instructions *A*, *B*, and *C* as a graph. However, instructions can influence other instructions in a program in multiple ways. The transitive representation obscures whether *C* depends on *A* solely through the blue edges or whether there is a direct dependency as well, shown by the red edge. Some privacy policies require such a distinction – for example, a user may want to ensure that a database store never has a direct flow from user input, but can only depend on user input via encryption. We call this property *history-preservation*: for an instruction, we want to preserve precisely how its dependencies arose, whether it was directly or via other dependencies. Paralegal's dependency analysis must therefore capture non-transitive dependencies, i.e., become *history-preserving*.

Furthermore, Flowistry outputs all dependencies together, not distinguishing between control dependencies (also called implicit data dependencies) and explicit data dependencies. Implicit data dependencies are a type of data dependency, since they can leak information about data in a program. A distinction between the two types of dependencies is desirable for some privacy policies.

Policy specification in Forge. Paralegal originally used the Forge language to express privacy policies as properties of a dependency graph in first-order logic [8]. After combining these policies with the graph from Flowistry, Paralegal used Forge's SAT solver backend to determine whether the policies were violated or satisfied. Though Forge is extremely flexible, Paralegal presents a class of problems that Forge is not optimized for, so its run-time and memory usage scaled very poorly, largely due to the slow parsing of the large relations representing dependencies in Forge's front-end. We found that when analyzing multiple functions for some complex applications, Forge took dozens of minutes to run and often failed after running out of memory, so the original Paralegal prototype required developers to analyze their program in parts.

1.2 Proposed Solution

In this work, we address Paralegal's limitations in expressivity and efficiency.

We modify Flowistry's analysis to output a dependency *graph*, which, unlike a relation, contains only non-transitive relationships between nodes, which makes it history-preserving. Paralegal also uses basic block information from the Rust MIR to differentiate edges as either data-flow or control-flow dependencies. This representation of the program is called the *Program Dependence Graph* (or PDG), a standard representation in program analysis.

We observe that in a corpus of 19 policies across seven real-world applications, policies only consisted of questions about reachability and recognizing patterns in the PDG. For these policies, Forge's first-order logic is unnecessarily powerful. We instead create the Rust Policy API, which is a library containing a set of graph operations over the PDG for writing privacy policies. These operations then execute efficiently as graph queries in Rust.

Therefore, this thesis presents the following contributions:

1. We propose utilizing a PDG as Paralegal's code model to add history-preservation and control-flow dependencies (§3.1).

2. We propose the Rust Policy API, a set of policy primitives for writing privacy policies for Paralegal that comprise graph operations that run over the PDG (§3.2).

3. We implement the PDG construction and the policy primitives as part of the Paralegal prototype (§4).

4. We evaluate these augmentations to Paralegal using our prototype. We evaluate the privacy policies that are enabled by history-preservation and control-flow, and evaluate the run-time of policies written in Rust compared to equivalent policies in Forge (§5).

This work was done in collaboration with others from the Paralegal team: Justus Adam, Sreshtaa Rajesh, Carolyn Zech, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf. My specific contributions are the conceptualization, motivations, example programs and policies for the control flow differentiation and history-preservation, the implementation of differentiating the control flow relation in Paralegal, the conceptualization and a large part of the implementation for the Rust Policy API, and the policy implementation and evaluation for the main three applications discussed. In the rest of this thesis, I will explain work that was done by collaborators as is necessary to understand my contributions.

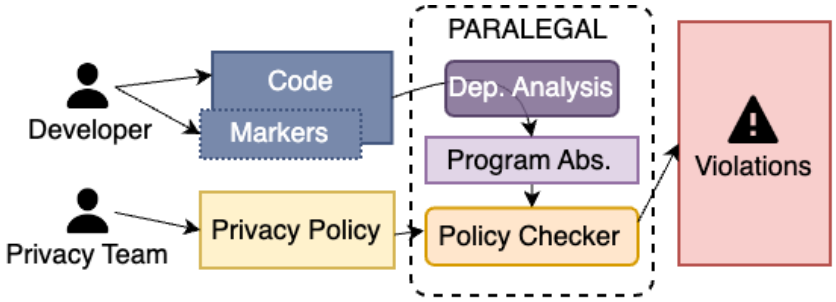


Figure 2: The system architecture of Paralegal: developers enhance their code with markers, which Paralegal checks against properties written by a privacy team and reports violations.

2 Background

2.1 Paralegal System Architecture

Paralegal helps developers catch privacy violations by extracting a model of the code using static analysis and checking it against a privacy policy (Figure 2).

First, a privacy team, which may include non-technical users like lawyers and product managers, defines appropriate privacy policies by considering relevant regulations, organization-specific policies, and application behavior. These policies could, e.g., guarantee that user data is exhaustively deleted or ensure that proper access control checks are enforced before modifying or storing sensitive data. Consider an example where the privacy team wants to specify that user data can be deleted at the user's request. The policy ought to be stated at a high level so that the privacy team can write it without intimate knowledge of the code and so that it is robust to program changes. To bridge the gap between high-level concepts (like "user data" or "deletion", which could be implemented in any number of ways) and the concrete source code, the privacy team must specify a set of *markers* to use in the privacy policies. In our example of allowing users to delete their data, the privacy team can specify a policy which is the following (in English): "A node marked `user_data` flows into a node marked `deletes`".[1] They implement this policy using the Rust Policy API described in §3.2.2.

Next, a developer must apply these markers to their code to imbue it with privacy-specific semantics into their code. The developer marks types that hold user data as `user_data` and functions that delete data as `deletes`. They must also specify which function entry points in the code Paralegal should target in its analysis – in a web application, these would be all of the web controllers, i.e., functions that handle HTTP requests. Listing 1 shows an example of marked source code, which contains a deletion endpoint where a user can delete themselves from the database.

[1]This property is necessary but not sufficient to guarantee that the higher-level property holds. For example, nothing in this property specifies that this flow happens in response to a user's request.

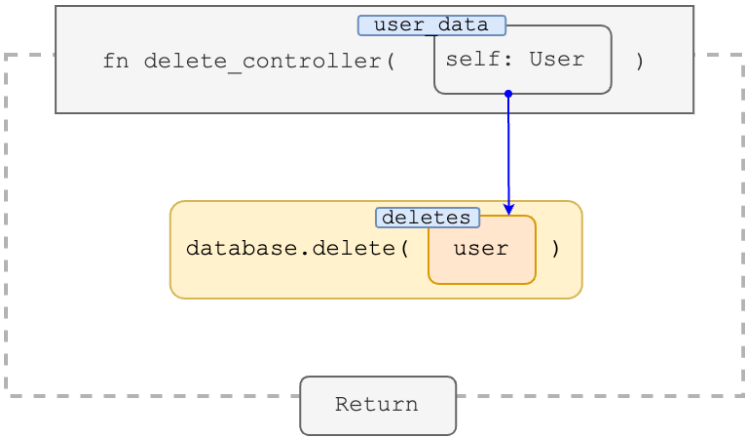


Figure 3: Paralegal's program representation for Listing 1. A light blue box indicates a marker and a blue edge indicates a data dependency.

```
#[paralegal::marker(user_data)]
struct User {
    ...
}

impl Database {
    #[paralegal::marker(deletes, arguments = [1])]
    fn delete(&self, user: User) {
        ...
    }
}

#[paralegal::analyze]
fn delete_controller(user: User) {
    database.delete(user);
}
```

Listing 1: An example marked deletion program.

Then, Paralegal runs its dependency analysis over the source code. In this step, the markers added by developers serve two purposes: 1) they inform the analysis engine about important sections of the code to target, and 2) they inject program semantics into the resulting representation. The analysis outputs the program representation, which captures the dependency relationships between privacy-related data and operations in the source code. Figure 3 shows the program representation: there is a dependency between the inputted `user` and the first argument of the `delete` function call.

Finally, Paralegal feeds the program representation and privacy policy into a policy checker, which assesses whether the program upholds the policy and outputs possible violations. For our example, the policy would pass. We imagine that the entire process of analysis to checking happens as part of the development process as a CI tool or IDE plugin,

so that developers keep privacy top-of-mind, and can identify policy violations and restore compliance as they code.

2.2 Related Work

Paralegal builds on a wide range of prior work in program analysis, graph queries, and tools for privacy and security enforcement.

