BROWN

Paralegal: Practical Static Analysis for Privacy Bugs

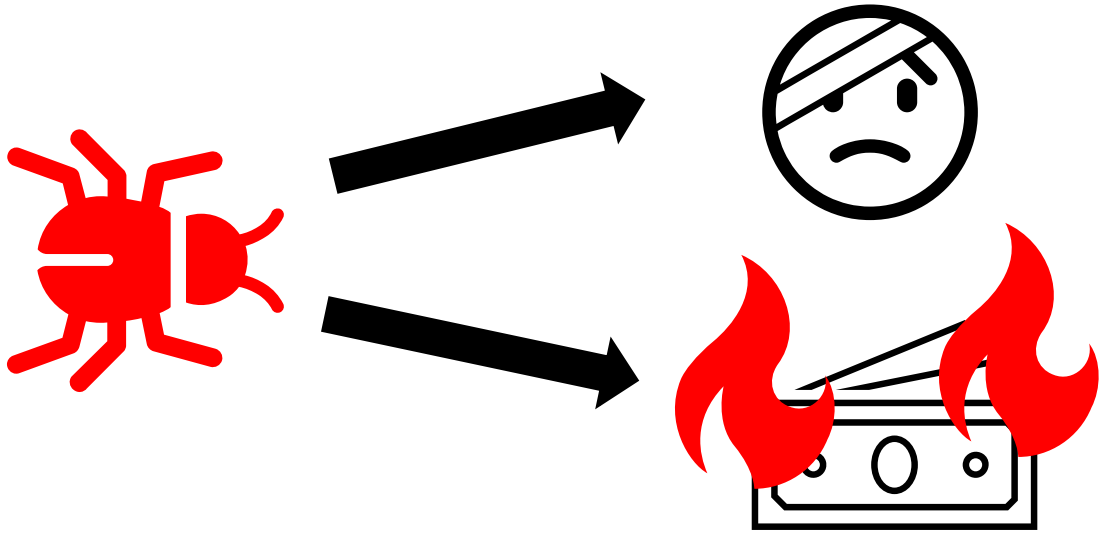
Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Nathan Harbison,

Mithi Jethwa, Will Crichton, Shriram Krishnamurthi, Malte Schwarzkopf

Brown University

Privacy Bugs are Dangerous

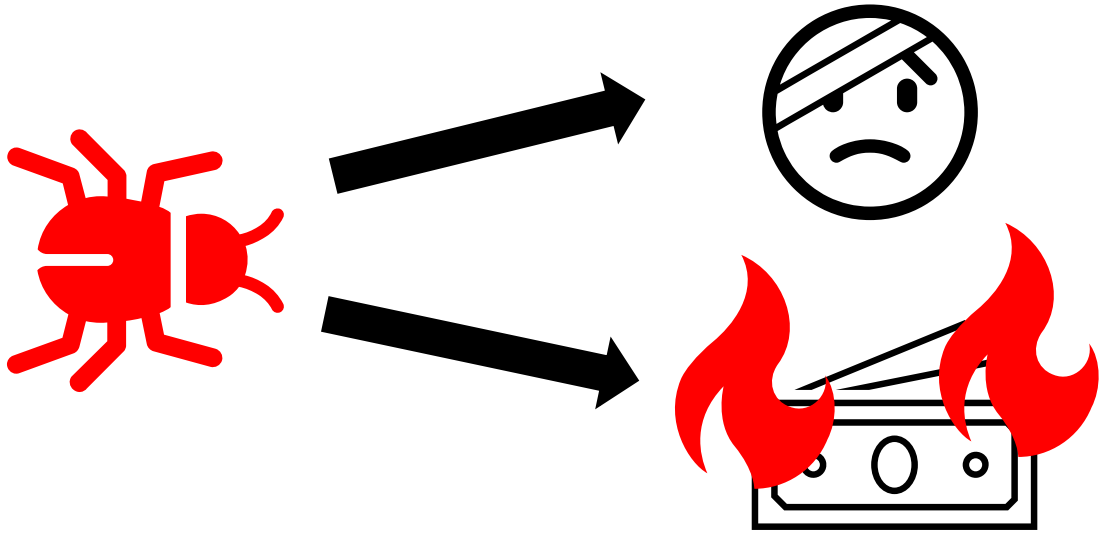
- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation



Privacy Bugs are Dangerous

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Plume



A Data Deletion Bug in Plume

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

```
struct Post { ... }
+
```

Plume

Real Bug; happened twice!

```
// Forgot to delete Comments
```

Developers Need Practical Tools to Find Privacy Bugs

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Plume

```
  struct Post { ... }
+ struct Comment { ... }

  fn delete_row(db, id, table) { ... }

  fn delete_user(user, db) {
    for post in user.get_posts(db) {
      delete_row(db, post.id, "posts");
    }

    // Forgot to delete Comments

    delete_row(db, user.id, "users");
  }
```

Real Bug; happened twice!

