Paralegal: Practical Static Analysis for Privacy Bugs

Abstract

Privacy bugs are important, but finding them in today's software requires onerous manual audits. Code analysis tools can help in theory, but they must be practical and ergonomic to succeed, and no tools today fit the bill.

Paralegal is a new system that statically analyzes privacy properties in Rust programs. Key to Paralegal's practicality is its distribution of work between the program analyzer, privacy engineers, and application developers. Privacy engineers express a privacy policy using a vocabulary of domain-specific *markers* that application developers then apply to source code entities. Paralegal extracts a Program Dependence Graph (PDG) from Rust code, leveraging Rust's unique type system to automatically model the behavior of black-box code. Paralegal augments the PDG with the developers' markers, and checks privacy policies against the marked PDG.

In evaluation with eight real-world Rust applications, Paralegal alerts developers to real privacy bugs, including two previously unknown ones. Paralegal expresses a wider range of policies than IFC and checks code bases more reliably than state of the art code analysis tooling. Finally, Paralegal runs in seconds on large, real-world code bases, and its markers are easy to maintain as code evolves.

1 Introduction

Applications that handle sensitive user data must comply with privacy policies and legal frameworks like the GDPR [32], access control, and data retention limitations. Even within a single organization, the number of developers modifying a shared codebase on a daily basis makes it difficult to correctly implement and adhere to these requirements [33, 76]. Today, organizations rely on manual audits by privacy experts or external consultants to check if their code respects privacy properties. Manual audits are laborious, error-prone, and unlikely to happen frequently [62, 65].

Paralegal is a program analysis tool that helps developers find possible privacy problems in their code before deployment (Figure 1). With Paralegal, privacy engineers articulate high-level privacy policies as queries on flows between abstract code elements, such as "all types marked as user data must flow into deletion functions" or "access to sensitive data must always be influenced by an authorization check." [Malte: This very heavily invokes IFC to me; the old version of the intro did a more extensive data deletion example, which I found helpful.] Application developers provide meaning to abstractions like "user data" by annotating concrete code elements with *markers*. Paralegal then statically analyzes the relationship between marked elements in the extracted from the code program dependence graph (PDG), and reports policy violations to developers. We expect developers to use Paralegal to check their code for privacy bugs e.g., as part of an integrated development environment (IDE) or a continuous integration (CI) toolchain.

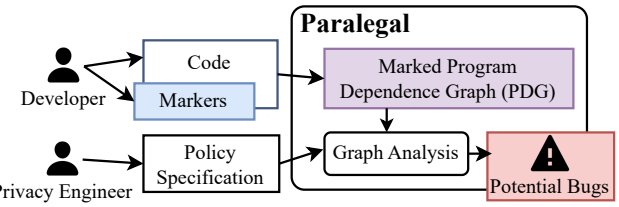

Figure 1: *Paralegal checks application code annotated with markers against a privacy policy by extracting a marked Program Dependence Graph (PDG) that captures program dependencies. It reports potential privacy bugs to developers.*

Paralegal builds on a rich literature on static bug finding and policy enforcement tools (§2). A key challenge for a *practical* analysis tool is to handle realistic codebases written in popular languages, which make extensive use of external libraries and change frequently. Prior systems handle library dependencies—i.e., how does data flow through code unobservable by the analyzer—either by providing a model for a highly specific domain, or require onerous manual specification of library code behavior. In addition, these systems generally require policies directly embed brittle references to source code elements (e.g., identifiers, expressions, or named functions), which increases policy maintenance work for frequently changing code. We believe that these drawbacks have

prevented their practical adoption.

Paralegal takes a new approach by separating concerns between the program analyzer, privacy engineers, and application developers. This separation is possible thanks to two key insights. First, Paralegal leverages Rust's ownership type system to infer a reasonable approximation of the behavior of library code, and integrates approximate behavior of external code with the PDG. This automates the previously-manage handling of library code, but requires careful design to preserve accuracy. Second, Paralegal decouples policies from the source code via markers defined by privacy engineers. Application developers then markers to source code elements and maintain the association without editing the policy. Paralegal's analyzer propagates markers into the PDG and leverages them to improve accuracy and scalability.

We evaluate our Paralegal prototype by applying it to eight real-world Rust web applications. Paralegal finds both known and unknown privacy issues in them: it would have caught five known bugs, and it finds two previously unknown bugs. We compare our approach with IFC and CodeQL, a practical code analysis tool supported by GitHub. We find that Paralegal can express a broader range of policies than IFC, that Paralegal finds bugs more reliably than CodeQL, and that Paralegal's markers reduce the complexity of policies compared to CodeQL. We also investigate Paralegal's maintenance effort in evolving applications by applying it to 1,000+ commits spanning 2.5 years of development of Atomic [4], and find that marker changes are rare and no policy changes were needed. Finally, we find that Paralegal's optimizations to reduce marked PDG size allow it to run in seconds, making it suitable for frequent use.

Paralegal is open-source [2] and out prototype is currently being evaluated for use at a large internet company. This company already has extensive static analysis tooling, but sees value in Paralegal as a complement to it. Specifically, teams at the company are exploring applications of Paralegal to ensure secrecy of cryptographic keys, to enforce encryption-at-rest, and to check that mitigations for speculative execution attacks are executed in hypervisor code.

In summary, this paper makes four key contributions:

1. The Paralegal static analyzer, which checks high-level properties against low-level, evolving code bases.
2. A flexible policy framework that combines a high-level property language with developer-applied markers to reason about application code in terms of PDGs.
3. The marked PDG abstraction and techniques to efficiently generate precise marked PDGs from Rust code.
4. Case studies reporting on our experience of applying Paralegal to eight real-world Rust web applications.

Paralegal has some limitations. As a static analyzer, it can only reason about information known at compile time and must abstract over all possible executions. Paralegal's policy flexibility also means that there is no universally sound approximation during PDG construction. As such, Paralegal's soundness and completeness are policy-dependent.

2 Motivation and Background

A practical privacy bug finder must be ergonomic for developers and deal with the realities of real-world codebases, including widespread use of libraries and frequent code change.

Specialized tools achieve practicality by targeting a single domain or type of policy. These systems operate on widely-used programming languages (e.g., Java, Python, Javascript, or SQL) and bake an understanding of the domain—such as Android apps in DroidSafe [38], tabular data analytics in PrivGuard [70], or ORM-based MERN (MongoDB, Express.js, React.js, NodeJS) apps in RuleKeeper [35]—into the system. Importantly, this domain modeling helps these systems understand the semantics of API functions, libraries, and frameworks without having to analyze their code. However, these tools are limited to checking domain-specific properties.

At the other extreme are general **security-typed programming languages**. This category includes languages designed for information flow control [42, 57, 61] and security-typed ORMs [44], as well as proof assistants that encode security policies into dependent types [10]. These languages generally require ecosystem buy-in by means of extensive annotations. Policy-flexible one's require a purely-functional programming style and often manually-authored proofs, and none are widely used in practice.

Prior **general policy checkers** that let users encode policies as queries over a model of code written in a mainstream language fail to handle libraries and have poor developer ergonomics. Research systems that extract an AST-based and/or flow-based program representation and provide a query language over it exist e.g., for C [72], Java [41], PHP [9], Node.js [52], and Ethereum contracts [66]. All of these either ignore library code or require developers to provide and maintain manually-written models to convey its behavior. For example, consider CodeQL [8], a "semantic code analysis engine" maintained by GitHub that has backends for languages including C++, Java, and Python. CodeQL's developers maintain extensive, manually-written models of C++ libraries such as `std` and `boost` [15, 16], and users must equally model other libraries they use. These systems also rarely integrate with the codebase being analyzed, but instead encourage policy writers to query syntactic code constructs. For example, CodeQL policies often use regular expressions over identifiers to select source code elements. While helpful for use cases where the goal is to identify syntactic patterns (e.g., retry loops [64]), this design makes CodeQL brittle for enforcing custom semantic properties over changing code.

Paralegal targets Rust, a mainstream programming language whose ownership type system provides Paralegal with the unique ability to approximate the effects of external library code. Paralegal requires no manual flow models or domain-specific semantic hints to enforce policies for programs with data flow into and out of library code. [Justus: A bit too strong.

```
impl Database {
  #[paralegal::marker(make_delete_query, arguments = [id])]
  fn prepare_delete(&mut self, id: u32, table: &str) {...}
}

impl User {
  #[paralegal::analyze]
  fn delete_user(&self, db: &mut Database) {
    let my_data: UserData = self.get_my_data();
    db.prepare_delete(self.id, "users");
    for blog in &my_data.posts {
      db.prepare_delete(post.id, "posts");
    }
    for comment in &my_data.comments {
      db.prepare_delete(comment.id, "comments");
    }
    db.execute();
  }
}
```

Figure 2: *Paralegal alerts developers to missing code (red) to delete a user's comments when deleting their account in Plume [60]. (Code simplified and error handling omitted.)* *[Carolyn: Recall Rahel's feedback: we need more marker examples. This listing should have a single user datatype (e.g. Blog) with the user_data marker attached.]*

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

(a) *The initial Plume user deletion policy. Bullets are in blue, [SK: Not clear what a "bullet" means; concept does not show up until page 7.] variables are in purple, and markers are in teal.*

```
Somewhere:
1. For each "user data" type marked user_data:
  A. There is a "source" that produces "user data" where:
    a. There is a "deleter" marked make_delete_query where:
      i) "source" goes to "deleter"
      and
      ii) There is a "execute" marked executes where:
        A) "deleter" goes to "execute"
```

(b) *The revised Plume user deletion policy.*

Figure 3: *Example specification for user data deletion written in Paralegal's policy language (§4.3).*

Even in our use cases we needed one model, but it required very few.] In addition, Paralegal decouples policy specification and source code using lightweight *markers* that developers attach to program elements and maintain with their code. This layer of abstraction makes Paralegal ergonomic for developers as it separates policies from the frequently-changing application code.

