Using SAT Solving and Dependency Analysis to Communicate Privacy Problems in Code

Sreshtaa Rajesh

Advisor: Malte Schwarzkopf
Reader: Shriram Krishnamurthi

Department of Computer Science
Brown University
Providence, RI
May 2023
Contents

Acknowledgements 4

Abstract 5

1 Introduction 6
  1.1 Key Idea: Privacy Linting 6
  1.2 Challenges 6
  1.3 Contributions 7

2 Background 7
  2.1 Related Work 7
    2.1.1 Developer Burdens of IFC 7
    2.1.2 User-Friendly Error Messages 8
    2.1.3 Solver-Generated Repairs 8
  2.2 Mapping Program Dependencies with Flowistry 8
  2.3 Property Specification and Model Checking with Forge 9

3 A Motivating Example 9

4 Design 10
  4.1 Privacy Linter Overview 10
  4.2 Finding Error Locations in Source Code 11
    4.2.1 Delta Approach 11
  4.3 Generating Relevant Context 12
  4.4 Repair Suggestions 12
    4.4.1 Subtractive Repair 12
    4.4.2 Additive Repair 12
  4.5 Unsatisfiable Cases 13
    4.5.1 Dual Unsatisfiable Cases 14

5 Implementation 15

6 Evaluation 16
  6.1 Paralegal Evaluation 16
  6.2 Repair Evaluation 17
  6.3 Sample Error Message Walkthrough 18

7 Discussion 20
  7.1 Scalability 20
  7.2 Callsite Generation 21
Acknowledgements

I would like to first and foremost thank Malte Schwarzkopf for being a wonderful research advisor, mentor, professor, and friend. Malte’s support has truly made all the difference to me in my time doing research at Brown, and I can truly say that I would not be studying and researching systems today if not for him.

I’d like to also thank Shriram Krishnamurthi for being such a strong advocate for me all these years. Shriram’s teaching was my first exposition to CS in college, and it was a formative experience that set the course for the rest of my undergraduate experience. I truly have learned so much from him.

Next, I’d like to thank my wonderful team of student researchers on this project. Justus, Livia, and Carolyn are some of the loveliest people around, and it’s truly been a pleasure to work with such bright minds.

Additionally, thank you to Iris Bahar for bringing me on board in Summer 2021 for my first research project. I find that first-time experiences can sometimes be make-or-break, and working with such an inspiring advisor drove me to seek out additional research opportunities.

Finally, thank you to my family, my friends, the ETOS group, and my ExploreCSR cohort for your constant support. It means the world to me.

This research is done in collaboration with Justus Adam, Livia Zhu, Carolyn Zech, Malte Schwarzkopf, Will Crichton, and Shriram Krishnamurthi. The thesis focuses on my contributions, but discusses work done by others.
Abstract

Developers are prone to writing buggy code that compromises the functionality of the software and the privacy of those who use it. We present the idea of a privacy linter—a tool that can detect potential privacy bugs introduced during development similar to the way that code linters detect instances of bad style. A central part of our design for a privacy linter is minimizing user burden; the tool is only effective if developers find it worthwhile to use. Our broader privacy linter research addresses the burden of integrating the tool with an existing codebase (as well as other key challenges surrounding privacy linting), while the work presented in this thesis seeks to reduce the burden of interacting with the linter’s feedback.

When the linter detects a potential privacy bug, it produces an error message. This error message must be clear—the developer should understand exactly what problem the linter detected. It should also be actionable—the feedback should ideally lead developers to a potential fix. We address two forms of actionability in this work: locating the detected problem and providing suggestions for repairing it. Unlike traditional code linters, which use AST-level analysis to output actionable feedback, diagnosing the root cause and location of a potential privacy problem requires a semantic understanding of code. To this end, a key insight of this work is the use of a SAT solver to explore various modifications to an abstracted model of source code that cause violated privacy policies to pass. We show that these modifications (which we refer to as deltas) contain information regarding the location of the detected problem as well as potential repairs.

Finally, we realize these ideas in our prototype privacy linter, Paralegal, and evaluate our error messages on a homework submission application deployed at Brown.
1 Introduction

Laws like the GDPR have solidified privacy and the protection of customer data as a top priority for companies, and they impose heavy fines for failing to adhere to regulations. Yet, writing privacy-compliant code continues to be difficult for even well-meaning developers, because humans are prone to writing buggy code. This has motivated a large repertoire of research [4, 6, 10, 12, 13] that focuses on tools and frameworks to help developers write safe code, and information flow control (IFC) has emerged as an idea that can ensure that data only flows within permitted boundaries. Static IFC—which is done at compile-time—has powerful guarantees, but has faced an uphill battle to widespread adoption due to the fact that it imposes strict guidelines on types and/or program structure or otherwise requires heavy code modifications that most developers do not want to implement in practice [6, 13]. Therefore, we identify a need for a class of tools that prioritize usability, ease of adoption, and low integration burden, but can still provide useful insights about privacy compliance.