2.2.1 The Program Dependence Graph

The PDG is a widely used concept in program analysis first introduced by Ferrante et. al in 1987 [6]. It is comprised of data and control dependencies for every operation in a program. Traditionally, it is useful for program optimization – by understanding which operations either depend or are independent of others, compilers can apply out-of-order execution and vectorization to speed up a program's runtime. Paralegal extends the notion of a PDG with markers that denote program semantics to create an SPDG, or Semantic Program Dependence Graph, which it uses to check privacy properties.

2.2.2 Security and Privacy Compliance Tooling

Various systems exist that enforce security and privacy policies. They broadly fall into two categories: dynamic and static enforcement.

Dynamic IFC. Dynamic analysis systems, such as Riverbed [29], Jacqueline [31], and Resin [32], enforce properties at runtime. These tools mainly involve attaching policies to program values and tracking how the data moves as the program executes. These systems are flexible and provide strong guarantees with languages that do not provide much information about the program before runtime, such as Python and PHP. However, these systems often come at the cost of a high runtime and memory overhead and often fail to provide information about implicit flows. Additionally, since they only provide information at runtime, they cannot catch incompliance *during development* and may lead to program crashes and other unwanted behavior for users in production.

Static Analysis. On the other side of the spectrum, many tools rely on static analysis to enforce privacy and security properties. They can provide information during the development process and avoid the high costs of dynamic enforcement.

Some systems, like Jif [17] and STORM [14], do so through the type system. However, these systems are often restrictive, require the usage of modified or niche languages (such as LiquidHaskell), and are difficult for developers to use and understand, e.g., because they come with a high annotation burden. Therefore, they are seldom used in real systems.

Other systems use static analysis, similar to Paralegal, including RuleKeeper [7], PrivGuard [30], PQL (Program Query Language) [16], and Pidgin [13]. RuleKeeper is restricted

to the MERN (MongoDB, Express, React and Node.js) framework, while PrivGuard is limited to data analysis applications and requires each library function to have a hand-written function summary that imposes a high user burden. PQL searches for violating patterns in program code. Matching patterns in *code* is fundamentally lower level than reasoning about concepts in a PDG, so PQL provides less robustness to code changes: simple code changes like renaming a function would require a modification to the policy. Additionally, PQL can only reject violations (e.g., some value cannot be released), whereas Paralegal can additionally enforce positive statements about code (e.g., some value must be encrypted). Pidgin is fairly similar to Paralegal in that it provides users with a policy language that runs queries over a PDG, but it targets Java and uses a less precise analysis. Additionally, the policy language is tied to the names of particular functions in code rather than high-level markers, and the policy language primitives are lower-level than the ones that we defined for Paralegal.

2.2.3 Graph Query Languages and Systems

The proliferation of graph database systems for programs like social media apps has led to several graph query languages and systems in industry, such as Neo4j's Cypher [18] or Oracle's PGQL [21]. There has also been a recent push to standardize these many systems into the GQL Standard [10]. These projects aim to create an intuitive and generic SQL-like syntax to express queries on graph databases and involve pattern-matching and common graph traversal algorithms like Dijkstra's shortest path algorithm. There are also many graph query libraries in Rust, such as PetGraph [20]. These systems are designed to be generic, so they serve a different use case than the graph query system for Paralegal, which is specifically tied to the structure of the SPDG and the use case of privacy policies.

3 Design

3.1 Extending Expressivity

To extend the expressivity of Paralegal, we changed Paralegal's program representation from a data-flow dependency relation to an SPDG. We made two additions: distinguishing control-flow dependencies and adding history-preservation in the data-flow analysis.

3.1.1 Control Flow

Control-flow dependencies, also known as implicit data flows, capture whether a value determines the execution of some operation via control structures like `if` statements and `for` loops. Flowistry's dependency analysis added these to the data-flow relation, since they are a type of data flow. However, they represent a different type of dependency than explicit data-flow dependencies. By differentiating them from explicit data-flow dependencies, we can write policies that specifically require that certain values determine the execution of a function, such as ensuring that an authorization check controls a protected action such as accessing personal data. Or, we can write policies that forbid that certain values determine execution, such as secret values in a cryptography library (which would create a side-channel, vulnerable to timing attacks). Consider the following programs:

```
if (user.has_read_permissions) {
    let data = get_personal_data(user);
    send(data);
}
```

Listing 2: An example program that validly checks user permissions before releasing sensitive data.

```
let data = get_personal_data(user);
send(data);
```

Listing 3: An example program that does not check user permissions before releasing sensitive data.

Without differentiating data- and control-flow dependencies, the two programs would be indistinguishable from one another, as shown in their dependency graphs in Figure 4. However, Listing 3 is obviously a violation of privacy – it sends personal data without checking user permissions!

Clearly, control dependencies must be separated from explicit data dependencies. Flowistry extracts control-flow dependencies from the program's MIR, which is Rust's Mid-level Intermediate Representation [28]. MIR represents code as a CFG (control-flow graph), which is a graph of basic blocks, where edges between the basic blocks provide exactly the control-flow dependencies we would like to represent in Paralegal's program representation. As seen in Figure 6, `_2`, the Rust MIR location corresponding to the permissions check, is in the

`switch` basic block 1. This block branches to determine whether control flows to basic block 2, which fetches the user data, or basic block 5, which flows to the return. So, given a basic block *b*, we can find all MIR locations that have a control-flow influence on *b* by traversing up the edges in MIR from *b* and finding any MIR locations that are passed into a `switch` basic block. We separate this dependency analysis from the explicit data dependency analysis in Paralegal's program representation by labeling edges with whether they represent control or data flow dependencies.

Figure 5 shows the separated data and control dependency graph, with data-flow edges in blue and the control-flow edges in green. The dependency graph for Listing 2 shows that the execution of `get_personal_data` and `send` depends on the `User`. A policy can differentiate this from the separated data and control dependency graph for the code in Listing 3, which has no control-flow edges and therefore still looks like Figure 4.

This allows a policy writer to write the following policy in English that accepts Listing 2 and rejects Listing 3: "if a node marked `user_data` flows to a node marked `send`, a node marked `user` has a control-flow influence on the `send` node".

3.1.2 History-preservation

History-preservation is critical for any policies that consider the definitive order in which data flows through or is transformed in a program. Consider the following programs:

```
let data = get_personal_data(user);
let anonymized_data = anonymize(data.clone());
send(anonymized_data);
```

Listing 4: An example program that validly anonymizes sensitive data before release.

```
let data = get_personal_data(user);
let anonymized_data = anonymize(data.clone());
send(data.append(anonymized_data));
```

Listing 5: An example program that sends both anonymized and non-anonymized data.

With only transitive dependencies, it would be impossible for a policy to differentiate the dependence graphs for the two programs. With some elisions for simplicity, the transitive data dependence graphs are shown in Figure 7. The transitive relation would only show that `send` is influenced by both `user_data` and `anonymize` in both programs. However, this second program is clearly a violation of privacy: a user of Paralegal would like to specify that personal data always must flow *through* anonymization if it is sent.

Without our modifications, Flowistry only finds transitive dependencies. Using the semantics of Rust, Flowistry uses the MIR (shown in Figure 6) to conduct a fixpoint traversal of the graph, which determines the state of the dependency relation at each instruction. For each instruction *i*, Flowistry traverses up all of the ancestors in the CFG, which are all of the instructions that could have been executed before; for each of these ancestors,

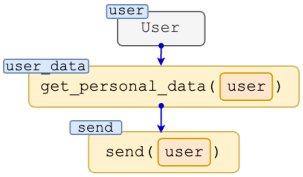


Figure 4: The dependency diagram for Listing 2 and Listing 3.

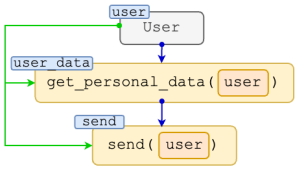


Figure 5: Data and control-differentiated dependency diagram for Listing 2. The green edges indicate the new control-flow dependencies.