Next, we explain how Paralegal works; §8 will contrast Paralegal with bug finding techniques other than static analysis.

3 Paralegal Overview

Paralegal statically catches privacy violations by extracting a model of Rust application code and checking it against a privacy policy written by privacy engineers. We illustrate Paralegal's workflow using an example based on a real-life bug in Plume, a federated blogging application [60].

Plume lets users create posts and comments. If a user deletes their account, the application must delete their posts, comments, and the user metadata. Figure 2 shows the user deletion code. `delete_user` on `User` takes a database as an argument. It constructs deletion database queries for the user themselves and each of the user's posts and comments, then executes those queries. However, this function contains a privacy bug: the code in red is missing, so user comments are not deleted [58]. Given a suitable privacy policy, Paralegal catches this bug.

First, a privacy engineer determines a policy and writes it down using Paralegal's policy language (Figure 3a). They write the policy in terms of *markers*. Here, the marker `user_data` describes the concept of personal data, and `deletes` describes the concept of deleting data. Markers allow privacy engineers to talk about the program at a level of abstraction above the implementation. The privacy engineer produces the policy in Figure 3a and sends it to a developer.

The developer leverages their application knowledge to apply the policy's markers to relevant program entities. They apply the `user_data` marker to the types `Post`, `Comment`, and `User`. When they go to apply the `deletes` marker, they discover that there is no correct place to put it. If they apply it to the `prepare_delete` function, the policy would pass if a deletion query is simply *constructed*; nothing ensures that the application actually *executes* the query. They cannot put it on `execute` either: `execute` is a generic function that handles all types of queries, so applying the `deletes` marker here would allow *any* executed query to satisfy the policy.

Since the privacy engineer is unfamiliar with implementation details, they wrote a policy that does not quite fit with the application logic. The developer is faced with a choice: they can either refactor the application to work with the policy as written, or they can work with the privacy engineer to revise the policy. They go back to the privacy engineer, explain the problem, and together they revise the markers and write a new policy—for all types of user data, there must exist a query to delete them, *and* that query must be executed (Figure 3b). The developer applies the new markers to the application. This give-and-take between privacy engineers and developers is a common Paralegal workflow.

The developer then regularly runs Paralegal to check the policy as they work on the application (e.g., in CI or as an IDE plugin). When faced with the erroneous `delete_user` func-

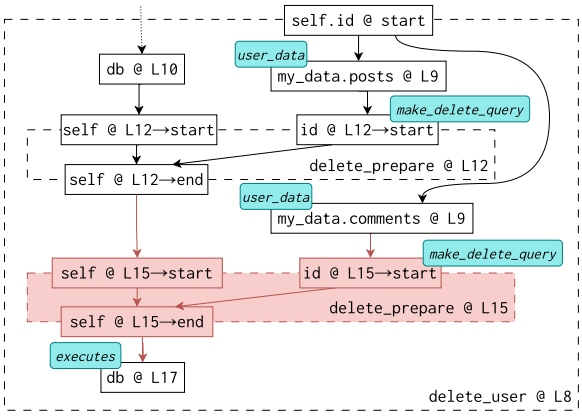

Figure 4: *Partial and simplified PDG for the Figure 2 example. Solid rectangles are PDG nodes, dashed rectangles are function scopes, and teal bubbles are markers. The red subgraph represents the missing comment-deletion code. "Lk" = line k in Figure 2. "start" and "end" refer to the function entry and exit locations, respectively.*

```
1  'Deletion Policy' not satisfied.
2  No entrypoints satisfied rule                1.A.a.i
3  Entrypoint `delete_user'
4    Did not satisfy                            (Rule 1.A.a.i)
5      "source" goes to "deleter"
6    for this "source":                         (Rule 1.A)
7      9 | let my_data = self.get_my_data();
8        |     ^^^^^^^
9    for the "user data" type: "Comments"       (Rule 1)
10   Help: There is a "deleter" here:
11    10 | db.prepare_delete(self.id, "users");
12       |                   ^^^^^^^
```

Figure 5: *Paralegal's error for the Plume deletion bug.*

tion, Paralegal reports the error shown in Figure 5. Internally, Paralegal produced the marked PDG shown in Figure 4 and detected that there is no path from a node marked `Comment` to a node marked `make_delete_query` (and from `make_delete_query` to `executes`). The developer fixes the bug by adding the red code in Figure 2, and Paralegal's check passes again.

4 Design

Paralegal is comprised of three components: PDG construction (§4.1) as an abstraction of the program, markers (§4.2) as a semantic vocabulary, and policies (§4.3) to enforce properties articulated with markers on the program abstraction.

4.1 Program Dependence Graph

The PDG [34] is a generic representation for a program that can be reliably extracted for applications in any domain. For example, Figure 4 shows a simplified PDG for the Plume application in Figure 2. Paralegal constructs this PDG from Rust's MIR, a control-flow graph (CFG) intermediate representation in the Rust compiler.

Constant c Variable x Field i Function f Binop $\oplus$

$$
\begin{aligned}
\text{Prog } P &::= G^{+} \\
\text{CFG } G &::= I^{*} \\
\text{Instr } I &::= p = rv \mid \texttt{goto}\ n \\
&\quad \mid \texttt{if}\ p\ \texttt{then}\ n_1\ \texttt{else}\ n_2 \\
&\quad \mid \texttt{return} \\
\text{Place } p &::= x \mid p.i \mid *p \\
\text{Rval } rv &::= c \mid p \mid p_1 \oplus p_2 \\
&\quad \mid \&p \mid f(p^{*})
\end{aligned}
\qquad
\begin{aligned}
\text{PDG } H &::= (N, E) \\
\text{Node } n &::= p\,@\,cs \\
\text{Edge } e &::= ek\,@\,cs \\
\text{EKind } ek &::= \text{data} \mid \text{ctrl} \\
\text{Loc } \ell &::= f.k \\
\text{CallStr } cs &::= \ell_0 \to \ldots \to \ell_k
\end{aligned}
$$

Figure 6: *Grammar for a core subset of the Rust CFG (top and left) and for the Paralegal PDG (right).*

To check policies on realistic codebases with reasonable precision and recall (as we evaluate in §7.1), the PDG must be sensitive to dependencies at a fine granularity. Specifically, the PDG needs to have three key properties:

Flow-sensitivity allows the PDG to distinguish between the values of data at different program locations. For instance, `db` at Line 10 is not the same as `db` on Line 17, so the PDG has separate nodes for each. By contrast, a flow-*insensitive* PDG would represent `db` with a single node, only reflecting the value of `db` at the end of the program, once all of the queries have been prepared and executed. This single-node representation would allow a program that calls `execute` *before* preparing the queries to pass the policy, even though such a program would not actually delete any data.

Context-sensitivity allows the PDG to distinguish between different calling contexts to the same function. Without context sensitivity, the PDG would represent the body of the `prepare_delete` function only once. This would mean that, for instance, its `self` argument is only one node. Each call to `prepare_delete` (Lines 10, 12 and 15) then connects the inputs to that one `self` node. It would therefore appear as though `self.id`, `post.id`, and `comment.id` are *all* arguments to the first `prepare_delete` call (Line 10), even though only `self.id` is. As a consequence, the policy could not detect a bug where a developer moves the `execute` call to Line 11.

Field-sensitivity allows the PDG to distinguish between different fields of a structure. For example, a field-sensitive PDG distinguishes `posts` and `comments`, even though they belong to a single variable, `my_data`. A field-*insensitive* PDG would not be able to detect the original bug (shown in red), because `my_data` (and therefore `comments`) flows to `delete`.

4.1.1 Definitions

Formally, the Rust CFG follows a superset of the grammar in Figure 6-left. In this simplified model, instructions are

either assignments, unconditional jumps, conditional jumps, or returns. The left-hand side of an assignment is a *place*, or an expression that refers to a specific region of memory (a variable, a field of a place, or a dereference of a place). The right-hand side can be constants, places, operations on places (including address-of), or function calls. The actual Rust CFG contains more details such as array indexing, but this subset is sufficient to illustrate Paralegal's behavior.

The Paralegal PDG follows the exact grammar in Figure 6-right. A node in the PDG is a place p at a call string cs, and an edge is an operation of kind ek at a call string. A call string represents the sequence of locations $\ell_0 \dots \ell_k$ that uniquely identify a given instruction in a call tree.

The fundamental goal of the Paralegal PDG is to encode *dependence* within a Rust program. Dependence is a hyperproperty [13] of a program: informally, in a program P, a variable y depends on a variable x if there exists two executions σ_1, σ_2 of P such that $\sigma_1(x) \neq \sigma_2(x) \implies \sigma_1(y) \neq \sigma_2(y)$ with all else held equal. [SK: This notation is a bit weird, and also, if ALL else is equal...] There are two kinds of dependencies: data-dependencies, where the value of x directly affects y (such as $y = x + 1$), and control-dependencies, where the value of x indirectly affects y (such as **`if`** x **`then`** $y = 0$ **`else`** $y = 1$).

A PDG represents dependencies as paths between nodes. Say a PDG contains the edge:

$$p_{\mathsf{src}} \mathbin{@} cs_{\mathsf{src}} \xrightarrow{ek \mathbin{@} cs_{\mathsf{eff}}} p_{\mathsf{dst}} \mathbin{@} cs_{\mathsf{dst}}$$

Then it should be the case that p_{dst} at cs_{dst} directly depends on the value of p_{src} at cs_{src} due to the effect ek at cs_{eff}. For example, a PDG for the instruction $y = x + 1$ would contain the edge $x \xrightarrow{\mathsf{data}} y$ (ignoring call strings). If there exists a path from p_{src} to p_{dst} in the PDG, or more formally:

$$p_{\mathsf{src}} \mathbin{@} cs_{\mathsf{src}} \xrightarrow{*} p_{\mathsf{dst}} \mathbin{@} cs_{\mathsf{dst}}$$

Then it should be the case that p_{dst} at cs_{dst} transitively depends on p_{src} at cs_{src}.