Developers Need Practical Tools to Find Privacy Bugs

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Plume

Real Bug; happened twice!

```
  struct Post { ... }
+ struct Comment { ... }

  fn delete_row(db, id, table) { ... }


  fn delete_user(user, db) {
    for post in user.get_posts(db) {
      delete_row(db, post.id, "posts");
    }

    // Forgot to delete Comments

    delete_row(db, user.id, "users");
  }
```

Developers Need Practical Tools to Find Privacy Bugs

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitatio

Tooling must:

- Support relevant
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Could existing tools find subtle privacy bugs in real systems?

```
struct Post { ... }

struct Comment { ... }

                                    e) { ... }

                                  posts(db) {
  delete_row(db, post.id, "posts");
}

// Forgot to delete Comments

delete_row(db, user.id, "users");
}
```

Plume

Real Bug; happened twice!

Does Information Flow Control Help?

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Existing Approaches

- **Information Flow Control**

IFC Is Invasive And Lacks Expressiveness

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Existing Approaches

- **Information Flow Control**

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Use Property-Specific Bug Finders/Fuzzers?

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Existing Approaches

- Information Flow Control

- **Fixed Property Fuzzers / Bug Finders**

Developing Per-property Tools is Infeasible

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Existing Approaches

- Information Flow Control

- **Fixed Property Fuzzers / Bug Finders**

Automatically Add Runtime Checks?

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Existing Approaches

- Information Flow Control

- Fixed Property Fuzzers / Bug Finders

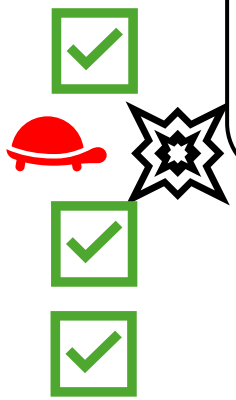
- **Runtime Enforcement**

Disruptive Runtime Checks Add Overhead

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)



Existing Approaches

- Information Flow Control

- Fixed Property Fuzzers / Bug Finders

- **Runtime Enforcement**

Use a Code Query Engine to Find Bugs?

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

Existing Approaches

- Information Flow Control

- Fixed Property Fuzzers / Bug Finders

- Runtime Enforcement

- **Code Query Engine**

CodeQL 

Approximate Data Deletion Policy in CodeQL

```
import cpp
import semmle.code.cpp.dataflow.new.DataFlow
import semmle.code.cpp.dataflow.new.TaintTracking
import semmle.code.cpp.controlflow.ControlFlowGraph
import semmle.code.cpp.ir.IR

predicate is_user_data(Type outer) {
  exists(Type t |
    is_user_data_direct(t) and
    outer.refersTo(t)
  )
}

predicate is_user_data_direct(Type t) {
  t.getName() = "Comment" or
  t.getName() = "Post"
}

predicate is_db_method(Function f) {
  f.getName().regexpMatch("get_posts|get_comments")
}

module SourceSinkCallConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) { is_user_data(source.getType()) }

  predicate isSink(DataFlow::Node sink) { is_delete_2(sink) }
}

predicate is_root(Function n) { n.getName() = "delete_user" }

predicate reachable_from_root(ControlFlowNode n) {
  reachable(n) and
  (
    is_root(n.getControlFlowScope())
    or
    exists(ControlFlowNode caller |
      caller.(FunctionCall).getTarget() = n.getControlFlowScope() and
      reachable_from_root(caller)
    )
  )
}
```

```
module Taint = TaintTracking::Global<SourceSinkCallConfig>;

predicate is_delete_2(DataFlow::Node e) {
  exists(Call c, Expr arg |
    c.getTarget().getName() = "delete_row" and
    c.getArgument(0) = arg and
    (e.asExpr() = arg or e.asIndirectExpr() = arg)
  )
}

from Type user_data, FunctionCall f, DataFlow::Node source, DataFlow::Node sink,
Location sink_loc
where
  is_user_data_direct(user_data) and
  source.getEnclosingCallable().getName() = "delete_user" and
  source.asExpr() = f and
  source.getType().refersTo(user_data) and
  is_db_method(f.getTarget()) and
  Taint::flow(source, sink) and
  is_delete_2(sink) and
  is_reachable_from_root(sink.asExpr()) and
  sink.getLocation() = sink_loc
select user_data, source, sink, sink_loc.getFile().getBaseName(),
sink_loc.getStartLine()
```