1.1 Key Idea: Privacy Linting

We envision a tool called a privacy linter, which developers periodically run on a code base during development to identify potential privacy problems that may have been introduced before the code is deployed. We believe a good privacy linter should satisfy the following critical qualities:

1. Necessitate few changes to source code, but still provide developers with useful feedback regarding the compliance of their system.
2. Support a flexible and expressive range of privacy policies.
3. Ideally, give developers guidance on where in code potential problems originate and how to fix them.

The work done in this thesis focuses on the third item: the generation of human-readable error messages for the privacy linter. However, a key insight of this work is that our abstraction choices to satisfy the earlier items directly enable us to easily generate high-quality error messages that are extensible to additional functionality, such as suggesting repairs to code.

1.2 Challenges

Realizing effective error messages for a privacy linter requires solving a number of research problems. First, unlike traditional linters, reasoning about privacy requires semantic understanding of code and cannot be gleaned from purely syntax-level pattern matching. Therefore, we must find a way to map high-level semantic properties to low-level syntax in order to communicate potential privacy problems to users.

Broadly, a good error message should have the following qualities:
1. Correctness: The error message should correctly state the privacy policy that was violated.

2. Helpfulness: It should point developers to the lines in the source code involved with the possible bug.

3. Actionability: Ideally, it should also give developers suggestions for how to fix detected bugs.

1.3 Contributions

This thesis presents four contributions in the broader space of privacy linting. First, we present techniques that exploit a solver to explore modifications (deltas) to an abstracted representation of a program. Second, we show that these deltas can be used to localize detected problems in code as well as perform a set of basic repairs. Third, we use dependency analysis to augment error messages with relevant context that may have influenced the possible violation. Finally, we evaluate the linter (and specifically the quality of error messages) on a homework submission application currently deployed at Brown.

2 Background

The following section discusses background information pertinent to the work done in this thesis. § 2.1 discusses prior research pertinent to the topic of error message generation and privacy linting. § 2.2 and § 2.3 discuss tools that our prototype privacy linter, Paralegal, uses.

2.1 Related Work

2.1.1 Developer Burdens of IFC

Information flow control (IFC) has historically been a powerful tool to reason about certain properties relevant to privacy, such as access control policies. As a result, a large body of research exists in the IFC space, including frameworks to help developers write safer code.

IFC can guarantee properties—such as non-interference—about the flow of data within an application. Static IFC typically comes with a high integration burden that either imposes a strict program structure, restricts types that programmers can use, or defines a new language for developers to learn. STORM [4], one such IFC framework, requires developers to learn and write applications and policies in a specialized language known as LiquidHaskell. The policies are enforced by LiquidHaskell’s type system, and as a result, provide sound compile-time guarantees about the policies [4]. From an ergonomics perspective, however, these errors report as a type unification error, which do not provide developers with helpful information regarding the source of the failure or how to fix it. Dynamic IFC [12, 10] involves modifications to the runtime of a language, which in turn lowers the developer integration burden. However, since these frameworks perform their checks at runtime, many application
bugs may only be caught after deployment when users are already interacting with the system. Additionally, current dynamic IFC systems traditionally respond to prohibited flows by either denying requests, substituting default values in for sensitive values, or blocking the flow of sensitive values altogether. While these strategies do indeed mitigate the impacts of a data leak, they do not actually inform developers about the underlying cause of the problem, and as a result, bugs may go unfixed for long periods of time.

2.1.2 User-Friendly Error Messages

The research done in this thesis focuses on generating high-quality error messages for a privacy linter. Prior work examining novice programmers’ interactions with the Racket language’s error messages presents two key findings [5]. First, a good error message should refrain from providing an explicit solution to detected issues, as any single solution may not be universally applicable to all possible causes of such an error. A novice presented with an explicit solution might unquestioningly implement it even if the solution does not apply to their particular problem. Conversely, it is imperative that an error message does not lead a user to an incorrect fix [5]. Pyret, a functional language meant for computer science education, builds on these ideas to implement effective error messages with carefully chosen vocabulary and visual aids (such as the use of color and highlighting) [11]. Our work uses this research as a starting point for basic qualities of good error messages, though with a couple of key differences. The primary use of a privacy linter is to provide feedback about the privacy compliance of code, meaning that there is greater utility in providing suggestions for fixes. Conversely, the linter’s feedback is best-effort and does not promise correctness in terms of its diagnosis or repair.