We use the phrase "it should be" to indicate that this is the ideal case where the PDG faithfully represents dependencies. Paralegal guarantees that the PDG contains all true dependencies for programs which use no unsafe features (e.g., FFI or pointer arithmetic). We discuss the cases outside this guarantee in §4.1.3.

4.1.2 Analysis

To construct a PDG, Paralegal statically analyzes each instruction that is reachable from the entrypoints of the analysis. Much of the PDG construction is standard practice—for details, see Ferrante et al. [34]. Our analysis departs from standard practice in three key ways.

Monomorphization. Given a function call $f(p_1, \dots, p_n)$, Paralegal must determine which function f refers to. For example, the expression `x.to_string()` dispatches the trait method `to_string` based on the type of `x`. Paralegal monomorphizes function calls using context-sensitive statically-available type information. So if `x` has type `i32`, then Paralegal will be able to recursively analyze the implementation of `<i32 as ToString>::to_string`. Here, Paralegal leverages Rust as the target language for analysis. By design, Rust strongly encourages the use of static over dynamic dispatch—most popular Rust libraries do not use dynamic dispatch at all [67]. Therefore, Paralegal can frequently monormophize function calls in Rust applications.

Modular approximation. For dynamically-dispatched methods and third-party libraries, Paralegal may not find the implementation of f. In these cases, prior systems have either asked users to manually model the behavior of f (e.g., CodeQL maintains a large model of the C++ standard library), or these systems have made unsound assumptions about the behavior of f (e.g., Pidgin [41] assumes such functions have no side effects).

Paralegal instead uses the Rust type system to approximate the behavior of f automatically, soundly, and precisely. Paralegal builds on the technique used in the Flowistry information flow analyzer for Rust [27]. Specifically, a Rust function's type signature provides two key pieces of information:

1. *Mutability:* In Rust, a program is not allowed to mutate data accessible from an immutable reference. Paralegal can therefore assume that function calls do not mutate immutable references. For example, Rust's `HashMap<K, V>` has a method (slightly simplified) for removing a value:

   ```
   fn remove(&mut self, key: &K)
   ```

   Paralegal can assume that `HashMap::remove` only mutates `self` and not `key`.

2. *Aliases:* In Rust, all references are annotated with a *lifetime* that indicates which objects a reference can point to. This means Paralegal can perform an alias analysis on black-box functions. For example, `HashMap<K, V>` has a method for getting a value by key:

   ```
   fn get<'a, 'b>(&'a self, key: &'b K) -> Option<&'a V>
   ```

   This method returns a reference to a value of type `&V`. Because the reference has lifetime `'a` and not `'b`, Paralegal can assume that `&V` points to `self` and not `key`.

A key finding from the Flowistry evaluation [27] is that this *modular approximation* of a function's behavior is highly accurate compared to a more precise analysis with access to the function's source code. Therefore, Paralegal does not lose much precision with this technique, while saving developers substantial time and energy.

Function cloning. Paralegal achieves context-sensitivity via function cloning [71], where each call-site duplicates the sub-graph of the called function. This technique is maximally

precise, but it risks exponential growth in the size of the call graph. We demonstrate in §7.4 that Paralegal nonetheless runs reasonably fast on large Rust applications.

4.1.3 Limitations

Paralegal's analysis may construct a PDG that includes false dependencies and omits true dependencies. Some reasons are incidental and can be addressed with further engineering (discussed further in §5). But some reasons are fundamental, which we discuss here.

Overapproximation. Paralegal's PDG construction is similar to most static analyses in that it may include false dependencies by abstracting away relevant details of the code. For example, Paralegal will assume a branch could always reach both targets. This causes a false dependency in cases like `z = if false { x } else { y }`.

Unsafe code. Rust uses `unsafe` blocks to permit certain operations that cannot be verified safe by the compiler. This includes FFI (e.g., calling C libraries) and working with "raw" pointers (as opposed to "safe" references). Paralegal may omit true dependencies induced by unsafe code, such as aliases induced by pointer arithmetic.

Paralegal mitigates this limitation by using its type-based approximation. A common pattern in Rust FFI is to carefully encapsulate unsafe code within an API presenting a safe interface. Paralegal can then analyze such an API just at the interface level without observing the unsafe internals.

Interior mutability. Rust provides "interior mutability" primitives like `RefCell<T>` which permit mutating data behind an immutable reference—that is, one can turn an `&RefCell<T>` into an `&mut T`. This special case violates the assumption of the type-based approximation that a function cannot mutate immutable references. Therefore, Paralegal may omit true dependencies when approximating calls to functions using interior mutability. This limitation also extends to concurrent shared memory, such as locks and mutexes.

External effects. Paralegal may omit true dependencies induced by effects on external systems like a file system, OS, or database. For example, if `f` is a `File` called `foo.txt`, then Paralegal understands that `f.write(bytes)` affects `f` because `write` requires a mutable reference to `f`. But Paralegal does not understand that `f2 = File::open("foo.txt")` is effectively an alias on `f`, and that `f2.read()` should depend on `f.write(bytes)`.

4.2 Markers

Privacy engineers write policies in terms of semantic concepts, e.g., a policy might check if "sensitive data" flows to "public sinks." They define these concepts as Paralegal *markers*, which application developers apply to source code through inline annotations (e.g., `#[marker(sensitive_data)]`) or an external configuration file. Paralegal supports marking function arguments, return values and type definitions. This distribution of labor makes policies significantly less verbose compared to systems without this distinction (§7.2.2).

Marker propagation. Paralegal propagates these markers to the PDG, creating a marked PDG. Markers which are attached to function arguments propagate to the PDG nodes representing the respective formal arguments or return of the function. [Carolyn: This is confusing–function arguments should only go to formal parameters, not the return, unless you're accounting for the case when an argument flows to the return value? Also FWIW,] Paralegal also allows markers on types and emits a relation of such marked types. [Carolyn: I don't understand this sentence: how are you defining relation?] For each node which is of a type that *contains* a marked type `T`, e.g. `T`, [Carolyn: how does T contain a T? All types contain themselves by this logic, which means this isn't a special case at all.] `&T`, `Vec<T>`, Paralegal emits the association of the type and with the node, from which a policy may later recover the association with the marker. [Carolyn: What is an "association of the type"?] Paralegal's PDG is field-sensitive, except if the covered code never uses the fields individually, in which case Paralegal overtaints and associates marked fields of a type with the node representing the entire type.

Markers also inform the edges added to the PDG, since they represent sources, sinks, barriers etc. for the policy. [Carolyn: I don't get what this sentence is trying to say–I know that markers represent semantic concepts, but what does that have to do with edges?] At marked nodes, Paralegal collapses field and allocation sensitivity [Carolyn: what is allocation sensitivity?] , adding incoming and outgoing edges for all nodes that represent memory locations that are reachable from the marked node. [Carolyn: How is this different from what we do at unmarked nodes?]

[Carolyn: I didn't understand the weeds of this Marker Propogation section. I also think the section is missing a "so what" conclusion about why the reader should care about these technical weeds–how do these design choices make Paralegal more precise, efficient, etc.?]

Adaptive inlining. Paralegal also uses markers to optimize PDG generation. Policies can only talk about paths between marked nodes, so subgraphs of the PDG without any markers are irrelevant to policy checks. Therefore, when analyzing an instruction that calls a function f, Paralegal omits generating the subgraph for f and instead approximates its effects via type signature if there are no markers reachable from fs body. This optimization substantially improves the performance of PDG generation (as we show in §7.5).

4.3 Policies

A policy is an assertion about permissible paths in the marked PDG. Paralegal provides a high-level language with controlled natural language syntax that mimics the structure of legal documents. Policies are compiled by Paralegal into a Rust program that uses a low-level API to query information from the marked PDG.

The base primitives of the DSL are markers, named variables, and expressions that relate markers to each other via

their relationships in the PDG.

Primitives include:

- `a "value" marked sensitive`
  binds PDG nodes marked `sensitive` to the name "value".
- `"value" goes to "sink"`
  Checks if a path from `"value"` to `"sink"` exists in the PDG.
- `"value" affects whether "operation" happens`
  Hods if `"operation"` has a control dependency on `"value"` (or if there is a data-dependency from `"value"` to an intermediate that controls `"operation"`).
- `"value" goes to "sink" only via "disclosure"`
  Hods if every path from `"value"` to `"sink"` contains at least one node in the set `"disclosure"`.

Primitives are then composed as first-order logical formulas. For example, a formula like

$$\forall x \in X.\ \exists y \in Y.\ P(y) \implies (P(x) \vee S(x,y))$$

can be expressed in the policy DSL as:

```
1. For each "x" in X:
  A. There is a "y" in Y where:
    a. If P("y") then:
      i. P("x")
      or
      ii. S("x", "y")
```

Paralegal compiles this program into a Rust program that uses standard iterator operations:

```
X.iter().all(|x|
  Y.iter().any(|y|
    !P(y) || P(x) || S(x, y)))
```

The DSL requires policy writers to explicitly delineate the scope of each line in their policy with bullets (e.g., 1, A.) to avoid ambiguity. This structure of nested clauses, inspired by the design of legal documents, seeks to achieve readable and unambiguous properties.

The policy DSL intentionally abstracts away some details of the PDG. For example, if a policy writer wants to specify that `A` has control flow influence on `B`, in the DSL they write that "`A` affects whether `B` happens." Paralegal compiles this relation to `influences_ctrl(A, B)`. Often, this is what the policy writer intended. If, however, the policy required a *direct* control flow edge between `A` and `B`, any intermediate data flow (even an innocuous dereference) would cause the policy to fail. [SK: Is this backward? The policy writer wants a strict thing, but expresses a loose thing. Loose means more programs pass, not more programs fail?] For these scenarios Paralegal allows policy writers to specify their policies directly in the lower level query API. The query API provides direct access to the marked PDG and can express more subtle policies but is more verbose and error messages must be created manually.