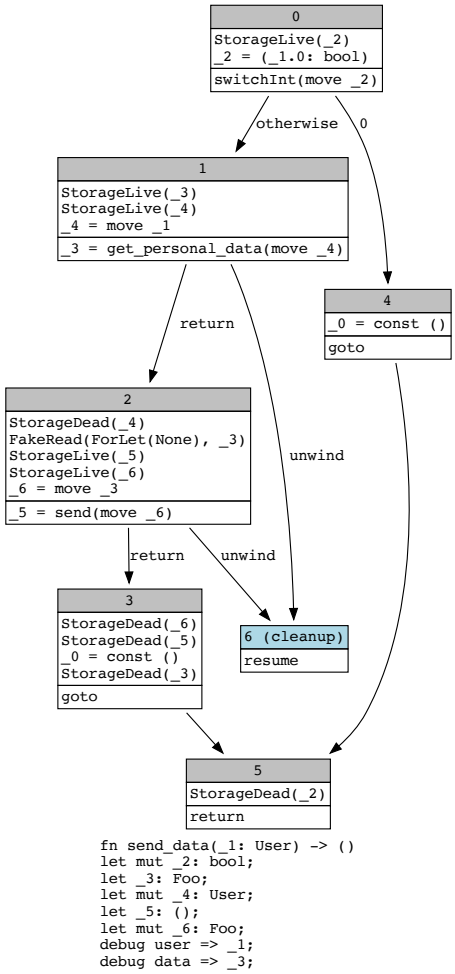


Figure 6: The MIR for Listing 2: `_2`, which comes from the `User`, controls the execution of both `get_personal_data()` and `send`.

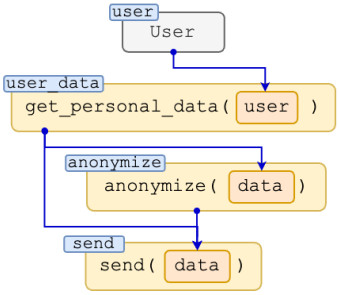


Figure 7: Transitive data dependency diagram for Listing 4 and Listing 5.

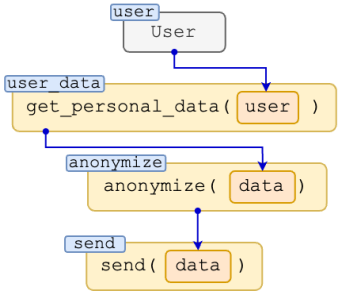


Figure 8: History-preserving data dependency diagram for Listing 4.

Flowistry computes a "transfer function" which checks how the state at i can change given the ancestor's state.

We modified Flowistry so that it outputs non-transitive data dependencies. The fixpoint computation explained above is the same, but the transfer function is changed so that Paralegal does not compute the transitive closure of data dependencies. In the new program representation, the edge between `user_data` and `send` is no longer present for Listing 4 as seen in Figure 8. This allows a policy writer to write the following policy that accepts Listing 4 and rejects Listing 5: "All nodes marked `user_data` must flow through a node marked `anonymize` before flowing to a node marked `send`".

3.2 Speeding Up Paralegal

Previously, Paralegal combined policies in Forge with the SPDG to create a Forge model. Then, the checker transformed this model into a SAT formula and ran a solver on the formula to verify whether the SPDG adhered to the policies. Though flexible, this process was not optimized for Paralegal's workload, and its runtime scaled super-linearly with the size of the SPDG, which is closely correlated with the size of the program. The SPDG of the largest application we evaluated grew to over 1,500 nodes and almost 10,000 edges. Even after implementing optimizations that cut down the size of the SPDG, we found that using Forge created a bottleneck for the end-to-end runtime of Paralegal which was untenable on large applications.

We noticed that across the seven applications and 19 policies we looked at (explained in §5), all were sequences of graph reachability and pattern-matching questions over the SPDG. Thus, we pivoted to defining and implementing these graph operations in the Rust Policy API library. Implementing the policy API as a Rust library was a natural choice because Paralegal is a Rust library that analyzes Rust applications. Rust is also an overall performant language, so we expected that the language itself would not pose performance problems.

3.2.1 The Rust Policy API Types

An SPDG in Paralegal contains a set of controllers (functions annotated as analysis entry-points with `#[paralegal::analyze]`). The dependencies within a controller begin with the controller's inputs, which are sources of data. Data is consumed by the arguments of function calls. Functions have return values, which again are sources of data. Finally, the controller itself has a return value, which is a consumer of data. For simplicity, we unify these sources and consumers of data into the concept of nodes. The SPDG consists of nodes, types, and markers.

The Rust Policy API utilizes two main types: `Node`s and `Path`s. `Node`s are the nodes described above, and `Path`s are a series of edges between nodes, where edges can represent either data dependencies or control dependencies.

Nodes. A node represents a producer or consumer of data in the SPDG and is always tied to a particular controller. There are four types of nodes:

- `ControllerArguments`, which are arguments to an analyzed controller;
- `CallArguments`, which are actual parameters to function calls;
- `CallSites`, which are the data the function call returns; and
- `Return`, which is the return of the analyzed controller.

Consider the following program that stores a form in a database:

```
#[paralegal::marker(user_data)]
struct Form {...}

#[paralegal::marker(check), return]
fn check_write_permissions(&user: User) -> bool {...}

impl Database {
    #[paralegal::marker(store), arguments = [1]]
    fn store(&self, form: Form) {...}
}

#[paralegal::analyze]
fn submit_form(form: Form) {
    if check_write_permissions(&form.user) {
        db.store(form);
    }
}
```

Listing 6: An example program that could be checked using the Rust Policy API.

Figure 9 shows the SPDG for Listing 6. `ControllerArguments` are shown in purple. `CallArguments` are shown in orange. `CallSites` are shown in yellow. The `Return` is shown in red.

Paths. We found that there are three types of paths between nodes that a policy needs to distinguish between:

- `Data`, which are comprised of only data dependency edges;
- `Control`, which are comprised of only control dependency edges; and
- `DataAndControl`, which can contain a mix of data and control dependency edges.

In Figure 9, data dependency edges are shown in blue, and control dependency edges are shown in green.

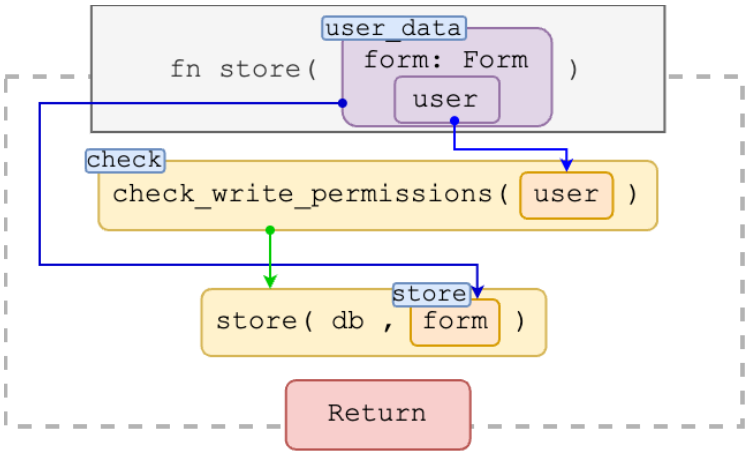


Figure 9: The SPDG for with different `Node` types for Listing 6.

The path between `ControllerArgument:form` and `CallSite:check_write_permissions` is of type `Data` (`Form` data-flows to the `CallArgument`, which data-flows to the `CallSite`). A policy that specifies that the data a user provides should be stored utilizes this path type.

The path between `CallSite:check_write_permissions` and `CallSite:store` is of type `Control`. A policy that requires that the `check` node determines the execution of the `store` node would use this path type.

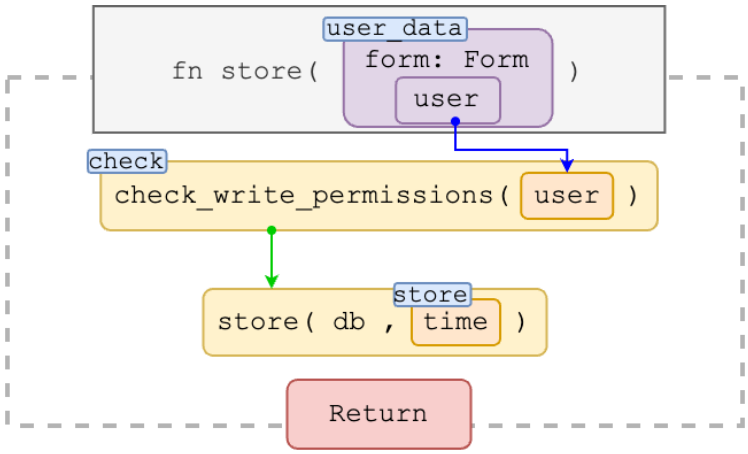


Figure 10: A variation of the SPDG from Figure 9 for with different `PathTypes`.