2.1.3 Solver-Generated Repairs

A key part of the linter’s error messages involve solver-generated repair suggestions to help developers fix potential issues. The concept of solver-generated repairs exists in prior work. One system, Alchemy [3], discusses an approach at systematically translating Alloy specifications for persistent databases into update operations that preserve desired invariants about the database at each stage. This work provides insights on applying repairs to a database in the case that an update operation breaks an invariant. Namely, a good repair should restore the invariant without breaking another invariant or undoing the update operation entirely, implying that the set of valid repairs depends on the type of update [3]. This idea inspired our investigation of linter repairs, which also differ in effectiveness based on the kind of error detected in source code.

2.2 Mapping Program Dependencies with Flowistry

Paralegal abstracts source code into a graph of program dependencies that capture relationships between various program entities while omitting low-level source code details that are
not relevant to privacy. To do this, we use Flowistry [2], a dependency analysis crate that uses Rust’s borrow/mutability semantics to obtain a precise graph of program dependencies. More specifically, Flowistry can determine for any given variable in a Rust program, what other variables could have influenced its value. This type of taint-tracking analysis has historically been infeasible for practical use because it tends to overapproximate dependencies (i.e., in any scenario where an argument is passed into a function, consider its value modified by the function). However, Rust’s restrictive memory model and explicit mutability annotations make it possible to drastically reduce the overtaint, since we know exactly when values can possibly be modified (i.e., if they are marked as mutable or borrowed via a mutable reference).

2.3 Property Specification and Model Checking with Forge

Paralegal specifies its privacy policies in first-order logic and uses a model checker to formally verify whether a system complies with those policies. To do this, the linter uses Forge [7]: a specification language with a model checker built atop a SAT solver. Users can model systems and define predicates written in first-order logic that check certain properties about their model. From here, Forge provides two routes of action: 1) the model checker can verify that a concrete instance (provided by the developer) satisfies the predicates, or 2) the solver can come up with its own instance that satisfies a set of constraints, typically captured in the predicates.

3 A Motivating Example

We consider Websubmit [9], a homework submission system implemented in Rust using the Rocket framework [1]. This application handles user data, namely students’ answers to homework questions. Websubmit follows GDPR regulations, so it must obey the “right to be forgotten”: a user must be able to delete all their data from the system at any point. This policy can be specified as follows:

There exists one HTTP endpoint, accessible to each site user, which invokes a deletion method on every item of every type of personal data stored by the application about that user.

Suppose now that in their rush to deploy this application, a developer forgets to implement deletion logic for student answers in their deletion endpoint. Now, Websubmit is incompliant with the GDPR. This is a classic example of a bug that we would like a privacy linter to point out before deployment. Listing 1 shows an ideal conceptualized error message that the linter could display.
In the following sections, we discuss the pipeline from analyzing compliance to generating these user-facing error messages.

4 Design

The description of the privacy linter in § 4.1 is prior work that this thesis builds upon. The following subsections discuss the design of the linter’s error messages, which is the specific contribution of this thesis.

4.1 Privacy Linter Overview

To check compliance, the linter must first produce an abstract representation of source code. There are a few reasons for this. First, we would like the linter to be scalable to large, industry-sized codebases. However, verification times become prohibitive even for relatively small applications if the linter considers every low-level implementation detail. Second, privacy engineers who write the properties should not be required to have knowledge of day-to-day changes in the codebase, given that many of them will not impact code paths that are relevant to privacy. This means that a good abstract representation of source code will only reflect meaningful code modifications in the system model.

To this end, we introduce our abstract representation—the system model. The system model is a graph of program elements, where edges are relationships between elements that may be significant to privacy. Edges include: data-flow dependencies between callsites that have the ability to influence the value of another callsite or call-argument; control-flow dependencies that indicate which program elements control whether other program elements get executed; and markers on program elements that introduce semantics for privacy policies to reason about.

The former two types of edges (data and control-flow dependencies) comprise the program dependency graph (PDG). The PDG is built with a custom version of Flowistry [2], which uses Rust’s borrow/mutability semantics to obtain—for a given program entity—a low-overtaint and high-precision map of which lines of code could have influenced its value and vice versa. The latter type of edge (markers) are taken from lightweight annotations supplied by developers that “mark” parts of source code (such as types or function headers) with semantics for privacy properties to reason about. For example, the formalized policy

Listing 1: Ideal error message for a privacy linter.

Error: Found violation of policy "GDPR Right to be Forgotten".
Type Answer is never deleted.
You can fix this by adding a call to:
    delete_answer
In the function:
    forget_user
for the "right-to-be-forgotten" in § 3 would realistically be specified in terms of markers on program entities, i.e. that 1) there exists a sink that attaches a deletes marker to any entity that flows into it, and 2) the sink applies a deletes marker to every type marked sensitive.

The system model is represented as a partial instance in Forge, where the PDG enriched with marker annotations are relations over program elements such as types, callsites, and call-arguments. We imagine that a team of privacy experts will have already formalized privacy policies using first-order logic. From here, we run Forge’s model checker on the system model and verify its compliance with the formalized privacy policies.