A full grammar for the policy DSL is in Appendix B.

4.3.1 Policy Soundness

In cases where Paralegal cannot statically determine if a dependency will actually exist at runtime, Paralegal will over-approximate the dependencies in the program. This is the sound choice for policies reason about non-interference, such as access control and data confidentiality. These policies boil down to some version of "private data *must not* flow to public sinks", e.g. absence of a flow. By recording all *potential* edges Paralegal guarantees that if the policy passes there is no possibility of a flow. However it is insufficient for soundness with respect to all policies in Paralegal. The policy language allows reasoning about the existence of flows, as is the case in "user data *must flow* to a deletion sink". If Paralegal recorded only *guaranteed* dependencies, then the policy passing means there is indeed a flow.

In conclusion, in a system with Paralegal's expressiveness the choice of sound approximation depends on the policy. This dependency is non-trivial as Paralegal allows policy writers to freely mix statements about existence and absence of flows, meaning a single policy may not have a globally consistent sound approximation choice. Therefore Paralegal cannot currently guarantee soundness for all policies, though we have in practice not observed false-negatives. We believe that future work can augment the PDG construction to make policy-informed approximation choices, creating a system that is sound with respect to both types of policies.

4.4 Error Messages

Paralegal's error reporting helps developers diagnose a failing policy. As shown in Figure 5, error messages print the violated rules and the source code location that instantiated a given quantified variable. Paralegal reports them with a diagnostics framework inspired by the Rust compilers's error messages, relating graph nodes to snippets of the source code. Developers using the Rust API can also use this diagnostics framework directly to create customized error messages.

5 Implementation

Our Paralegal prototype consists of 15.1k lines of Rust and is implemented as a plugin to the Rust compiler.

Multi-crate support. Paralegal facilitates analysis across multiple crates by persisting the Rust intermediate language MIR to disk as well as the outlives relationships of lifetimes, an output of the Rust type checker, and any Paralegal annotations. During PDG construction the bodies of reachable functions are lazily loaded and a PDG is generated if markers are reachable. Paralegal's PDGs can this span all crates for which cargo initiates compilation in the process of building the target application. A PDG cannot extend to externally provided shared or precompiled libraries; in such cases the modular approximation is used. Source location information is preserved and error messages may reference precise locations in any loaded crate.

Await. Paralegal deliberately discards all control flow introduced by the state machine created from desugaring `await`. A malicious actor can craft code which hangs the program indefinitely using this state machine. However as this is a limited threat and Paralegal targets well-meaning developers

this is an acceptable limitation and prevents false-positives in policies involving control flow.

Marker limitations. Paralegal's marker annotations currently only support function arguments and types. These represent the common boundaries for semantic meaning of source code elements. In our use cases we found two instances however where markers needed to be applied to fields of a type or constants. This can be worked around but we plan to add support for this in the future.

6 Case Studies

We now discuss our experience applying Paralegal to eight real-world Rust applications. We tried to pick popular, production-level Rust applications that cover a range of domains, features, and coding styles (Figure 7). Their policies cover privacy properties from classic access control and security to data deletion and expiration, as well as purpose limitation.

6.1 Policies

We initially wrote policies by examining source code, then writing Graph Query API policies about the flows we expected to see in the PDG. But these policies used more markers and were more closely tied to source code than necessary. Hence, we switched to defining the policy first, using application functionality and documentation (as a privacy engineer would), then applying those policies to source code.

We found this approach made it easier to define policies, and that these new policies were clearer, more concise, and more portable to other applications. Privacy-related properties of applications are often obvious from the UI, functionality (e.g., account deletion), or documentation. By contrast, navigating a large, unfamiliar codebase to search for privacy-relevant sections is much harder. This experience inspired us to design the policy DSL. In all but two cases (see §6.3), our DSL policies, written without knowledge of source code, matched application semantics without revision.

We found that Paralegal was expressive enough to represent all the policies we wanted to check. Two applications (mCaptcha and Plume) use identical data deletion policies. Three others (Hyperswitch, Lemmy, and mCaptcha) use the same access control policy except for their application-specific marker names. For policies that were fundamentally dynamic, we were able to define static approximations. For instance, Freedit, a social media platform, stores a user's viewing history, but deletes the data after three days. Since the current time is only available at runtime, Paralegal cannot directly verify that Freedit obeys the three day expiration limit, However, it can approximate this policy by checking that `viewing history` flows to an `expiration check` and that `expiration check` has control flow influence on a `deletes`.

6.2 Markers

We found it easiest to apply markers to applications with modularized code that has clearly defined semantics. Applications with specialized delete or authorization check functions were simpler to apply markers to than applications that inline such logic inside large functions. We also found that many of our markers (e.g., `user data`) could be applied to types alone. Idiomatic Rust programs often define fine-grained, custom types which are clearly named for the type of data they represent.

We mainly made two types of source code change. The first is because our prototype cannot apply markers to constants or to fields of a type, so we defined no-op functions `mark_{marker_name}(&data)`, whose sole purpose is to apply the appropriate marker to `data`. Second, if applications inlined privacy-critical functionality inside a larger function, there was no way of applying a marker to just the relevant lines. With additional policy specification effort these cases can be accommodated, but we opted for an approach with cleaner separation of concerns and extracted the logic into a helper function and marked that function. In this way Paralegal encourages developers to cleanly demarcate privacy relevant code. A comprehensive overview of four additional small tweaks we made is in the appendix §A.1.

6.3 Selected Experience Reports

Two cases stand out in our experience of developing policies for our use cases which we describe here in more detail.

Atomic is a graph database that lets users create, edit, and share graph-structured data [4, 7]. Each time a user modifies a database resource, Atomic stores a signed *commit* record. Before creating a commit, the application must verify that the user has permissions to modify that resource. Crucially, this authorization must happen *before* updating the the database resource [6]. Our Paralegal policy asserts that if a resource is modified, the resource flows to an authorization check, and the authorization check has control flow influence on the modification.

When we ran this policy on the application, it failed, even after the commit where the developers fixed the bug. Upon investigation, we realized that we had missed an edge case: if a resource is new, the application *first* modifies the resource to set default permissions, *then* executes the permission check. We had missed this logic initially because it was unclear in the application's documentation. This exceptional case can be integrated with no source changes by augmenting the policy to exempt this case (which we do in the experiment §7.3). However this unnecessarily burdens the policy with matching a specific code pattern. Instead we advocate for a solution that extracts this benign modification into a helper function with an exception marker and exempt this marker in the policy (which we use in the experiments in §7.4). This version ensures only a specific instance of the code pattern is exempt and it clearly delineates the exception in the code itself.

mCaptcha is a proof-of-work–based CAPTCHA service focused on privacy [54]. Website owners register sites with mCaptcha and embed mCaptcha API calls into their sites. mCaptcha's PoW algorithm features a tunable "difficulty",

Application	Type	LoC	Policies	Unique Markers	Marked Locations	Entry pts.
Atomic [7] (v0.34.2)	Graph DB	9.6k	Access Control	4	4	1
Contile [26] (v1.11.0)	Advertising	4.9k	Purpose Limitation	3	5	1
Freedit [37] (v0.6.0-rc.3)	Social	6.6k	Data Retention/Expiration	5	5	4
Hyperswitch [39] (v0.2.0)	Payments	198.9k	Credential Security, Limited Data Collection	6	7	3
mCaptcha [54] (v0.1.0)	Authentication	10.6k	Data Deletion, Limited Data Collection	5	5	2
Lemmy [50] (v0.16.6)	Social	31.4k	Access Control	8	145	72
Plume [60] (v0.7.2)	Blogging	21.4k	Data Deletion	7	7	1
WebSubmit [3] (v1.0)	Homework	1.6k	Data Deletion, Access Control	11	18	3

Figure 7: *Case study applications with code size, policies, and Paralegal marker statistics. "Marked Locations" indicates the number of source code entities (arguments, returns, etc.) to which we applied markers; "Entry pts." is the number of analysis entry points.*

Application	Bug Description	Reference	Paralegal Policy
Plume [60]	Comments and Media not deleted when a user is deleted.	commit 19f1842, issue 806	If a user flows into a delete function, all types marked as user data must flow into a delete function.
Atomic [7]	Users can grant themselves write access to data without prior access.	commit 46a503a	Write permissions must be checked before a resource is updated.
Lemmy [50]	Banned or deleted users can log into a server.	commit b78826c	(1) A user deletion check and a ban check must influence every database access, except those reading the active user. (2) Community deletion check and user ban check must influence every database write to a community.
	Deleted users can perform actions in a server.	commit 2966203	
	Users can write to a deleted community.	commit 2402515, issue ANON	
	Banned users can act in communities.	issue ANON	

Figure 8: *Paralegal found seven bugs, including two previously unknown ones, in three applications: Plume [60], Atomic [7], and Lemmy [50]. "ANON" refers to issues filed by this paper's authors, which we elide during review for anonymity.*

designed to balance security and latency.

In addition to adding a data deletion policy, which we reused from Plume (Figure 3a), we observed that mCaptcha collects optional consent to "gather performance statistics [...] and make them available to other mCaptcha installations" for difficulty tuning [40]. We wrote a policy to enforce that mCaptcha checks whether a user has opted in before storing their data, but the code failed this policy.

On discussing the issue with the mCaptcha developers, it turned out that we had misinterpreted their privacy goals—statistics are always collected, even if website owners don't opt into sharing them. The developers, however, indicated that they would be open to reconsidering this choice [53]. In addition, the discussion helped the mCaptcha developers find a (related) data integrity bug: they unintentionally deleted statistics when a user revoked their opt-in consent. This illustrates how the discipline enforced by Paralegal helps developers reflect on their code and find issues.