Both of the above paths would fall under the type `DataAndControl`. To understand why the DataAndControl path is necessary, consider a case where the direct data-flow edge between `ControllerArgument:form` and `CallArgument:store` did not exist (if we stored something other than the form's data, like the current time, shown in Figure 10). The path from `ControllerArgument:form` to `CallArgument:store` would be of type `DataAndControl` because `form` data-flows to `check_write_permissions` and `check_write_permissions` control-flows to `store`. There is no direct flow of `form` to `store`, but there is an *implicit* flow because the storage of time is dependent on the data-flow

of user data to the permissions check. If the application has the policy that user data is extremely sensitive and should never be stored either through implicit or explicit data flows, this `DataAndControl` path would therefore violate the policy in a way that the other two path types don't capture. This path type is mostly useful for policies that forbid paths instead of requiring paths.

Summary. We summarize the API types in Listing 7.

```
struct Node {
    ctrl_id: ControllerId,
    typ: NodeType,
}
enum NodeType {
    ControllerArgument,
    CallArgument,
    CallSite,
    Return,
}
enum PathType {
    Data,
    Control,
    DataAndControl,
}
```

Listing 7: The basic types of the Rust Policy API.

3.2.2 The Policy API

We designed the policy API for a combination of flexibility, concision, and efficiency. We provide a `Context` struct, which contains the interface for defining policies for an analyzed program. The API is comprised of four different types of functions:

- the Core API, which encompasses the most critical functionality needed to express privacy policies;
- the Efficiency API, which are additional functions needed to write policies that run efficiently;
- Type Operations, which reason about data types; and
- Common Operations, which are other operations that we have found to be common in privacy policies in practice.

We simplify some of the function signatures listed and elide some methods in the following sections for concision, but the full API is listed in Appendix A.

The Core API. At its very core, a privacy policy API needs to provide access to the nodes, identify which nodes have what markers, and determine the dependency relationship between nodes. So, the core API consists of four functions:

- `all_nodes_for_ctrl(ctrl_id) -> List<Node>`: Returns all nodes for a controller.
- `has_marker(marker, node) -> bool`: Returns whether the given Node has a marker applied to it directly or via its type.
- `flows_to(src_node, sink_node, path_type) -> bool`: Returns whether a node flows to a node through the given path type.
- `always_happens_before(starting_nodes, checkpoint_fn, terminal_fn) -> bool`: Returns whether starting nodes data-flow through a node satisfying the checkpoint predicate before reaching nodes satisfying the terminal predicate.

Using these functions, a policy writer can write a policy for Listing 6 that enforces that a node marked `check` control influences a node marked `store`:

```
let check_nodes = ctx.all_nodes_for_ctrl(c_id).filter(|node|
    ctx.has_marker("check", node));
let store_nodes = ctx.all_nodes_for_ctrl(c_id).filter(|node|
    ctx.has_marker("store", node));
check_nodes.any(|check_node| store_nodes.any(|store_node|
    ctx.flows_to(check_node, store_node, PathType::Control)
))
```

Listing 8: An example policy using the core API.

The Efficiency API. Indexes store the transitive dependency matrix to achieve a constant-time lookup for the `flows_to` primitive, further explained in §3.2.3. This matrix provides efficient access to entire rows; given this, we provide two additional functions:

- `influencers(sink_node, path_type) -> List<Node>`: Returns the `Nodes` that have a path of the given type to the given node.
- `influencees(src_node, path_type) -> List<Node>`: Returns the `Nodes` that have a path of the given type from the given node.

These primitives allow for increased efficiency. Consider the policy Listing 8: this program is inefficient if there are a large number of `store` nodes. We don't need to find all of the `store` nodes – we only want to assess, out of all of the nodes that a `check` node flows to, whether any of them have the `store` marker. So, using the `influencees` primitive, a policy writer can rewrite the policy like the following:

```
let check_nodes = ctx.all_nodes_for_ctrl(c_id).filter(|node|
    ctx.has_marker("check", node));
check_nodes.any(|check_node|
    ctx.influencees(check_node, PathType::Control).any(|influencee|
        ctx.has_marker("store", influencee)))
```

Listing 9: An example policy that uses the `influencees` primitve.

Listing 9 need not iterate through every `store` node for every `check` node, it simply needs to evaluate nodes that are guaranteed to be influenced by the `check` node. Of course, if the number of `store` nodes is small, the policy in Listing 8 may be more efficient. However, these primitives provide a choice for a policy developer or some future policy compiler to decide which representation is the most efficient given the structure of the SPDG.

Type Operations. Sometimes, users want policies to reason about data types, so it is insufficient to only talk about nodes and markers. We add two type operations to the API:

- `marked_types(marker) -> List<Type>`: Returns all types that are marked with the given marker.
- `nodes_with_type(ctrl_id, type) -> List<Node>`: Returns all nodes that have the given type in a controller.

For example, for the deletion policy discussed in §2.1, a policy writer might like to say that a node of every `sensitive` *type* of data flows to a node marked `deletes`. For example, if a user stores posts, comments, and likes, each represented as a different type, the policy would check that all three types of data are deleted. Using these new type operations, she can write the policy like so:

```
let sensitive_types = ctx.marked_types("sensitive");
sensitive_types.all(|typ|
    ctx.nodes_with_type(c_id, typ)).any(|user_data_node|
        ctx.influencees(user_data_node).any(|influencee|
            ctx.has_marker(influencee, "deletes")
)))
```

Listing 10: An example policy that uses the type operations.

Common Operations. Finally, through the course of writing policies for many production applications, we discovered a few common operations that we added as additional primitives:

- `has_ctrl_influence(influencer_node, target_node) -> bool`: This is similar to a simple `flows_to` using `PathType::Control`. However, it also accounts for the fact that in the PDG, there are often unseen intermediate data-flow nodes in between an influencer and its control-flow target, such as a `deref` call. For example, a

policy writer can replace the `flows_to` call in Listing 8 with a `has_ctrl_influence` call to adjust for such a case.

- `roots(ctrl_id, path_type) -> List<Node>`: Returns the "roots" of a program, or nodes that do not have any influencers of a given edge type. For `PathType::Control`, this will be code that is executed unconditionally, and for `PathType::Data`, this will be anything that "produces" data, like a function of no arguments (which likely fetches data from some other source).

3.2.3 Performance Optimizations

To improve the performance of policy evaluation, we added indexes to the implementation of the policy API and utilized lazy iterators where possible.

Lazy iterators. A policy developer can frequently optimize policies with an early return – for example, the policy from Listing 9 does not need to iterate through all nodes; after the first node it encounters that matches the criteria (marked `check` and flowing to a node marked `store`), it can stop and simply return `true`. Given this, we utilize Rust iterators wherever possible, which compute elements on-demand: only if the current node does not match the criteria does the iterator evaluate the next one. Doing so allows many policies to short-circuit, saving computation time particularly if the operations are expensive and there are many nodes to iterate through. Rust's Iterator API also lets users convert these lazy sequences easily into sets, for efficient repeated membership checks, and vectors, for efficient repeated iteration [12].

Indexes. Though non-transitivity is essential for our SPDG, policies mostly use the `flows_to` primitive, which only needs transitive flows. The control flow relationships in the MIR are already transitive, so we only build an index representing the transitive closure of the data dependency relations. This index is utilized in the `flows_to`, `influencers`, and `influencees` primitives (and by extension, the `has_ctrl_influence` and `roots` primitives which use the aforementioned three primitives).

We implemented a version of the library that similarly created an index for the transitive closure of mixed data and control flows for `PathType::DataAndControl`. However, since this index saw only limited use in the policies we evaluated, and it took a long time to compute (evaluated in §5.5), we decided to utilize a straightforward BFS graph traversal on both data and control dependency edges instead.

4 Implementation

We implement Paralegal's SPDG extraction tool as a Rust compiler plugin. We implement the Rust Policy API as a Rust library. In total, we added 4k lines and removed 1.5k lines from the Paralegal repository for the implementation of the work in this thesis.