4.2 Finding Error Locations in Source Code

When traditional linters find instances of bad style, they highlight the problematic lines of code. Similarly, a good error message for a privacy linter should point developers locations in source code that are involved with the bug.

If the model checker finds a counterexample to a given predicate, it means that the source code as represented by the system model is incompliant with the policy. In this case, we must communicate the problem back to the developer in the form of an error message. However, the counterexample presented by Forge is the entire graph. This is unhelpful; it can be equated to a spell-checking system highlighting an entire paper due to one typo. Thus, to achieve a helpful error message, we need to isolate the location(s) of the privacy violation in the code.

4.2.1 Delta Approach

Our approach takes advantage of the fact that Forge’s solver is good at exploring parameter spaces to find solutions that fit a set of logical constraints. This means that we can exploit the solver to explore modifications to the system model’s dataflow graph that make it compliant. In other words, for a given privacy policy \( P \), set of original dataflows \( F \), and solver-generated delta of edges \( \Delta E \), Paralegal queries the solver for a satisfiable solution to the formula:

\[
\exists \Delta E . \ (\text{not } P[F]) \text{ and } P[F - \Delta E]
\]

By virtue of how \( \Delta E \) is generated, we are guaranteed that it will contain edges involved in the privacy problem, but we are not guaranteed that it is minimal. In practice, however, we found that the deltas returned from this query are normally small; we attribute this to the underlying SAT solver’s strategy in coming up with a satisfiable instance to the query. However, we note here that if there arises a scenario in which \( \Delta E \) is large, there exists prior research on a minimization algorithm [8].

Once we have our delta, we have achieved two goals:

1. **Diagnosis:** The linter can reverse-engineer the lines of source code that correspond to the offending edges and display them as the location of the privacy violation.
2. **Repair**: If the developer removes the edges contained in the delta, they will have a compliant system. This is further discussed in § 4.4.1.

### 4.3 Generating Relevant Context

As mentioned previously, the edges in the delta correspond to locations in source code that are involved with a potential privacy bug. Once it has these source code locations, Paralegal feeds them once again to Flowistry, which performs dependency analysis to pick out other relevant parts of the source code that could have influenced these locations to display as additional context for developers. This type of analysis is advantageous given that interacting code is not necessarily colocated.

### 4.4 Repair Suggestions

Ideally, a privacy linter should also be able to give developers suggestions for how to fix detected bugs.

#### 4.4.1 Subtractive Repair

Notice that by using the delta approach detailed in § 4.2.1 to find the location of the error, we also have a simple repair: if the developer removes the lines of code that are contained in the delta, the predicate is guaranteed to pass. This type of repair is effective in scenarios where a flow that should not exist is present, such as sensitive data flowing into a more permissive sink.

On the other hand, simply removing any line related to a privacy problem will often change the semantics of the program in an unintended way. Let us suppose that we run the linter on the buggy version of Websubmit discussed in the § 3. Stepping into the solver’s shoes, we must find some set of edges to remove from our model to make it compliant. An obvious choice for this is the line of code that stores the answers in the database in the first place. By removing this line, we have solved our problem; we no longer have data about students that lingers in the application after an account is deleted. However, we have also reversed our application’s functionality to not being able to store answers at all.

The root of this issue is that not all privacy problems can be solved purely by removing code. In many cases, privacy leaks are caused by a missing flow, such as in our example, forgetting to invoke a deletion method. The linter’s repair capabilities would be impractical, therefore, if it could only suggest removals; it must also be able to suggest additions.

#### 4.4.2 Additive Repair

The challenge with suggesting additions is that Forge cannot manufacture a function call as easily as it can remove one. The system model is represented as a partial instance that locks down all the callsites and call-arguments available in the program, meaning that Forge
can explore modifications that remove some of these nodes and edges from the graph, but it cannot create additional call sites or call-arguments that would correspond to adding a missing function call. The implications of this with respect to repair is that we cannot ask Forge to directly provide a delta of edges to add to the PDG. However, we make an observation here that the primary mechanism by which formally-specified policies reason about source code semantics is through markers. The "right-to-be-forgotten" property in our example from § 3 does not fail because it detected a missing function call. Rather, it fails because the predicate expected some function argument with type \( \text{Answer} \) to have a \textit{deletes} marker, which it did not. If a linter were to point out the missing marker to a developer, they could then look at the set of places in the program annotated with a \textit{deletes} marker and deduce which function call is missing.