7 Evaluation

Our evaluation of Paralegal seeks to answer five questions:

1. Does Paralegal find bugs that result in privacy violations in real applications? (§7.1)
2. How does Paralegal's expressiveness and ability to handle real-world code bases compare to the state of the art? (§7.2)
3. How much developer effort is required to keep Paralegal's policies and markers in sync with a changing codebase? (§7.3)
4. Is Paralegal fast enough for interactive or CI use, and how does its runtime scale with the size of the analyzed codebase? (§7.4)
5. How do Paralegal's core techniques and optimizations contribute to its effectiveness? (§7.5)

Setup. All experiments run on a server with 8 Intel Xeon E3-1230v5 CPUs (3.4 GHz) and 64 GiB RAM, on Ubuntu 20.04 using Rust nightly-2023-08-25.

7.1 Finding Privacy Bugs with Paralegal

We applied Paralegal to eight applications (Figure 7) to investigate its ability to discover bugs. [SK: I think fig 5 is a more useful cite, especially given the current state of sec 6.] A good result for Paralegal would show that it finds previously reported bugs as well as new bugs, all without generating many false positives.

Figure 8 summarizes the bugs Paralegal found: Paralegal found two previously unknown privacy bugs that were confirmed by the developers, as well as five previously known privacy bugs that the developers had already fixed.

Plume and AtomicData. Paralegal found three known bugs in Plume and Atomic [5, 58, 59]. In Atomic Data, the policy passes Paralegal after the developers' fix [5]. In Plume, however, the policy still fails after the developers' fix [58],

since even though the application now deletes comments correctly, it still does not delete users' uploaded media (a separate known bug [59]).

Lemmy. We ran Paralegal on 72 HTTP endpoints in Lemmy. Paralegal found two bugs previously fixed by the Lemmy developers, and additionally reported two new bugs.

Known Bugs. A user may not access a Lemmy instance if their account has been banned or deleted. However, Lemmy's helper for authorizing already logged-in users omitted a check for whether their account is deleted [45]. Consequently, Paralegal flagged the instance authorization policy in all endpoints. After the fix [29], 71 endpoints passed the policy, but Paralegal still reported a failure in the login endpoint. Since the the login endpoint doesn't use the authorization helper for logged-in users, it still did not check for account deletion [48]. The login endpoint was also missing a check if the user had been banned. After a second fix by the Lemmy developers [30], all 72 endpoints passed the policy.

New Bugs. Lemmy prohibits users to write in deleted communities: if a community was removed for problematic content, for example, users must not be able to make new posts. The Lemmy developers already found missing community deletion checks in five endpoints [31, 46], but Paralegal found 16 further endpoints lacking these checks.

In addition, a banned user should not be able to write to a community. Paralegal reported that some Lemmy controllers are missing these community ban checks. This allows bypassing access control: for example, a banned community moderator can immediately unban themselves. The Lemmy developers confirmed both bugs [47, 49].

7.2 Comparison with Related Work

Next, we evaluate how Paralegal's expressiveness and ergonomics compare to prior approaches, using the 10 policies from our case studies. We consider two baselines: *(i)* classic IFC based on a lattice of security labels; and *(ii)* CodeQL, a recent code analysis engine deployed at GitHub [14]. We tried to express the policies of our case study applications for both baselines. In IFC, this boils down to imposing a label hierarchy and determining declassification points. For CodeQL, we implemented appropriate queries.

7.2.1 Comparison with IFC

In classic IFC, labels applied to data form a lattice, and the system ensures that data with low-security labels is free of influence from values with high-security labels (the "non-interference" property). A range of security and privacy concerns fit into this model, however it's expressivity is limited compared to generic tools (CodeQL and Paralegal). As such we expect it can express some, but perhaps not all of the policies.

Figure 9 (IFC column) shows that IFC can enforce five of the ten policies. Two properties, in Contile and Hyperswitch, directly fit IFC's notion of restricting flows of high-security values into low-security sinks. Policies in Lemmy, Atomic, Hyperswitch and mCaptcha require access control or consent checks prior to operations on data, which IFC can approximate via selective declassification. However, this strategy *requires* code changes (🔧) that turn the check into a data flow operation, as IFC cannot declassify (i.e., remove a label) via control flow. WebSubmit's Access Control policy is a complex, data-dependent property that requires a list to contain only "blessed" receiver email addresses; since IFC never prohibits mixing lower-security values into data, it alone is insufficient to enforce this policy. Disciplined use of data types, combined with IFC, can enforce this property, but requires substantial code rewrites. (Freedit's Data Retention policy contains a similar pattern.) Finally, Data Deletion and Retention policies rely on a "must reach" pattern that requires a value to reach a sink (i.e., deletion). Since IFC can only check the absence of prohibited flows, but not mandate the existence of data flows (i.e., enforce safety but not liveness), it fails to express or enforce these properties.

Application	Policy	IFC	CodeQL	Paral.
Atomic	Authorization	✓🔧	✗ †∗T¶	✓
Plume	Data Deletion	✗	✓	✓
Hyperswitch	Credential Security	✓	—	✓
Hyperswitch	Limited Collection	✓🔧	—	✓
Websubmit	Data Deletion	✗	✗ ‡A	✓
Websubmit	Access Control	✗	✗ ∗	✓
mCaptcha	Data Deletion	✗	✓ T¶	✓
mCaptcha	Limited Collection	✓🔧	✓ T¶	✓
Freedit	Data Retention	✗	✗ ‡	✓
Lemmy	Access Control	✓🔧	(✓) †¶	✓
Contile	Purpose Limitation	✓	—	✓

Figure 9: *Paralegal expresses and enforces properties that baseline approaches (classic IFC and CodeQL) struggle with. ✓ indicates success, (✓) success on some versions, ✗ failure; 🔧 denotes required code changes, and we indicate CodeQL results affected by †control flow analysis, ‡hidden source code, ∗taint propagation to/from structures, Aalias analysis, Tunconstrained templates, and ¶async code.*

7.2.2 Comparison with CodeQL

CodeQL extracts a program's AST and derived information into a database, and developers write queries against it in a Datalog-like language. Since CodeQL lacks support for Rust, we translated the relevant parts of each application into C++, the most similar language to Rust with CodeQL support. Two applications (Hyperswitch and Contile) have policies that touch large amounts of code (1.3–1.5k LoC, plus library functions), so we omit them. For Lemmy, we translate one representative endpoint, but omit 71 structurally similar ones. Our ports seek to faithfully reproduce the control and data flow of the Rust code, but replace Rust's `Result` with exceptions. We implement the queries using CodeQL's libraries

for data flow [17] and control flow analysis [18]. CodeQL's query language can express all ten policies, but its analysis engine fails to enforce some of them; when this happened, we debugged the issue sufficiently to pin it down to one or more limitations in CodeQL or its libraries, and stopped investigating further (i.e., our analysis may have missed further problems masked by the initial failure).

Policy Effectiveness. Figure 9 (CodeQL column) shows the outcome of each CodeQL policy. Plume's Data Deletion policy works; as do mCaptcha's policies (with caveats, see below) and Lemmy's Access Control policy in some versions of the code. Other properties fail due to combinations of limitations in CodeQL, identified by symbols:

(†) Control flow analysis in CodeQL is not inter-procedural. This causes a false positive with Lemmy's access control policy, which uses a helper function for access control checks. CodeQL's control flow analysis also ignores certain complex control flow patterns [21] (intentionally [20]), which causes a false positive in Atomic.

(‡) CodeQL assumes no data flows exist through library calls whose source code is unavailable. This affects Websubmit's Data Deletion policy, which hinges on detecting data flow into a value-type constructor provided by a database library; similarly in Freedit's Data Retention, CodeQL misses data flow through a database method. While CodeQL allows developers to write manual models for library functions, this is onerous and error-prone.

(*) Taint propagation to fields within a structure and from fields to the structure itself requires manual modeling [19]. This complicates policies over markers on a structure that expect to detect flows of that structure's fields into a sink; or policies over markers on elements of a container that expect to detect flows of the container itself. The CodeQL developers manually model taint propagation for collection elements for some standard library data structures but do not (yet) model `std::unordered_map`, which Atomic and Websubmit's Access Control use.

(A) CodeQL currently lacks an alias analysis for C++, due to performance issues [23]. As a workaround, the developers manually model methods, e.g., for `std::vector::push_back`, but CodeQL lacks a default model for initializer lists, which show up in Websubmit's code checked by the Data Deletion property, and require reasoning about aliasing.

(T) Our C++ ports replace trait-constrained Rust types with C++ templates. CodeQL cannot analyze code with uninstantiated template parameters [22], even when constrained by concepts; Paralegal can analyze uninstantiated, parameterized code by exploiting trait constraints with the modular approximation.

(¶) C++ has no `async` language feature but provides an `async` library function which serves a similar purpose. When we use it in place of Rust's async, CodeQL misses data and control flow dependencies through this function [24]. This causes both false positives and false negatives in Atomic, mCaptcha, and Lemmy. We work around this by using synchronous C++ code.

The last two limitations in part result from impedance mismatches between C++ and Rust, so we report them in Figure 9, but consider the affected policies enforceable.

By contrast, Paralegal has an inter-procedural control flow analysis, models the effects of unknown library code via modular approximation, collapses taints in and out of structures without developer effort, and relies on Flowistry's modular alias analysis for Rust. It also handles uninstantiated type variables constrained on traits by approximating the effect of trait methods, and handles Rust's async language feature. As a result, Paralegal successfully expresses and enforces all these policies.