4.1 Running Paralegal

Paralegal can be installed as a `cargo` plugin by building and installing the `paralegal-flow` binary. To use Paralegal to analyze a Rust program, a user must link their codebase against the `paralegal` library, which enables the necessary compiler features and provides macros to ergonomically add `#[paralegal::analyze]` annotations and markers to the codebase. The user can run `cargo paralegal-flow` in the codebase, which will run the dependency analysis and output the SPDG as a file. To write a policy, a user must create a new Rust project that imports the `paralegal-policy` library. There are library functions that read in and reconstruct the SPDG from the file, which the user can utilize along with the Rust Policy API functions. They simply run this project to check the policy.

4.2 Rust Policy API Implementation

In the SPDG output of Paralegal's analysis, there is a complex set of nodes and relationships between them, which are at times difficult to reason about. For example, there are three types of `CallSites`, which represent a function call: 1) an argument of a `CallSite`, which receives data and is a variant of the `DataSink` struct, 2) the `CallSite` itself, and 3) the return of the `CallSite`, which can be the source of data and is a variant of the `DataSource` struct. Previously, policies must explicitly relate `DataSink` `CallSites` to their `DataSource` `CallSites`. Policies also needed to explicitly relate all of them to the controller that they live in. Policy writers found these difficult to reason about and often made mistakes, which is why we unified these types into the `Node` type in the Rust Policy API. To remain consistent with the types of the data and control flow relations in the SPDG and to reduce memory usage, the transitive dataflow index uses the types of the SPDG: it has a domain of `DataSource` and range of `CallSiteOrDataSink`. The Rust API helpers efficiently translate from any type of `Node` under the hood to query the index so that policy writers do not need to worry about the actual shape of the `Nodes`. For example, when used as the `src` argument in `flows_to`, a `Node::CallArgument` (which relates directly to a `DataSink`) will be transformed into its related `CallSite` and used to query the index.

5 Evaluation

Our evaluation seeks to answer the following four questions:

1. What new privacy properties do the control-flow and history-preservation attributes of the PDG representation allow Paralegal to check? (§5.2)
2. Are all privacy policies from our case study applications, previously written in Forge, expressible as functionally equivalent policies using the Rust Policy API? (§5.3)
3. Do policies written using the Rust Policy API speed up the policy-checking process compared to equivalent policies in Forge? (§5.4)
4. How do the Rust Policy API's design decisions impact its efficiency? (§5.5)

All experiments were run on a commodity MacBook Air laptop running macOS Ventura 13.2.1 with an Apple M2 chip and 8 GB of RAM. We use Rust version `nightly-2023-04-12` and Forge v1.5.0.

5.1 Applications

This evaluation utilizes an existing evaluation setup for Paralegal, in which we selected production-level applications that handle and store identifying information and encompass a variety of sizes and web frameworks.

Our evaluation involves seven applications, with a focus on the first three applications, for which both compliant and incompliant versions exist: WebSubmit (§5.1.1), Plume (§5.1.2), and AtomicServer (§5.1.3). We scoped out the policies for the remaining applications but we did not fully implement a few of them. However, we have determined their general shape, which is all that is needed for our expressivity evaluation in §5.2. Future work will involve implementing them completely.

5.1.1 WebSubmit

WebSubmit [25] is a homework submission system deployed at Brown and written in 1.6k LoC of Rust using the Rocket.rs web framework [4].

We check three privacy policies for WebSubmit:

1. Data deletion: Tests compliance with a GDPR's "right to be forgotten" by ensuring that for all types of user data that are stored, a controller for deleting all of those types exists.
2. Scoped storage: Tests that all data about a user is associated with that user when stored, which is a precondition for the data deletion policy to function correctly.

3. Authorized disclosure: Checks the access control policy for student answers: students may view only their answers, instructors and TAs may view all student answers, and only instructors may view course feedback.

For each privacy policy, we created three versions of the policy and its related markers to test out different levels of user burden for adding annotations and writing policies:

1. Library: Only reasons about markers to library code (such as the database or Rocket framework). This represents a scenario where library authors maintain generic markers on their code and do not require additional effort on the part of application developers to add or maintain markers.

2. Application: Add markers that are specified and maintained by the application developer. Thus, policies can reason about application-specific semantics, such as the distinction between different types of users (e.g., student, instructor, and TA).

3. Strict: These policies can be arbitrarily tied to a particular implementation to try to catch as many privacy violations as possible. These policies do not reflect how we envision Paralegal would be used in practice, but serve to test the limits of Paralegal's enforcement.

5.1.2 Plume

Plume is a federated blogging service, also written using Rocket.rs [22]. We analyze Plume v0.7.2 (16.6k LoC).

The privacy policy for Plume is the same as the data deletion policy for WebSubmit; it checks that there exists a controller that deletes all types of user data.

5.1.3 AtomicServer

AtomicServer is a graph database server that lets users create, edit, and share graph-structured "atomic" data in web applications [3]. It uses the Actix web framework [1]. We analyze two versions of AtomicServer: v0.28.0, which has 9.6k LoC, and v0.34.2, which has 12.2k LoC.

AtomicServer supports access control to "resources" (data and other system entities). A user can update a resource's properties if they have the correct permissions. Due to a bug, AtomicServer could update a resource before checking write permissions, so users without write access to a resource could apply an update to give themselves write access and pass the permissions check [2]. To address this bug, the policy that we check for AtomicServer is for access control: write permissions must be checked before a resource is updated.

5.1.4 Other Applications

Lemmy. Lemmy is a federated Reddit-like platform [15]. Rather than providing a single centralized website, anyone can create a server tailored to their interests and moderation

preferences. Users create communities within those servers and post content to them. We check access control policies for Lemmy: user permissions must be checked before they can write to a server or read or write to a community.

Freedit. Freedit is another Reddit-like platform [9]. We check two Freedit policies, which together ensure that page views are stored for at most three days. The first policy ensures that the page view is stored with a timestamp. This is a precondition for the second policy, which checks that page views with a timestamp greater than three days old are periodically removed.

Hyperswitch. Hyperswitch serves as a centralized platform between various payment services and vendors [11]. We check three policies: 1) API keys can only be displayed upon creation or after hashing, 2) credit card information must be encrypted before storage, and 3) credit card information can only be stored if the user has consented.

Rust-Crypto. Rust-Crypto is a library for cryptographic operations in Rust [24]. We check that secret keys have no implicit flows, which may leak information about them through timing side-channels.

5.2 Expressivity

To evaluate what new privacy policies the enhancements of the SPDG representation enable, we consider whether each of the 19 privacy policies across the seven applications requires control flow and history-preservation.

Figure 11 shows the results: across the seven applications, only one (Plume) did not use either control flow or history-preservation.

Around a third (6 out of 19) of the privacy policies required history-preservation, almost entirely in the form of an `always_happens_before` combinator. In the case of Hyperswitch, the combinator checks that data undergoes encryption before storage. In the case of WebSubmit, it checks that all recipients are from authorized sources if they receive sensitive data. Only the strict data deletion policy for WebSubmit required the use of history-preservation without the `always_happens_before` combinator to check that all paths between retrieving and deleting user data do not remove any data.

Around half (9 out of 19) of the privacy policies required reasoning about control-flow dependencies. These took the form of checking whether an access-control check authorizes a protected action like storing or retrieving data (Atomic, WebSubmit's Authorized Disclosure property, Lemmy, Freedit, and Hyperswitch), checking that deletion happens unconditionally (WebSubmit's Deletion), and checking that there are no implicit data flows that could create side-channel attacks from secret values (Rust-Crypto).

Application	Policy	History-Preservation	Control flow
Websubmit	Scoped Storage (All)		
	Deletion (Library)		
	Deletion (Application)		
	Deletion (Strict)	✓	✓
	Authorized Disclosure (Library)	✓	
	Authorized Disclosure (Application)	✓	✓
	Authorized Disclosure (Strict)	✓	✓
Plume	Deletion		
Atomic	Access Control		✓
Lemmy	Instance Access Control		✓
	Community Access Control		✓
Freedit	Date Store		
	Expiration Check		✓
Hyperswitch	API Key Display	✓	
	Card Encryption	✓	
	Card Storage		✓
Rust-Crypto	No Implicit Flows from Secret		✓

Figure 11: Almost all applications and policies utilized history-preservation or control flow.

5.3 Functional Equivalence with Forge

To test whether the Rust policies were equivalent in functionality to the Forge policies, i.e., that we did not lose any expressive power by using the Rust Policy API compared to Forge, we evaluated the accuracy of the nine WebSubmit policies (§5.1.1), the AtomicServer policy (§5.1.3), and the Plume policy (§5.1.2) for all application versions.