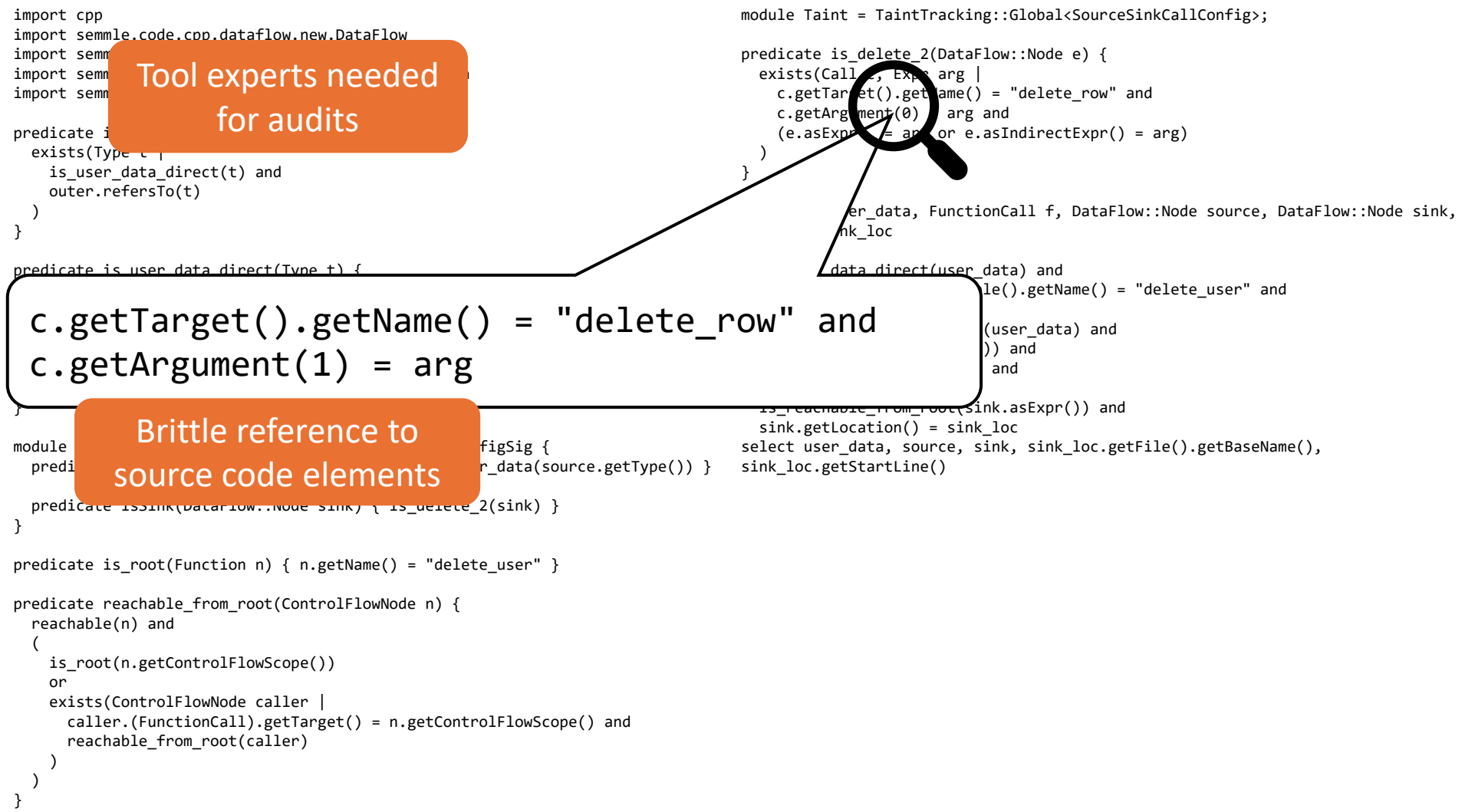
Audits Require CodeQL Experts

Tool experts needed for audits

```
import cpp
import semmle.code.cpp.dataflow.new.DataFlow
import semm
import semm
import semm

predicate i
  exists(Type t |
    is_user_data_direct(t) and
    outer.refersTo(t)
  )
}

predicate is_user_data_direct(Type t) {
  t.getName() = "Comment" or
  t.getName() = "Post"
}

predicate is_db_method(Function f) {
  f.getName().regexpMatch("get_posts|get_comments")
}

module SourceSinkCallConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) { is_user_data(source.getType()) }

  predicate isSink(DataFlow::Node sink) { is_delete_2(sink) }
}

predicate is_root(Function n) { n.getName() = "delete_user" }

predicate reachable_from_root(ControlFlowNode n) {
  reachable(n) and
  (
    is_root(n.getControlFlowScope())
    or
    exists(ControlFlowNode caller |
      caller.(FunctionCall).getTarget() = n.getControlFlowScope() and
      reachable_from_root(caller)
    )
  )
}
```

```
module Taint = TaintTracking::Global<SourceSinkCallConfig>;

predicate is_delete_2(DataFlow::Node e) {
  exists(Call c, Expr arg |
    c.getTarget().getName() = "delete_row" and
    c.getArgument(0) = arg and
    (e.asExpr() = arg or e.asIndirectExpr() = arg)
  )
}

from Type user_data, FunctionCall f, DataFlow::Node source, DataFlow::Node sink,
Location sink_loc
where
  is_user_data_direct(user_data) and
  source.getEnclosingCallable().getName() = "delete_user" and
  source.asExpr() = f and
  source.getType().refersTo(user_data) and
  is_db_method(f.getTarget()) and
  Taint::flow(source, sink) and
  is_delete_2(sink) and
  is_reachable_from_root(sink.asExpr()) and
  sink.getLocation() = sink_loc
select user_data, source, sink, sink_loc.getFile().getBaseName(),
sink_loc.getStartLine()
```