Policy Structure. Paralegal is designed to distribute work between markers, program analysis, and policy. CodeQL does not have markers (or a marked PDG), and it provides only low-level program analysis primitives. Therefore, we asked: how much of a CodeQL query corresponds to work done by Paralegal outside of the policy? We answered this question by qualitatively labeling each chunk of a CodeQL query (precisely, a CodeQL *predicate*) as either "policy" or "not policy" code. For instance, this CodeQL predicate in the Plume policy to find deletion functions is labeled as "not policy" because it serves the same function as a Paralegal marker:

```
predicate is_delete(DataFlow::Node n) {
  n.asParameter().getFunction().getName()
   .regexpMatch(".*deleteAny.*") and
  n.asParameter().getIndex() = 0
}
```

To check the reliability of this qualitative judgment, the first two authors independently labeled 30 predicates and, after resolving one conflict, achieved 96% agreement.

Across all CodeQL policies, 36% of code was labeled as policy code. From our observations, the remaining 64% of code fell roughly into three categories: (1) predicates identifying code elements (i.e., markers), (2) predicates defining the semantics of external code (i.e., models), or (3) predicates defining primitives for code analysis. This result shows that Paralegal's design effectively separates concerns that are otherwise mixed together in existing systems.

7.3 Developer Ergonomics

We now evaluate the maintenance effort imposed on developers by Paralegal as a code base evolves. A developer may need to update her Paralegal configuration as follows:

1. she may need to move a marker, e.g., because argument order changes or a function is renamed;
2. she may need to add markers to new code entities;
3. she may need to adjust marker names because of changes to the semantic meaning of a code element; or
4. the privacy engineer may need to change the policy because it made assumptions about application semantics that no longer hold or because the policy goal changed.

Mechanical marker movement is easy, but changes to what markers mean impose higher cognitive developer burden, and policy changes require involving the privacy engineer.

To evaluate this, we run Paralegal on all commits of Atomic, starting with the introduction of the (buggy) permission model, using the same policy and markers as before. A good result for Paralegal would show that changes to markers or policy are rare—i.e., that the policy is abstracted away from code details, and that the markers are at a granularity that avoids frequent changes.

The experiment covers 1,024 commits between June 9, 2021—when Atomic introduced the feature that our policy targets—and March 26, 2024. 87 of commits failed to compile and we skip them. The code that Paralegal analyzes for the policy in question is a subsection of the entire application, but changes over the course of the experiment. On average, Paralegal's analysis touches 593 lines (min: 242, max: 815). Within the 937 functional commits, 51 commits modify at least one of those analyzed lines (42 lines change on average).

Paralegal is able to detect the permission bug in each commit before the developers' fix and passes on every commit thereafter. [SK: Reader will want to know how many commits in and what date.] We found two commits that required changes to markers, and none that required changes to the policy. The marker changes are: *(i)* a simple rename of a function (`aba49fe`); and *(ii)* replacing a previously-marked function with one that takes additional configuration arguments (`e0cf2d1`). Both changes are simple marker movements, and caused Paralegal's policy checks to fail. Hence, Paralegal would have alerted the developer even if she neglected to move/adjust the markers previously.

7.4 Performance and Scalability

We now investigate Paralegal's runtime and how it scales with application size. Paralegal's runtime consists of two primary components: extracting the PDG from the Rust source code, and checking policies against the PDG. Our experiment measures these two components, in two settings: *(i)* when running Paralegal end-to-end for all analysis entry points; and *(ii)* on individual entry points. Runtimes do not include compiling dependencies with rustc. On a fresh codebase compiling dependencies and persisting the IR for multi-crate support makes up on average 95% of the runtime of the analyzer. However caching means none of this work is repeated on subsequent runs unless code in the dependencies changes, which is rare.

End-to-End Runtime. This experiment runs Paralegal for all endpoints in sequence, like a CI job would. A good result for Paralegal would show that it runs in seconds.

Figure 10 shows the results. Most applications finish in less than 3 seconds and the runtime is dominated in most cases by PDG construction. The Hyperswitch and mCaptcha crates are the largest and incur more overhead from running rustc. Hyperswitch also has the largest individual PDGs, increasing

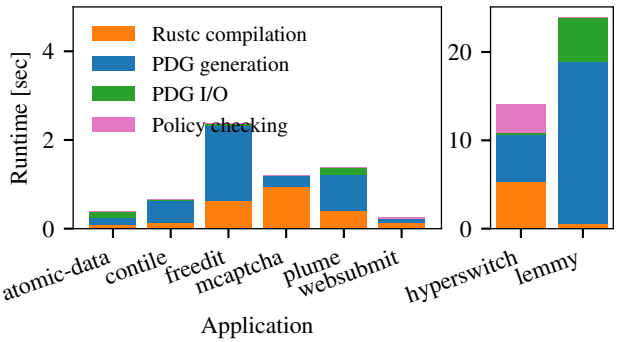

Figure 10: *Paralegal runs end-to-end in seconds to minutes for our applications, and most of its runtime is in marked PDG generation. Lemmy, with 72 entry points, takes longest.*

		Per entry point [average]		
Application	Entry pts.	LoC analyzed	Nodes	Edges
Atomic	1	209	1776	6012
Contile	1	487	3752	19112
Freedit	4	267	2272	5953
Hyperswitch	3	848	5593	20853
Lemmy	72	271	2326	10280
mCaptcha	4	32	291	671
Plume	1	189	1566	4751
WebSubmit	3	75	113	187

Figure 11: *Paralegal's marked PDGs have 0.4–5k nodes and 0.2–12k edges. PDG size is generally, but not strictly, proportional to the number of lines of code analyzed.*

policy runtime. Lemmy contains the most endpoints, meaning more PDGs are constructed. Construction takes advantage of caching, but writing the many graphs does not, leasing to more time spent on I/O. Figure 11 shows that PDG size grows roughly proportionally to the number of lines of code analyzed. Policy checking takes 0.3–6.5 seconds. Overall, Paralegal's end-to-end runtime is acceptable for a CI setting.

Per-endpoint Runtime. In a more interactive setting like an IDE plugin, Paralegal might run on one or a few endpoints at time (e.g., the endpoint the developer is editing). We therefore measure the per-endpoint runtime, as a function of the number of lines of code analyzed.

Figure 12 shows the results. Most endpoints take a few seconds, and runtime grows with the code size analyzed (cf. Figure 11). The outlier is a smaller controller in hyperswitch, which is caused by both a more expensive policy and a fixed overhead for running rustc on a large crate. These results suggest that Paralegal is fast enough to be used interactively.

7.5 Drilldown Experiments

Finally, we measure the impact of key Paralegal components: [SK: add ref back for each one] the adaptive inlining optimization and compiling a high-level policy language to Paralegal's Rust API.

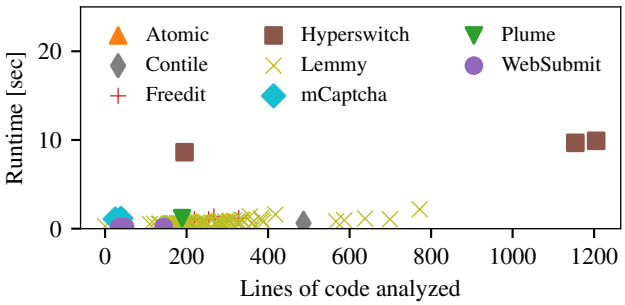

Figure 12: *Per-endpoint run times as function of the number of line of code analyzed for each entry point: Paralegal generally completes in under ten seconds.*

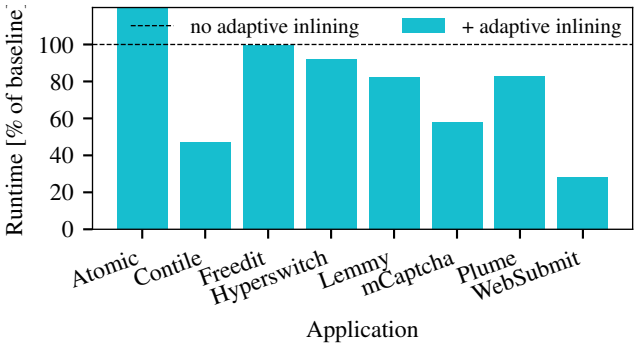

Figure 13: *The adaptive inlining optimization reduces end-to-end runtime up to 84% on large applications.* *[Justus: Ignore atomic for now, this seems to be a fluke. Experiments are being repeated with repetitions for assurance.]*

Adaptive Inlining. We run Paralegal on the eight applications with adaptive inlining turned off and on, and measure end-to-end runtime. A good result would show that the optimization reduces PDG construction, graph size, and the time to check a policy over the PDG.

Figure 13 shows the runtime, normalized to Paralegal without adaptive inlining (i.e., lower is better). Update numbers once we're certain. Adaptive inlining reduces end-to-end runtime by 1–84%, and speeds up PDG construction by 1.08–6.3× and policy checking by 1.29–9.2×. Lemmy and Contile benefit especially, owing to deep inlining chains that the optimization avoids. However, the optimization sacrifices some precision: Contile reports a false positive with adaptive inlining enabled (cf. Appendix A). Developers may wish to enable adaptive inlining during interactive use, but re-run with it disabled on errors or before releasing code.

WebSubmit experiences a 1.4% performance reduction because the application is too small for the optimization to avoid inlining any functions, but it must still perform work to check if it would be safe to avoid the inlining.

High-Level Policy Language. We compare Paralegal's runtime for policies written in its high-level policy language with the runtime of an equivalent, hand-optimized policy written against the Graph Query API. Manually-written policies may be more efficient because our compiler generates unoptimized code and because developers may leverage Rust features to hand-optimize their policy checks. We find that, across the eight applications, the manually optimized Graph Query API policies are only 2–10% faster than compiled high-level policies. This is a good result for Paralegal, as it indicates that compiled high-level policies achieve comparable efficiency to low-level Rust policies.

8 Related Work

Paralegal exists within an extensive literature on static analysis, policy enforcement, and bug finding.

[Malte: Add BlockAid.]

§2 covered the core prior work and its relationship to Paralegal. Here we discuss other categories of systems for policy enforcement that make different assumptions and design decisions.