This is exactly the process that we employ in order to implement additive repair. We introduce the idea of a \textit{marker assignment}, which builds on the discussion of markers in § 4.1. A \textit{marker assignment} is a relation between PDG nodes and markers; a \((n, m)\) tuple in the relation would indicate that the PDG node \(n\) has the marker \(m\) attached to it. Paralegal queries the solver for a delta of missing marker assignments on nodes of the PDG that would make the violated predicate pass. Formally, given predicate \(P\), original set of marker assignments \(M\), and a solver-generated delta of marker assignments \(\Delta M\), Paralegal queries for a change to the set of marker assignments satisfying:

\[
\exists \Delta M. \left( \text{not } P[M] \right) \text{ and } P[M + \Delta M]
\]

If there exists some \(\Delta M\) that satisfies this query, Paralegal displays a repair suggestion that lists the the missing markers as well as the line(s) of code corresponding to the PDG node that each marker is assigned to. Paralegal also extracts all program entities (either functions or type definitions) that the developer annotated with the missing markers and displays them as part of the repair suggestion to help developers determine how to satisfy the missing marker assignments.

### 4.5 Unsatisfiable Cases

There are instances in which one or more of the repair queries are unsatisfiable. For example, consider a sanitation policy for a private browser application that requires that a \texttt{wipe()} function is called at the end of a user’s session to wipe all of their browsing data. The predicate written for this policy would state this in terms of markers: if we assume that the developer annotates the \texttt{wipe()} function with a \textit{sanitizes} marker, then the policy would state that there must exist a callsite with a \textit{sanitizes} marker in the PDG. Suppose now that a developer forgets the \texttt{wipe()} function call. Now, our system model is incompliant, but there exist no edges that we can remove from the PDG to make it compliant. This means that querying for a removal delta will be unsatisfiable, and our privacy linter should be able to handle this scenario.
We make the observation that even unsatisfiable queries can still provide helpful information. In cases where the removal delta is unsatisfiable (like our \texttt{wipe()} example), the linter determines that the problem is not due to an absence of code, which implies that \textit{none} of the current source code is the root cause of the privacy violation. This is useful to developers, as it narrows down the reason for the violation and the type of fix necessary. From here, the linter will still display the additive repair, provided that it is satisfiable. In our example above, the additive repair will contain the \texttt{sanitizes} marker in the missing marker assignment, and the linter will display the \texttt{wipe()} function as an entity annotated with that marker.

Conversely, if the additive repair is unsatisfiable, then the developer knows that the privacy property cannot be satisfied by adding markers on program entities. In this case, the linter will still display the subtractive repair if it is satisfiable. Finally, if both deltas are unsatisfiable, the linter will default to solely printing the property that was violated. We believe these “dual-unsatisfiable” cases are rare, based on the rationale presented in the following section (§ 4.5.1).

### 4.5.1 Dual Unsatisfiable Cases

Property writers write properties over an enriched PDG, which contains dataflow edges as well as marker assignments. This means that if a property fails, then the source of the error can be traced back to an error in the dataflow or an error in the marker assignments. This gives us four potential sources of errors: an erroneous dataflow that should not exist; a missing dataflow that must exist; a missing marker on some node of the PDG; or an erroneous marker that should not be present on some node of the PDG. In order to reduce the risk of dual-unsatisfiable cases, the repair queries should broadly cover all four types of failures.

**Catching errors due to dataflow edges.** Dataflows in the PDG represent \textit{concrete} callsites and call-arguments. As discussed earlier, subtractive repair can catch errors related to erroneous dataflows. However, properties cannot directly detect \textit{missing} dataflow edges, because doing so would require property writers to reason about particular lines in the underlying implementation, which breaks the linter’s abstraction boundaries. Instead, to verify that a particular dataflow exists, the property must check that some node of the PDG flows into a sink labelled with a particular marker. If such a flow is missing from the enriched PDG, then the property fails. We note here that the direct cause of the verification failure is due to a missing \textit{marker assignment}, not a missing \textit{flow}. Since additive repair detects missing marker assignments, we have a way to detect the class of errors caused by missing flows.

**Catching errors due to markers.** We have discussed how additive repair covers errors caused by missing marker assignments. The last class of errors involves detecting \textit{erroneous} marker assignments. An erroneous marker assignment could mean one of two things: 1) the developer incorrectly marked a program entity, and it caused a predicate to fail or 2) data is flowing into a marked sink when it should not. The former is a developer error that falls outside the linter’s threat model; the solver will faithfully trust the markers placed on
program entities and reason about them accordingly. The latter is exactly the erroneous
dataflow problem covered by subtractive repair.

We have shown that two repairs proposed in § 4.4 cover the four most common reasons
for property failures. Therefore, if a dual-unsatisfiable case arises, it can likely be attributed
to a developer incorrectly marking their source code or the property referring to concrete
callsites and function arguments without referring to markers (which Paralegal discourages).

5 Implementation

We implemented error message generation in approximately 250 lines of Rust and 30 lines
of Forge. We now describe the pipeline of error message generation.