We successfully implemented all eleven policies using the Rust Policy API. The queries had the exact same semantics, only occasionally differing in procedural implementations; for example, WebSubmit's strict deletion property using the Rust Policy API uses a BFS graph traversal, while the policy in Forge checks the same property by computing and comparing transitive closures.

To test how well these policies can classify compliant vs. incompliant implementations, we then made alterations to WebSubmit's source code which correspond to parts, or "articulation points," of the policy that a developer might get wrong when coding the application. For each articulation point, we made three types of edits: an "Alternative" compliant implementation, a "Bug" implementation where a developer accidentally introduces a bug, and an "Intentional" implementation where a developer tries to intentionally circumvent the policy.

Since the AtomicServer and Plume policies were based on real bugs, we tested their policies on the production pre-bug fix and post-bug fix versions. We created our own buggy implementation of AtomicServer v0.34.2.

For the Rust and Forge policies to be functionally equivalent, we expect them to classify all implementations the same way. Our evaluation found that Paralegal classified all versions equivalently. All 32 incompliant implementations that the Forge policy flagged were also flagged by the Rust policy. All 18 compliant implementations the Forge policy validated were also validated by the Rust policy. All 13 violations the Forge policy did not flag were also not flagged by the Rust policy. All 6 compliant implementations Forge policy mistakenly flagged were also mistakenly flagged by the Rust policy. For these applications, the Rust policies provide equivalence to the Forge policies. In order to conclusively show equivalence, we would need to provide a proof, which we defer to future work.

5.4 Runtime

To evaluate the performance of Rust compared to Forge for different policies on the same application, we measure the runtime of checking all nine WebSubmit policies. To exclude the impact of running the dependency analysis, we measure the runtime of building the indices and running the policy function for Rust, and the runtime of executing the Forge file that contains the policy and SPDG for Forge. We take the mean, min, and max of these runtimes over all edits mentioned in §5.2 for each policy and measure speedup as mean Forge runtime over mean Rust runtime.

Figure 12 shows the results. Overall, the Rust policies outperform the Forge policies by $46-174\times$ for different policies and annotation levels on WebSubmit, with an average improvement of $87\times$. We do not find a correlation between the policy (Deletion, Storage, or Disclosure) and the relative speedup. We do find that the Rust Library policies have a consistently lower speedup compared to the other types of Rust policies. This slowdown comes from computing the dataflow index. Paralegal does not inline the body of a marked function, so for versions of the application with Library annotations, the markers are deeper down in the nested function calls, so the SPDG size is larger. We also find that computing the dataflow index takes 3-5$\times$ longer for the Library annotation level than the Application/Strict annotation levels, though the SPDG is only $1.5\times$ larger. This is likely because the size of the dataflow index passes a threshold to overflow the cache and cause more cache evictions during computation. A larger SPDG size generally also results in a longer running time in Forge, but at the scale of Websubmit, all variations take relatively similar amounts of time.

To evaluate the performance of Rust compared to Forge for running different policies on applications of different sizes, we measure the runtime of checking all policies on WebSubmit, Plume, and two versions of Atomic: v0.28, the version which contained the bug, and v0.34, a recent version that has both more code in general and more code relevant to the privacy policy we enforce. We calculate the mean, min, and max of these runtimes over all edits (both compliant and incompliant) for each application. Figure 13 shows the results. The Rust policies outperform the Forge policies by $24-86\times$, and the speedup does not seem to correlate with Forge runtime or program size. Atomic v0.34, the largest application, took

almost a minute for policy verification using Forge while the Rust policy ran in under a second.

In summary, every policy in Rust was completed in under a second and most took a fraction of a second, a runtime performance that suits an interactive environment such as a CI tool or IDE. Thus, using Rust policies, Paralegal is practical to catch and guide a developer to fix privacy policy compliance during the development process.

	Forge (s)			**Rust (s)**			
	Mean	**Min**	**Max**	**Mean**	**Min**	**Max**	**Speedup**
Deletion (Library)	8.93	8.03	10.04	0.17	0.16	0.22	51×
Storage (Library)	8.79	8.03	9.04	0.19	0.18	0.20	46×
Disclosure (Library)	11.34	9.05	16.05	0.20	0.12	0.35	56×
Deletion (Application)	7.13	7.02	8.03	0.05	0.04	0.09	130×
Storage (Application)	7.03	7.03	7.04	0.06	0.05	0.07	120×
Disclosure (Application)	8.43	8.02	10.04	0.07	0.05	0.09	127×
Deletion (Strict)	9.24	9.03	10.04	0.05	0.05	0.09	174×
Storage (Strict)	7.28	7.02	8.03	0.06	0.05	0.07	129×
Disclosure (Strict)	9.74	9.03	11.05	0.07	0.05	0.09	142×
Overall	8.90	7.02	16.05	0.10	0.04	0.35	86×

Figure 12: The runtime difference between running policies in Forge vs. Rust for the 9 WebSubmit policies.

5.4.1 Runtime Profile

To determine what contributed to the runtime of each system, we profiled running both the Forge policy and Rust policy for the largest application: Atomic v0.34. We found that 94% of Forge's runtime was used for parsing and macro expansion for the SPDG and policy. We encountered similar results for the Rust policy: 93% of Rust policy runtime was used for deserializing the SPDG from the input file. While the relative runtimes are similar, the absolute runtime of Forge parsing is much higher, so if we were to implement a more efficient parser for Forge that has a similar runtime as the one for Rust, the runtime speedup of using the Rust Policy API would be an order of magnitude less than what we observe here. This suggests that a SAT solver-based policy checker could be efficient enough for Paralegal's use cases if given a fast parser for the large graph input relation. However, we have not found the generality of SAT solvers to be necessary for properties of interest.

5.5 Runtime Drilldown

We evaluate the performance impact of the optimizations we implemented for the Rust Policy API by measuring the performance with and without the optimization.

	Forge (s)			Rust (s)			
	Mean	**Min**	**Max**	**Mean**	**Min**	**Max**	**Speedup**
WebSubmit	8.90	7.02	16.05	0.10	0.04	0.35	86×
Atomic (v0.28)	5.58	5.42	5.73	0.24	0.24	0.25	24×
Plume	10.0	7.66	12.34	0.25	0.18	0.32	40×
Atomic (v0.34)	50.26	46.48	54.04	0.88	0.83	0.94	57×

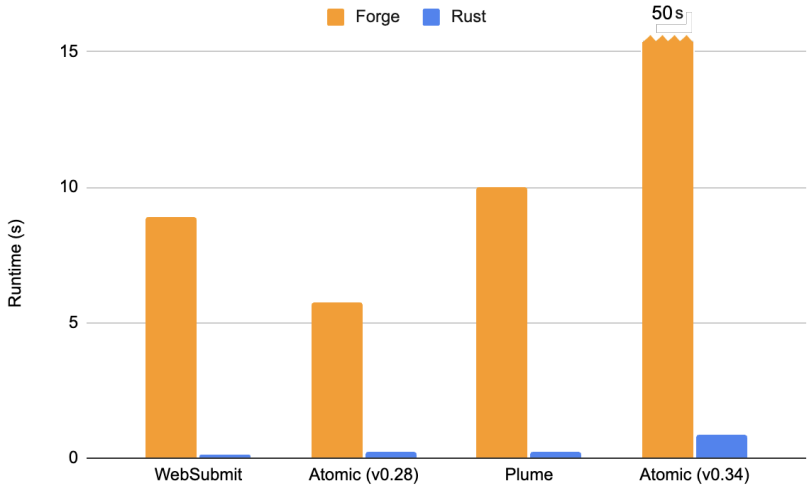


Figure 13: The runtime difference between running policies in Forge vs. Rust for WebSubmit, Plume, and the two versions of Atomic. Atomic v0.34 takes 50 seconds.

5.5.1 Removing the Data and Control Flow Index

The initial version of the Rust Policy API not only computed a transitive index for data flows, but also computed a transitive index for combined data and control flows. We evaluate the performance of running policies using a breadth-first search for `PathType::DataAndControl` compared to using the additional data and control flow index for all three applications.

Figure 14 shows the results, with speedup measured as the runtime of the Rust policy with the index over the runtime of the normal Rust policy, which uses a simple BFS graph traversal over both the data and control flow edges in the SPDG. The speedup is significant and ranges from $1.82 - 11.10\times$.