Dynamic IFC enforces security policies at runtime by attaching security labels to values and propagating them through the program. Dynamic analysis can increase precision (HLIO [12]), enforce policies that depend on runtime knowledge (Sesame [28]) and handle languages which are difficult to analyze statically (Jacqueline [73], Riverbed [69] and Resin [75]). However the tracking incurs runtime overheads which increase with the level of precision used and requires a runtime buy-in. In addition, violation cases lead either to crashes or confusing application behavior in deployment. Paralegal uses static analysis avoid both of these problems.

Bug-finding tools intelligently explore the state space of a program or API to find bugs. Bug finders are easy to adopt and handle a variety of code styles but are usually specialized to a domain such as concurrency [63], distributed systems [68], or file systems [11, 43, 56], and while some offer customization, such as the file-system bug finder eXplode [74], their properties are ultimately hard-coded.

Paralegal requires some annotation effort (Figure 7) and encourages refactorings for clarity, it allows for checking both custom and general policies while retaining the compatibility with arbitrary code styles.

Code linters identify issues by looking for syntactic patterns such as AST fragments [55] or function names [51]. This approach is simple and practical, but limited in expressiveness and precision. Paralegal uses a more semantic model of the code (a dependency graph) to permit more expressive policies. [SK: and more robust enforcement?]

9 Conclusion

Paralegal is a new static analysis tool for checking high-level privacy properties against practical Rust code bases. [SK: I find it very odd to associate "practical" with *code bases*. Sounds off.] Paralegal extracts a marked Program Dependence Graph (PDG) from Rust programs lightly annotated with markers, and checks privacy properties against it. [SK: Doesn't mention rich policy language

for properties or the DSL.]

Our evaluation shows that Paralegal finds actual privacy bugs in real Rust programs, requires modest developer effort as the code base evolves, and runs in seconds to minutes, making it suitable for interactive and CI use.

Paralegal is open-source software [2].

References

[1] *Actix Web is a powerful, pragmatic, and extremely fast web framework for Rust*. URL: https://actix.rs/ (visited on 04/19/2024).

[2] Anonymous. *Paralegal*. 2024. URL: https://github.com/ANON/paralegal (visited on 04/16/2024).

[3] Anonymous. *Websubmit-rs: A Simple Class Submission System*. Redacted for anonymized submission. URL: https://github.com/ANON/websubmit-rs (visited on 04/06/2022).

[4] *Atomic Data is a modular specification for sharing, modifying and modeling graph data*. URL: https://github.com/atomicdata-dev (visited on 04/19/2024).

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

[6] *AtomicServer Privacy Policy*. URL: https://github.com/atomicdata-dev/atomic-server/blob/46a503a/lib/src/commit.rs#L109 (visited on 04/19/2024).

[7] *AtomicServer: a lightweight, yet powerful CMS / Graph Database*. URL: https://github.com/atomicdata-dev/atomic-server (visited on 04/19/2024).

[8] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. "QL: Object-oriented Queries on Relational Data". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Edited by Shriram Krishnamurthi and Benjamin S. Lerner. Volume 56. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 2:1–2:25.

[9] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. "Efficient and Flexible Discovery of PHP Application Vulnerabilities". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pages 334–349.

[10] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. "Everest: Towards a Verified, Drop-in Replacement of HTTPS". In: *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Edited by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Volume 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 1:1–1:12.

[11] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. "Specifying and Checking File System Crash-Consistency Models". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pages 83–98.

[12] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pages 289–301.

[13] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (Sept. 2010). Edited by Andrei Sabelfeld, pages 1157–1210.

[14] *CodeQL*. URL: https://codeql.github.com/ (visited on 12/03/2024).

[15] CodeQL Developers. *CodeQL C++ library models*. URL: https://github.com/github/codeql/tree/39a67b6e2e6490a9bd010db50e148f647765e9f7/cpp/ql/lib/ext (visited on 12/03/2024).

[16] CodeQL Developers. *CodeQL C++ Semmle models*. URL: https://github.com/github/codeql/tree/39a67b6e2e6490a9bd010db50e148f647765e9f7/cpp/ql/lib/semmle/code/cpp/models/implementations (visited on 12/03/2024).

[17] *CodeQL Documentation: Analyzing data flow in C and C++*. URL: https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-cpp/ (visited on 12/03/2024).

[18] *CodeQL Documentation: Using the guards library in C and C++: The controls predicate*. URL: https://codeql.github.com/docs/codeql-language-guides/using-the-guards-library-in-cpp/ (visited on 12/03/2024).

[19] CodeQL Github Issue #18098. *[C++] How to detect taint on elements in a collection*. URL: https://github.com/github/codeql/issues/18098#issuecomment-2504050120 (visited on 12/03/2024).

[20] CodeQL Github Issue #18099. *[C++] Fails to detect control flow influence of nested "if"*. URL: https://github.com/github/codeql/issues/18099#issuecomment-2500005480 (visited on 12/03/2024).

[21] CodeQL Github Issue #18100. *[C++] Control Flow Influence not detected interprocedurally*. URL: https://github.com/github/codeql/issues/18100 (visited on 12/03/2024).

[22] CodeQL Github Issue #18122. *[C++] Templatized code not found*. URL: https://github.com/github/codeql/issues/18122 (visited on 12/03/2024).

[23] CodeQL Github Issue #18151. *[C++] Taint analysis does not appear to handle aliasing*. URL: https://github.com/github/codeql/issues/18151#issuecomment-2506541314 (visited on 12/03/2024).

[24] CodeQL Github Issue #18171. *[C++] Support for std::async and std::thread*. URL: https://github.com/github/codeql/issues/18171 (visited on 12/03/2024).

[25] *Contile: README.md*. URL: https://github.com/mozilla-services/contile/blob/8170ad2626ddc96a6f74ca794a9c33e9cb76e6e3/README.md#contile-tile-server (visited on 04/19/2024).

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

[27] 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.

[28] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. "Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pages 709–725.

[29] *Don't allow deleted users to do actions*. URL: https://github.com/LemmyNet/lemmy/commit/2966203 (visited on 04/19/2024).

[30] *Don't allow login if account is banned or deleted*. URL: https://github.com/LemmyNet/lemmy/commit/b78826c2 (visited on 04/19/2024).

[31] *Don't allow posts to deleted / removed communities*. URL: https://github.com/LemmyNet/lemmy/commit/2402515 (visited on 04/19/2024).

[32] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)". In: *Official Journal of the European Union* L119 (May 2016), pages 1–88.

[33] Ming Fan, Le Yu, Sen Chen, Hao Zhou, Xiapu Luo, Shuyue Li, Yang Liu, Jun Liu, and Ting Liu. "An Empirical Evaluation of GDPR Compliance Violations in Android mHealth Apps". In: *Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2020, pages 253–264.

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

[35] 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.

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

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

[38] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. "Information-Flow Analysis of Android Applications in DroidSafe". In: *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*.

[39] *Hyperswitch: An open source payments switch written in Rust to make payments fast, reliable and affordable*. URL: https://github.com/juspay/hyperswitch (visited on 04/19/2024).

[40] *Introducing mCaptcha net*. URL: https://mcaptcha.org/blog/introducing-mcaptcha-net (visited on 04/19/2024).

[41] 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.

[42] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. "Cocoon: Static Information Flow Control in Rust". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024).

[43] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. "Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems". In: *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. Rome, Italy, 2023, pages 718–733.

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

[45] *Lemmy #1656: Deleted users can still make comments and posts*. URL: https://github.com/LemmyNet/lemmy/issues/1656 (visited on 04/19/2024).

[46] *Lemmy #1827: Posting to removed community isn't disallowed*. URL: https://github.com/LemmyNet/lemmy/issues/1827 (visited on 04/19/2024).

[47] *Lemmy #2372: Banned users can act in communities*. URL: https://github.com/LemmyNet/lemmy/issues/ANON (visited on 04/19/2024).

[48] *Lemmy #2372: Deleted account error*. URL: https://github.com/LemmyNet/lemmy/issues/2372 (visited on 04/19/2024).

[49] *Lemmy #2372: Deleted users can act in communities*. URL: https://github.com/LemmyNet/lemmy/issues/ANON (visited on 04/19/2024).

[50] *Lemmy: A link aggregator and forum for the fediverse*. URL: https://github.com/LemmyNet/lemmy (visited on 04/19/2024).

[51] Baptiste Lepers, Josselin Giet, Willy Zwaenepoel, and Julia Lawall. "OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pages 33–45.

[52] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. "Mining Node.js Vulnerabilities via Object Dependence Graph and Query". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pages 143–160.

[53] *mCaptcha #161: Privacy Bug: Over-collection in Opt-in Performance Statistics*. URL: https://github.com/mCaptcha/mCaptcha/issues/ANON.

[54] *mCaptcha: Proof of work based, privacy respecting CAPTCHA system*. URL: https://github.com/mCaptcha/mCaptcha (visited on 04/19/2024).

[55] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Transactions on Software Engineering* 36.1 (Jan. 2010), pages 20–36.

[56] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. "Finding crash-consistency bugs with bounded black-box crash testing". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pages 33–50.

[57] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. New York, NY, USA: Association for Computing Machinery, Jan. 1999, pages 228–241.

[58] *Plume #1144: delete comments when deleting users*. URL: https://github.com/Plume-org/Plume/commit/19f18421bcd9cb9d1654de24f9a04747691036b7 (visited on 04/16/2023).

[59] *Plume #806: Media not deleted after account deletion*. URL: https://github.com/Plume-org/Plume/issues/806 (visited on 04/19/2024).

[60] *Plume: A federated blogging engine based on ActivityPub*. URL: https://github.com/Plume-org/Plume (visited on 04/19/2024).

[61] François Pottier and Vincent Simonet. "Information Flow Inference for ML". In: *ACM Trans. Program. Lang. Syst.* 25.1 (Jan. 2003), pages 117–158.

[62] 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).

[63] Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. "WAFFLE: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pages 111–126.