A user runs a first pass of Paralegal over their codebase, which conducts dependency
analysis, extracts the PDG and markers to form the system model, and outputs it as a partial
instance in Forge. We then use the model checker to verify that the system model complies
with formalized privacy policies. If a predicate fails verification, Paralegal queries the solver
once for a removal delta, which contains the location of the error and serves as one form of
repair, and another time for an additive delta, which serves as a second form of repair. The
deltas are then serialized into JSON.

Implementing the delta approach required making certain modifications to the policy
code. Predicates and the helpers used to write them could originally take the fixed relations
from the partial instance for granted and directly compute over them. However, the delta
approach requires the solver to explore modifications to data flows and marker relations in
the partial instance. Therefore, we had to modify predicates and their helpers to compute
over abstract relations passed in as arguments. This way, if we want to prove some property
about the original system model, we can pass in the original flows and labels relations
into predicates. However, if we would like the system model to explore modifications (as is
the case with repair), we can extend or shrink these sets with the deltas.

Following the generation of removal and additive deltas, the user runs a second pass
of Paralegal with a flag that takes in the path to the JSON of the deltas. During this pass,
Paralegal deserializes the JSON into a Rust struct and extracts a source code location from
each element in both deltas. Then, it performs dependency analysis via Flowistry on each
location from the removal delta (containing the edges responsible for the violation). This
provides a set of additional source code locations that may have influenced the violation, to
be printed as relevant context. After this, it looks up every marker in the additive delta and
gathers a list of functions in the program that can apply the marker. From here, all that is left
to do is print the error message.

The error message is printed in the following manner. We first print the property that
was violated, followed by the MIR that corresponds to the locations in the removal delta.
After this, we print relevant context by converting the set of additional source code locations
(gathered from dependency analysis) to MIR. Finally, we print the suite of repair suggestions.
For removing lines of code, we print the lines of MIR that correspond to the removal delta. For adding markers, we first state the missing marker, then state the line it was missing from, and finally list all functions in the program that can apply that particular marker. A sample error message is shown in § 6.3.

6 Evaluation

We ran Paralegal, our prototype privacy linter, on Websubmit, a homework submission system written in approximately 1,600 lines of Rust using the Rocket web framework.

We evaluated Paralegal on its ability to detect violations of three different privacy properties:

1. **Data Deletion**: Detailed in §3;

2. **Scoped Storage**: A precondition for deletion that requires that a user’s personal data be stored alongside a unique identifier for that user; and

3. **Authorized Disclosure**: Encodes Websubmit’s access control policy: students can view their own answers, and when they lead class discussions, they can also view their peers’ answers. Instructors can see all student answers.

For each privacy violation that Paralegal successfully identified, we then evaluated the quality of its error messages and repair suggestions. In the following sections, we first provide context on the broader evaluation of Paralegal and then discuss the error message evaluation.

6.1 Paralegal Evaluation

For each property, we made a series of edits to the source code that fell into one of three categories: *Alternate*, *Bug*, or *Malicious*. *Alternate* represents an alternative syntax for the same functionality. This could be replacing an iterative function such as `map` with a `for` loop that achieves the same thing. Ultimately, the linter should not report any errors for these edits since they do not change the semantics of the program. On the other hand, the *Bug* class of edits correspond to mistakes made by a well-meaning developer. This contrasts from the *Malicious* class of edits which correspond to someone intentionally trying to circumvent the linter. We would ideally like the linter to catch both kinds of errors.

We repeated this experiment across three levels of property writing and annotation effort. The lowest level (*Library*) involved generic properties written around a series of markers on library functions. The middle level (*Application*) included markers on application functions and types, so that properties could reason about basic application semantics, such as user roles. Finally, the highest level (*Strict*) tailored the privacy properties closely to the specific implementation.

Shown in Table 1 is the linter’s accuracy across the categories of edits that caused the system to be incompliant with one of the properties. The linter evaluation was done across
Table 1: Evaluation of Paralegal on Websubmit for the Application-level property writing and annotation burden.

<table>
<thead>
<tr>
<th>Property</th>
<th>Edit Type</th>
<th>Errors Found</th>
<th>Total Errors Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Deletion</td>
<td>Bug</td>
<td>2/3</td>
<td>3/6</td>
</tr>
<tr>
<td></td>
<td>Malicious</td>
<td>1/3</td>
<td></td>
</tr>
<tr>
<td>Scoped Storage</td>
<td>Bug</td>
<td>0/1</td>
<td>1/2</td>
</tr>
<tr>
<td></td>
<td>Malicious</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>Authorized Disclosure</td>
<td>Bug</td>
<td>3/3</td>
<td>3/3</td>
</tr>
<tr>
<td></td>
<td>Malicious</td>
<td>3/3</td>
<td></td>
</tr>
</tbody>
</table>

all three effort levels, but for the purpose of this thesis, we evaluated only the Application (mid-tier) effort level. As seen, Paralegal finds a majority of errors but fails to report errors for certain edits in data deletion and scoped storage.