The speedup is positively correlated with program size. The control flow relation created by Paralegal is transitive, and we propagate these edges across function call boundaries for nested function calls, so large programs with deeper call chain depths will cause the number of control-flow edges to explode. The transitive closure of both the control flow relation and the data-flow graph would therefore grow significantly for large programs, both taking a longer time to calculate and use due to increased cache evictions. For the application with the biggest speedup, v0.34 of AtomicServer, building the two indexes took the vast majority (over 99%) of the total runtime and over five times longer than building only the dataflow index.

Furthermore, reasoning about paths of type `DataAndControl` is not common in policies; across all of the policies for these applications, only WebSubmit's Application and Strict levels of the Authorized Disclosure property required the use of this path type. However, even the policy that does use the index does not use it often enough to make up for the up-front computation. We found that a straightforward graph traversal was sufficient.

	Rust (ms)			Rust w/ Add. Idx (ms)			
	Mean	**Min**	**Max**	**Mean**	**Min**	**Max**	**Speedup**
WebSubmit	103	41	353	188	80	629	1.83×
Atomic (v0.28)	243	240	252	931	904	957	3.84×
Plume	252	180	323	1683	1240	2125	6.69×
Atomic (v0.34)	882	825	939	9788	9327	10250	11.10×

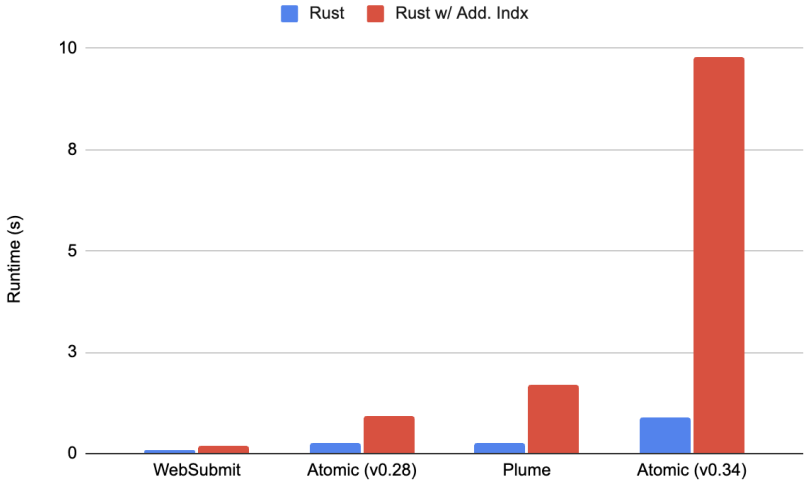


Figure 14: The runtime difference between running policies in Rust and Rust with the additional transitive data and control index.

5.5.2 Lazy Iterators

We evaluate the performance of running policies using the Rust Policy API with and without lazy iterators for all three applications. Figure 15 shows the results, with speedup measured as the runtime of the Rust policy without lazy iterators over the runtime of the normal Rust policy. The performance gain from using lazy iterators is modest; the speedup ranges from nothing (for the Plume policy, which collects all iterators into vectors anyway) to 8% (for the Atomic policy run on Atomic v0.34).

	Rust (ms)			Rust No Lazy (ms)			
	Mean	**Min**	**Max**	**Mean**	**Min**	**Max**	**Speedup**
WebSubmit	103	41	353	109	41	341	1.01
Atomic (v0.28)	243	240	252	247	246	247	1.06
Plume	252	180	323	252	180	323	1
Atomic (v0.34)	882	825	939	952	898	1006	1.08

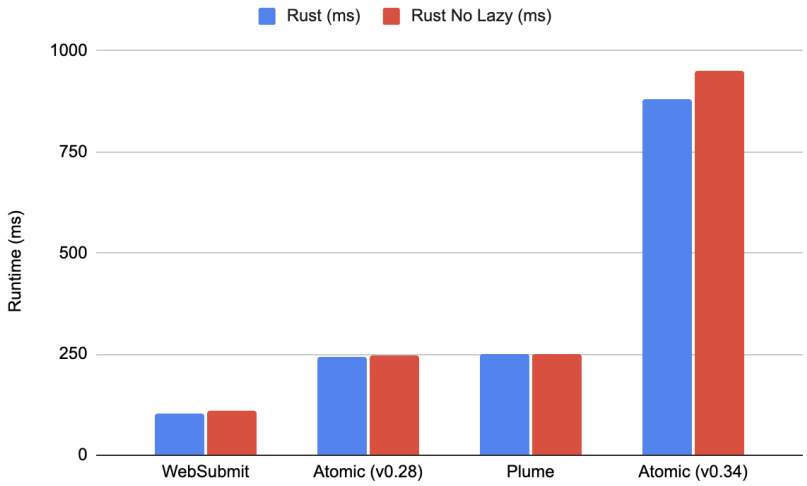


Figure 15: The runtime difference between running policies in Rust and Rust without the use of lazy iterators.

6 Discussion

From our evaluation results, expanding Paralegal's model to include control-flow and history-preservation enabled a large collection of new privacy policies, and utilizing graph traversals for policies with Rust demonstrated large speedups compared to enforcing policies using Forge.

The expressivity improvements enlarge the number of settings in which users can utilize Paralegal to enforce privacy policies. However, since Paralegal is a static analysis tool, there are fundamental limitations in the policies it can enforce. It cannot be used to directly enforce policies that rely on dynamic information knowable only at runtime, such as the type or name of the user currently logged in, which can currently only be approximated. For example, if we wanted to check a policy that only allows users who are designated admins to delete a database, Paralegal can approximate this in a policy by enforcing that an authorization function is run and influences the execution of the deletion, but the policy cannot guarantee that the authorization function *actually* only allows users that are admins to delete the database. Furthermore, Paralegal currently cannot reason about specific values of data in the program, such as whether the result of an authorization check is `true` or `false`.

To do so, we would need to integrate a method like abstract interpretation into the Paralegal prototype.

The efficiency improvements allow policies to be checked quickly and enable Paralegal to be used in interactive settings like an IDE for the applications we tested in our evaluation. However, an assessment of even larger applications (in particular, we attempted to analyze Lemmy [15], an application three times larger than Atomic v0.34) was not completed in this work due to scalability issues with the dependency analysis portion of Paralegal, which is currently under development. Furthermore, additional optimizations can be made to the Rust Policy API itself, such as building the transitive data or the combined data and control flow indexes lazily as policies are being executed.

We could improve the ease of using Paralegal's policies. On the developer's side, one potential improvement is to add built-in error messages into the policy primitives; for example, the Rust Policy API could provide a custom `any` operator that takes in an iterator over `Nodes` could automatically emit an error message based on information about a failing node. We could investigate other heuristics to create error messages. On the policy writer's side, we do not expect policy writers to be too technical, so they may be unable to code the Rust policies. Thus, future work involves creating a controlled natural language policy specification that a non-technical privacy team could create that Paralegal would compile into Rust policy code. This would also allow us to implement policy rewriting to modify policies for efficiency, such as swapping out the `flows_to` and `influencers`/`influencees` primitives based on SPDG attributes as mentioned in §3.2.2.

Finally, a considerable cost of our transition from Forge policies to Rust policies is that Forge's model-checking provided us with a way to generate repair suggestions for the developer to fix their privacy violations that Rust does not [23]. As it was a pressing issue for us to improve Paralegal's performance, we moved to Rust policies. However, repairs are not incompatible with the new policy checker, so a future line of work involves research into how we can re-integrate repair suggestions. The original Forge repairs could suggest that developers remove nodes and edges, which is similar to the built-in error message mechanism mentioned above. They could additionally suggest markers be added to nodes in the SPDG, which we could accomplish using heuristics in Rust or even dispatch to Forge again. We could offer various tiers of Paralegal, with higher tiers providing more features like repair suggestions at the cost of being more expensive in terms runtime by utilizing Forge. Further work could involve research into how to suggest other types of repair the original Forge suggestions could not generate, like creating new call sites.