[64] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. "If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Austin, Texas, USA, 2024, pages 63–78.

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

[66] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. "Securify: Practical Security Analysis of Smart Contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pages 67–82.

[67] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. "Verifying dynamic trait objects in rust". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pages 321–330.

[68] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. "Model Checking Guided Testing for Distributed Systems". In: *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. Rome, Italy, 2023, pages 127–143.

[69] 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.

[70] 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.

[71] John Whaley and Monica S. Lam. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. PLDI '04. Washington DC, USA: Association for Computing Machinery, 2004, pages 131–144.

[72] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pages 590–604.

[73] 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.

[74] Junfeng Yang, Can Sar, and Dawson Engler. "Explode: a lightweight, general system for finding serious storage system errors". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pages 131–146.

[75] 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.

[76] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. "Automated Analysis of Privacy Requirements for Mobile Apps". In: *2016 AAAI Fall Symposium Series*. Sept. 2016.

A Case Studies In-Depth

(1) Atomic is a graph database server that lets users create, edit, and share graph-structured data [4, 7]. It uses the Actix web framework [1].

Each time that a user modifies a database resource, Atomic stores a signed *commit* record. Before creating a commit, the application must verify that the user has permissions to modify that resource. Earlier "parent" commits for the resource specify permissions for later commits. For instance, a commit that creates a new resource must also specify which users are allowed to edit that resource. For each commit, the application must check the permissions on the commit's parent. Crucially, this authorization must happen *before* updating the in-memory copy of the database resource [6]. If the authorization check happened *after* applying the commit, the user could first change their parent commit to one that gives them the requisite permissions, thereby guaranteeing that they pass the permission check.

To encode this policy in Paralegal, we define a `commit` marker for the `Commit` type. We then create a `modify_resource` marker for all operations that modify in-memory references to resources. We add a `sink` marker to the write operation that flushes the modified resource back to the database. Finally, we add an `auth_check` marker for checking the user's permissions. The Paralegal policy enforces that if a `resource` flows to a `modify_resource` then to a `sink`, the `resource` flows to an `auth_check`, and there is a control flow influence from the `auth_check` to the `modify_resource`.

(2) Contile is the backing server of the Mozilla Tile Service, which serves as an intermediary between advertisers and the landing page of the Firefox web browser to "ensure customer privacy" in the advertising context [25]. Contile services small ads, called *tiles*, to Firefox users.

The information available to the application is centered around the `Tags` struct, which features the following comment referencing the fields `tags` and `extra` of the struct:

> Not all tags are distributed out. 'tags' are searchable and may cause cardinality issues. 'extra' are not searchable, but may not be sent to [Metrics].

The Paralegal policy ensures that the data from the `extra` field is withheld from advertiser search and that it is never sent to the metrics server. We marked the `extra` field as sensitive data. We then marked the sinks to which `extra` should not flow: two functions in `MetricsBuilder` and the `RequestBuilder::send` function in the `reqwest` http library.

When analyzing Contile with our adaptive depth optimization, many of the functions that modify `Tags` are abstracted only by type. Since those functions receive a `&mut Tags`, the resulting PDG reflects a potential flow between `extra` and `tags` in these functions. This causes a false-positive result. When running without the optimization, the field sensitivity is preserved, so the policy passes as expected.

(3) Freedit is a forum application that supports both Twitter and Reddit-like interaction modes [37]. Freedit's code comments state that it only retains a user's viewing history for three days [36].

We write two Paralegal policies. The first requires that when page view data is stored, a timestamp for its expiration date is stored alongside it. The second requires that there exists code that checks the database for expired page view data and deletes it. We marked the identifier for the table that stores pageview data for a user. We also mark the library functions that insert and delete data from the data store, as well as a library function for getting the current time.

(4) Hyperswitch is a payment router that provides a unified interface for interacting with common payment processors. We analyze its `router` crate.

Hyperswitch's UI asks users to opt-in to saving their credit card information for future transactions. We write a policy that mandates the user's selection determines (via control flow) whether the credit card details are indeed stored. Hyperswitch's documentation states that plain-text API keys are only available at creation. We write a Paralegal policy that states that an API key may only be released once to the user creating it, and that it can only be access through its hash afterwards.

We marked the type identifying credit card details and the function returning the user's storage decision. Additionally, we marked the API key type, the permissible hash function, and the response type for the key's initial creation. We also mark the return values of all endpoints to ensure that we capture all data sent to users. We analyze the the controller that creates API keys and two controllers that handle credit card data.

(5) Lemmy is a federated Reddit-like platform [50]. Rather than providing a single centralized website, anyone can create an instance tailored to their interests and moderation preferences. Users create communities within those instances and post content to them.

Our Lemmy policies focus on its access control rules: *(i)* if a user is banned or deleted from an instance, they may not read nor write data in that instance; and *(ii)* if a user is banned from a community or the community is permanently removed, users may not write data to that community.

We define four authorization check markers: one for an instance ban check, one for an instance deletion check, and two more for the respective community checks. We define `instance` and `community` markers for database accesses pertaining to data relevant to instances or communities respectively. Authorization checks need to happen in 72 HTTP endpoints in Lemmy that perform database reads and writes. Our policy stipulates that respective authorization checks need to take place in each controller where instance or community database accesses take place and that such checks must happen before the access and have a control flow influence. In addition to the bugs we report we also found two more con-

trollers that violate this policy. The controllers in question create new sites or communities. In this special circumstance a limited number of database accesses are performed before the checks to do initial setup of the new site or community. We manually verified that performing these accesses is safe and then marked this limited number of access locations as exceptions from the policy.

(6) mCaptcha is a proof-of-work based CAPTCHA service focused on privacy [54]. Website developers register their sites with mCaptcha and invoke the mCaptcha service API when end-users visit those sites. If a developer deletes their mCaptcha account, mCaptcha must remove all data associated with it. We realize this with a policy similar to §3's example and mark the `Identity` type from `actix_identity` as well as the `delete_user` method.

mCaptcha's PoW based algorithm features a tunable "difficulty factor", designed to balance security and accessibility to legitimate users. To optimize this parameter, mCaptcha can share statistics with other mCaptcha installations. Publishing this data requires explicit developer opt-in [40]. We initially wrote a policy to enforce that performance data could only be collected from websites that opted in. This verify-before-collect policy failed, and after discussing the issue mCaptcha developers, it turned out that we misinterpreted their privacy goals: statistics are always collected, even if developers don't opt into sharing them. A corrected verify-before-sharing policy passes Paralegal. However, the discussion helped the mCaptcha developers find a (related) data integrity bug: they deleted statistics when a user revoked their opt-in consent, even though they didn't intend to [53]. This illustrates that Paralegal-induced discussions can be helpful to developers.

(7) Plume. Plume is a federated blogging service [60], and the basis of our example in §3. If a user deletes their Plume account, the application must delete their personally identifiable data. We formalize this policy with Paralegal by marking the user and the types of user data: `Comment`, `Blog`, `Post`, `Media`, and `Notification`. Additionally, we placed a marker on the `delete` function in the `diesel` database ORM, for a total of seven markers.

(8) WebSubmit [3] is a homework submission system deployed at a U.S. university and written in 1.6k LoC of Rust using Rocket.rs. We consider three policies: *(i) data deletion*, which tests compliance with a GDPR-style "right to be forgotten" by ensuring that an endpoint for deleting all of a user's data exists; *(ii) scoped storage*, which ensures that the user's identity is stored alongside their data; and *(iii) authorized disclosure*, which encodes the access-control policy: students may view their own answers, TAs and instructors may view all students' answers, and instructors may view course feedback.

We marked the data type containing student answers, deletion functions, the return value of each controller (to cover externalizing data), user identifiers provided by the framework as well as functions that retrieve instructors and TA's from the config.

The analysis covers the endpoint that stores student submitted answers and the deletion controller.

A.1 Source Code Changes

The following table lists, for each application, how many no-op functions we had to introduce to attach markers and how often they needed to be called.

Application	# Marker Functions (calls)
Contile	2 (5)
Freedit	1 (4)
mCaptcha	1 (1)
Hyperswitch	1 (1)
WebSubmit	2 (2)

In addition we had to make the following adjustments to analyze the code bases:

- **Contile**: We inlined one call to `Result::map_err`. This is a small function critical to the policy, but because it is in the (precompiled) standard library our multi-crate analysis could not access its code.
- **Hyperswitch**: We changed the type `PlaintextApiKey` to use an explicit `prefix` and `key` fields instead of a single string so we can attach a marker to the latter field.
- **Lemmy** Applied a flow model to `actix::web::block` the closure it receives can be analyzed without having to deal with the `unsafe` code of `block`.
- **mCaptcha** Stubbed compile time generated code by the "cachebust" utility as it failed to run on our machine and is irrelevant to the policy.

B Policy DSL Grammar

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

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

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

⟨*scope*⟩ ::= `Everywhere:` | `Somewhere:` | `In` ⟨*controller*⟩`:`

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

⟨*bullet*⟩ ::= ⟨*number*⟩. | ⟨*number*⟩) | ⟨*letter*⟩. | ⟨*letter*⟩)

⟨*operator*⟩ ::= `and` | `or`

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

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

⟨*for each*⟩ ::= ⟨*bullet*⟩ `For each` ⟨*variable intro*⟩`:`

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

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

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

⟨*body*⟩ ::= ⟨*bullet*⟩ ⟨*relation*⟩ ⟨*operator*⟩ ⟨*body*⟩
| ⟨*conditional*⟩
| ⟨*bullet*⟩ ⟨*relation*⟩

⟨*conditional*⟩ ::= ⟨*bullet*⟩ `If` ⟨*relation*⟩ `then:` ⟨*clause body*⟩

⟨*only via relations*⟩ ::= ⟨*only via relation*⟩ | ⟨*only via relation*⟩
⟨*operator*⟩ ⟨*only via relations*⟩

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

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