6.2 Repair Evaluation

For every edit that corresponded to a true positive (the linter correctly flagged an error), we queried the SAT solver for the location of the error message and subtractive repair (via a removal delta), and an additive repair (via an additive delta). To assess whether Paralegal correctly located the error in source code, we manually inspected the source code that corresponded to the removal delta and checked whether it contained lines that were disallowed. Lines could be disallowed either because they themselves were prohibited flows (such as unauthorized disclosure), or because they were incompliant in the absence of other flows (such as storing sensitive data without a way to delete it).

Moving on to repairs, our evaluation scored Paralegal’s repair suggestions on two metrics:

1. **Correctness**: The suggested repair must be satisfied in order to lead to compliant source code with correct semantics.

2. **Completeness**: A developer needs to follow only the suggested repair—without any additional steps—to reach compliant code with correct semantics.

Since different problems require different fixes, we evaluated both repair suggestions as a single unit where we scored the one with better correctness. If both were correct, then we scored the repair with better completeness. This draws a parallel to a developer being shown a suite of potential fixes and them choosing the best one to go forward with.

To perform our assessment, we first came up with a manual repair for each true positive that roughly corresponded to how an all-knowing developer would fix the privacy problem. From here, we assessed correctness by whether a subset of the edges in the removal delta were indeed absent from the correct code, or whether a subset of the markers in the additive delta were indeed present on their respective entities in the correct code. We note here that
Table 2: Error message evaluation on two metrics: ability to detect error locations and quality of repairs (measured in terms of correctness and completeness)

<table>
<thead>
<tr>
<th>Correct Locations</th>
<th>Correctness</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/4</td>
<td>4/4</td>
<td>2/4</td>
</tr>
</tbody>
</table>

we allowed the correct repair to be a subset of the deltas because of limitations in Forge that make it difficult to guarantee a minimal delta.

To assess completeness, we enumerated the steps of making the code compliant (corresponding to the manual fix) for each edit, and then compared those steps to the steps suggested by the better repair. If the solver’s suggestion accounted for all necessary steps, then it was complete.

Looking at the results in Table 2, we see that 4/4 repair suggestions were correct, and 2/4 of the correct suggestions were also complete. The complete repair suggestions correspond to bugs that solely required deleting a line of code. In contrast, the incomplete suggestions correspond to scenarios in which a function call needed to be replaced. In terms of repair, this means that the developer would need to remove a callsite and then add a different callsite to reach semantically correct code. In summary, Paralegal’s repair mechanisms are best suited for one-step repairs that either involve only additions or only subtractions. In compound cases, Paralegal can help get a developer started, but ultimately the developer must also use their knowledge of the codebase to complete the fix.

6.3 Sample Error Message Walkthrough

Figure 1 shows a real error message produced by Paralegal on a buggy version of Websubmit that violates the GDPR "right-to-be-forgotten" in the manner described in § 3. The structure of the error message is as follows: the first section (red) highlights what property was violated as well as the location of the error; the second section (yellow) displays other lines of code that may influence the violation; and the final section (green) presents a suite of repair suggestions.

We walk through this error message in the shoes of a developer. First, we see the lines involved in the violation. Several of these are Rust utility functions, such as into and clone that the developer must sort through, because the linter currently does not ignore these functions. However, the final line involved with the violation is the following:

```rust
_68 = backend::MySqlBackend::replace(move _69, move _72, move _74) -> [return: bb40, unwind: bb105]
```

The backend::MySqlBackend::replace() method stores data in the database; this call is currently invalid because we are not permitted by the GDPR to store sensitive data without an endpoint to delete it. Therefore, from our own knowledge of the system, we can verify that Paralegal correctly identified the location of the error.
Moving to the repair suggestions, we see that the subtractive repair suggestion mirrors the example subtractive repair suggestion from § 4.4.1. Though it leads to code that complies with our policy, deleting the `backend::MySqlBackend::replace()` line would remove our ability to store student answers. Deeming that this repair is not appropriate for the problem at hand, the developer moves to the next repair. The additive repair listed after suggests that the `deletes` marker was expected on an element of the `Form<LectureQuestionSubmission>` type. As shown in Listing 2, we see that this type contains a map of student answers. This type is passed in as an argument (named `data`) to the controller responsible for collecting student answers. The signature for this controller is shown in Listing 3.