7 Conclusion

A privacy linter is a new system that allows developers to catch privacy compliance errors during development and can improve an organization's adherence to privacy requirements. We present methods to improve the expressivity and efficiency of our prototype privacy linter, Paralegal, by differentiating control-flow and adding history-preservation to its program model as well as enforcing privacy policies written as graph traversals in Rust. Our evaluation demonstrated that the expressivity enhancements enabled a wide range of new privacy policies. It also indicated that the Rust Policy API is equivalent in functionality to the previous Forge policy verifier for the policies we tried, and that it speeds up policy checking by $23 - 86\times$. Future work is needed to add error messages and improve the ergonomics of using the linter, but these improvements take us one step closer to making privacy linting a reality in production systems.

References

[1] *Actix Web is a powerful, pragmatic, and extremely fast web framework for Rust*. URL: https://actix.rs/.

[2] *AtomicServer Bug Fix: Prevent Unauthorized Commits*. URL: https://github.com/atomicdata-dev/atomic-server/commit/46a503a.

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

[4] Sergio Benitez. *Rocket – Simple, Fast, Type-Safe Web Framework for Rust*. URL: https://rocket.rs/.

[5] 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 2022. New York, NY, USA: Association for Computing Machinery, June 2022, pages 1–14.

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

[7] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. "RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Dec. 2022, pages 1014–1031.

[8] *Forge: A language built for teaching formal methods and modeling*. URL: https://github.com/tnelson/Forge.

[9] *freedit: The safest and lightest forum, powered by rust.* URL: https://github.com/freedit-org/freedit.

[10] *GQL Standard*. URL: https://www.gqlstandards.org/home (visited on 11/26/2023).

[11] *Hyperswitch: The open-source payments switch*. URL: https://github.com/juspay/hyperswitch.

[12] *Iterator in std::iter - Rust*. URL: https://doc.rust-lang.org/std/iter/trait.Iterator.html (visited on 11/26/2023).

[13] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. New York, NY, USA: Association for Computing Machinery, June 2015, pages 291–302.

[14] 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: *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 2021, pages 441–459.

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

[16] Michael Martin, Benjamin Livshits, and Monica S. Lam. "Finding Application Errors and Security Flaws Using PQL: A Program Query Language". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pages 365–383.

[17] 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.

[18] *Neo4j Cypher Manual*. URL: `https://neo4j.com/docs/cypher-manual/current/introduction/` (visited on 11/26/2023).

[19] Naomi Nix, Annabelle Timsit, and Cat Zakrzewski. *E.U. slaps Meta with record $1.3 billion fine for data privacy violations*. URL: `https://www.washingtonpost.com/technology/2023/05/22/meta-fined-eu-facebook-data-privacy/` (visited on 11/26/2023).

[20] *petgraph: Graph data structure library for Rust.* URL: `https://github.com/petgraph/petgraph`.

[21] *PGQL: Property Graph Query Language*. URL: `https://pgql-lang.org/` (visited on 11/27/2023).

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

[23] Sreshtaa Rajesh. "Using SAT Solving and Dependency Analysis to Communicate Privacy Problems in Code". Honors Thesis. Brown University, May 2023.

[24] *Rust-Crypto: A (mostly) pure-Rust implementation of various common cryptographic algorithms*. URL: `https://github.com/DaGenix/rust-crypto/`.

[25] Malte Schwarzkopf. *Websubmit-rs: A Simple Class Submission System*. Oct. 2021. URL: `https://github.com/ms705/websubmit-rs` (visited on 04/06/2022).

[26] 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).

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

[28] *The MIR (Mid-Level IR) - Rust Compiler Development Guide*. URL: https://rustc-dev-guide.rust-lang.org/mir/index.html (visited on 11/26/2023).

[29] 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)*. 2019, pages 615–630.

[30] 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)*. 2022, pages 3753–3770.

[31] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, Dynamic Information Flow for Database-Backed Applications". In: *ACM SIGPLAN Notices* 51.6 (June 2016), pages 631–647.

[32] 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.

A Full Rust Policy API

Below is the full API for the Rust Policy library.

```
impl Context {
    /// Construct a [`Context`] from a [`ProgramDescription`].
    ///
    /// This also precomputes some data structures like an index over
    markers.
    pub fn new(desc: ProgramDescription) -> Self

    /// Find a type, controller or function id by its name.
    pub fn find_by_name(&self, name: impl AsRef<str>) -> Result<DefId>

    /// Find all types, controllers and functions with this name.
    pub fn find_all_by_name(&self, name: impl AsRef<str>) -> Result<&[
    DefId]>

    /// Find a type, controller or function with this path.
    pub fn find_by_path(&self, path: impl AsRef<[Identifier]>) ->
    Result<DefId>

    /// Dispatch and drain all queued diagnostics without aborting the
     program.
    pub fn emit_diagnostics(&self, w: impl Write) -> std::io::Result<
    bool>

    /// Returns whether a node flows to a node through the configured
    edge type.
    pub fn flows_to(&self, src: Node, sink: Node, edge_type: EdgeType)
     -> bool

    /// Returns whether there is direct control flow influence from
    influencer to sink, or there is some node which is data-flow
    influenced by `influencer` and has direct control flow influence
    on `target`.
    pub fn has_ctrl_influence(&self, influencer: Node, target: Node)
    -> bool

    /// Returns iterator over all Nodes that influence the given sink
    Node.
    pub fn influencers<'a>(
        &'a self,
        sink: Node<'a>,
        edge_type: EdgeType,
    ) -> Box<dyn Iterator<Item = Node> + 'a>

    /// Returns iterator over all Nodes that are influenced by the
    given src Node.
    pub fn influencees<'a>(
```

```
    &'a self,
    src: Node<'a>,
    edge_type: EdgeType,
) -> Box<dyn Iterator<Item = Node> + 'a>

/// Returns an iterator over all objects marked with `marker`.
pub fn marked(
    &self,
    marker: Marker,
) -> impl Iterator<Item = &'_ (DefId, MarkerRefinement)> + '_

/// Get the type(s) of a Node.
pub fn get_node_types(&self, node: &Node) -> Option<&HashSet<DefId
>>

/// Returns whether the given Node has the marker applied to it
directly or via its type.
pub fn has_marker(&self, marker: Marker, node: Node) -> bool

/// Returns all DataSources, DataSinks, and CallSites for a
Controller as Nodes.
pub fn all_nodes_for_ctrl(&self, ctrl_id: ControllerId) -> impl
Iterator<Item = Node<'_>>

/// Returns an iterator over the data sources within controller `c
` that have type `t`.
pub fn srcs_with_type(&self, ctrl_id: ControllerId, t: DefId) ->
impl Iterator<Item = Node>

/// Returns an iterator over all nodes that do not have any
influencers of the given edge_type.
pub fn roots(&self, ctrl_id: ControllerId, edge_type: EdgeType) ->
 impl Iterator<Item = Node>

/// Returns the input [`ProgramDescription`].
pub fn desc(&self) -> &ProgramDescription

/// Returns all the [`Annotation::OType`]s for a controller `id`.
pub fn otypes(&self, id: DefId) -> Vec<DefId>

/// Enforce that on every data flow path from the `starting_points
` to `is_terminal` a node satisfying `is_checkpoint` is passed.
pub fn always_happens_before<'a>(
    &self,
    starting_points: impl Iterator<Item = Node<'a>>,
    mut is_checkpoint: impl FnMut(Node) -> bool,
    mut is_terminal: impl FnMut(Node) -> bool,
) -> Result<AlwaysHappensBefore>
```

```
    /// Return all types that are marked with `marker`
    pub fn marked_type<'a>(&'a self, marker: Marker) -> impl Iterator<
    Item = DefId> + 'a

    /// Return an example pair for a flow from an source from `from`
    to a sink
    /// in `to` if any exist.
    pub fn any_flows<'a>(
        &'a self,
        from: &[Node<'a>],
        to: &[Node<'a>],
        edge_type: EdgeType,
    ) -> Option<(Node, Node)>

    /// Iterate over all defined controllers
    pub fn all_controllers(&self) -> impl Iterator<Item = (
    ControllerId, &Ctrl)>

    /// Returns a DisplayDef for the given def_id
    pub fn describe_def(&self, def_id: DefId) -> DisplayDef
    /// Returns a DisplayNode for the given Node
    pub fn describe_node<'a>(&'a self, node: Node<'a>) -> DisplayNode
    <'a>
}

impl AlwaysHappensBefore {
    pub fn report(&self, ctx: Arc<dyn HasDiagnosticsBase>)
    pub fn holds(&self) -> bool
    pub fn assert_holds(&self) -> Result<()>
    pub fn is_vacuous(&self) -> bool
}
```