**Listing 2:** Type definition for LectureQuestionSubmission

```rust
pub (crate) struct LectureQuestionSubmission {
    answers: HashMap<u64, String>,
}
```

**Listing 3:** Signature of incompliant controller

```rust
pub fn questions_submit_internal(
    apikey: ApiKey,
    num: u8,
    data: Form<LectureQuestionSubmission>,
    backend: &State<Arc<Mutex<MySqlBackend>>>,
    config: &State<Config>,
) {
}
```
The repair suggestion is thus stating that this argument (containing student answers) needs to flow into a sink that applies the `deletes` marker inside the controller. With no additional information, it is reasonable that a developer with knowledge of the application could eventually infer that the missing marker belongs to the answers contained inside data that never reach a deletion endpoint.

The ease of coming to the correct solution is further assisted by the second component of the additive repair: the printed list of program entities that can apply the `deletes` marker. These are refined by what type of program entity they are (i.e. function, type, etc.). In this case, we see that every possible place that a `deletes` marker could originate is a function, giving the developer a strong hint that the fix to their bug involves adding a function call. Finally, since the concrete function names are printed, we reduce the broad range of possible function calls to a narrow set that includes the correct `delete_answer` function.

Through this walkthrough, we show that 1) the information contained in the error message is sufficient to point out a real privacy problem in an application and 2) it is plausible for the developer to reach a fix for the problem given the repair suggestions.

### 7 Discussion

This work, though preliminary, has a couple of open questions that may inform further directions of research.

#### 7.1 Scalability

On larger programs, the solver may take prohibitively long to either find an instance for the delta queries or determine that the queries are unsatisfiable. Given that all of our current queries only have one degree of freedom for the solver to explore (either removing edges or adding markers), the existence of scalability problems restricts our ability to extend repair queries to ones with multiple degrees of freedom, such as callsite replacement (involving a callsite deletion and a subsequent insertion), or having the solver explore modifications to dataflow and control-flow edges. Additionally, scalability issues are a deterrent to usability. Thus, an important future line of work is to find ways to optimize the solver’s search for a satisfiable instance.

One consideration for achieving this is that currently, the system model gets cluttered by Rust utility functions, such as `deref` and `unwrap`, which are called often and nearly always attached to some other function call that either takes in a reference (in the former case) or returns a `Result` type that must be extracted. This increases the size of the PDG, which in turn increases the solver’s computation time.
Figure 2: Inserting a callsite into an existing flow can change program dependencies. The left side shows the original code and PDG, and the right side shows the modified code and PDG. Broken red edges are removed from the PDG after the callsite insertion and solid green edges are added.

7.2 Callsite Generation

We can infer missing callsites through a related symptom—missing markers. However, the developer still must determine which function call is missing (although Paralegal can narrow this down) and furthermore, they must determine where to add the missing function call.

Our current process to generate repairs is not easily extensible to directly generating new callsites. For one, there are a limited number of edges one could remove from a PDG, but there are an unlimited number of edges one could add. Additionally, when we add a callsite to source code, it can change the program dependencies, as is illustrated in Figure 2. The left side of the figure shows the PDG for a call to `foo()` that flows into a call to `bar()`. If we insert a call to `baz()` between our original two calls (as shown on the right side of the figure), then the call to `bar()` no longer depends on the call to `foo()` because the inserted callsite may have overwritten the value of `x`. This means that the new PDG removes the original edge and adds two edges corresponding to the new flow.

From this example, we show that callsite generation involves more complex edits to the dataflow graph than the delta queries: at a basic level, it requires performing more than one operation on the PDG (i.e. edge additions and subtractions). Accurately reflecting the updated dependencies is also a challenging research problem in itself, because it requires information regarding the pointer semantics of the underlying source code that is not immediately available in the abstracted system model. For example, if the call to `baz()` in Figure 2 instead took an immutable reference of its argument, then `x` would not have been overwritten, so `foo()` would still have an edge to `bar()`.

These requirements for an effective callsite generation repair suggest that our current use of the solver is not well-suited to the problem. Instead, we hypothesize that a good solution to this open question would require more advanced forms of program synthesis.
8 Conclusion

Manual privacy audits are necessary today in ensuring that software systems are secure and compliant with company and state policies, but relying solely on the human eye to catch privacy violations has proven to be insufficient. Tools like the privacy linter can potentially fill this niche if developers can be convinced to use them.

A central tenet of designing a user-friendly linter is effective communication about problems detected in code via error-messages; we detail our goals for this communication in § 1.2. To meet these criteria, in § 4 we present a process for generating user-friendly error messages for a privacy linter that extends the linter’s existing abstraction choices. Namely, we exploit the solver to explore modifications to the enriched PDG representation of source code, and we show that these modifications can locate errors and suggest repairs. Finally, we evaluate our error messages on a deployed application in § 6 and show that they provide useful and actionable feedback to developers. These promising results show that a practical privacy linter designed with developer ergonomics in mind is realizable, and we plan to push further in future work to expand our prototype linter into a tool that can easily be integrated with real-world applications.
References