Brittle References to Changing Source Code



Manual Modeling for Libraries Requried

```
import cpp
import semmle.code.cpp.dataflow.new.DataFlow
import semm
import semm
import semm

predicate i
  exists(Type t |
    is_user_data_direct(t) and
    outer.refersTo(t)
  )
}

predicate is_user_data_direct(Type t) {
```

```
module
  predi
  predicate isSink(DataFlow::Node sink) { is_delete_2(sink) }
}

predicate is_root(Function n) { n.getName() = "delete_user" }

predicate reachable_from_root(ControlFlowNode n) {
  reachable(n) and
  (
    is_root(n.getControlFlowScope())
    or
    exists(ControlFlowNode caller |
      caller.(FunctionCall).getTarget() = n.getControlFlowScope() and
      reachable_from_root(caller)
    )
  )
}
```

```
module Taint = TaintTracking::Global<SourceSinkCallConfig>;

predicate is_delete_2(DataFlow::Node e) {
  exists(Call c, Expr arg |
    c.getTarget().getName() = "delete_row" and
    c.getArgument(0) = arg and
    (e.asExpr() = arg or e.asIndirectExpr() = arg)
  )
}
```

```
er_data, FunctionCall f, DataFlow::Node source, DataFlow::Node sink,
nk_loc

data_direct(user_data) and
le().getName() = "delete_user" and

(user_data) and
)) and
and

is_reachable_from_root(sink.asExpr()) and
sink.getLocation() = sink_loc
select user_data, source, sink, sink_loc.getFile().getBaseName(),
sink_loc.getStartLine()
```

Tool experts needed for audits

```
c.getTarget().getName() = "delete_row" and
c.getArgument(1) = arg
```

Brittle reference to source code elements

Thousands of lines modeling stdlib

Manual models needed for libraries

This Approach is not Practical

```
import cpp
import semmle.code.cpp.dataflow.new.DataFlow
import semm
import semm
import semm

predicate i
  exists(Type t |
    is_user_data_direct(t) and
    outer.refersTo(t)
  )
}

predicate is_user_data_direct(Type t) {
```

Tool experts needed for audits

```
c.getTarget().getName() = "delete_row" and
c.getArgument(1) = arg
```

Brittle reference to source code elements

```
module
  predi                                    figSig {
                                           r_data(source.getType()) }

  predicate isSink(DataFlow::Node sink) { is_delete_2(sink) }
}

predicate is_root(Function n) { n.getName() = "delete_user" }

from UserData user_data, FunctionCall f, DataFlow::Node source, DataFlow::Node sink,
  exists(ControlFlowNode caller |
    caller.(FunctionCall).getTarget() = n.getControlFlowScope() and
    reachable_from_root(caller)
  )
)
}
```

Code Query Engines

```
module Taint = TaintTracking::Global<SourceSinkCallConfig>;

predicate is_delete_2(DataFlow::Node e) {
  exists(Call c, Expr arg |
    c.getTarget().getName() = "delete_row" and
    c.getArgument(0) = arg and
    (e.asExpr() = arg or e.asIndirectExpr() = arg)
  )
}

er_data, FunctionCall f, DataFlow::Node source, DataFlow::Node sink,
nk_loc

data_direct(user_data) and
le().getName() = "delete_user" and

(user_data) and
)) and
and

is_reachable_from_root(sink.asExpr()) and
sink.getLocation() = sink_loc
select user_data, source, sink, sink_loc.getFile().getBaseName(),
sink_loc.getStartLine()
```

Thousands of lines modeling stdlib

Manual models needed for libraries

Developers Need a Better Tool to Find Privacy Bugs

Practical Static Analysis with Paralegal

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

Paralegal

Tooling must:

- Support relevant policies
- Impose minimal changes to code & behavior
- Policies robust to code changes
- Support realistic applications (in scale & library use)

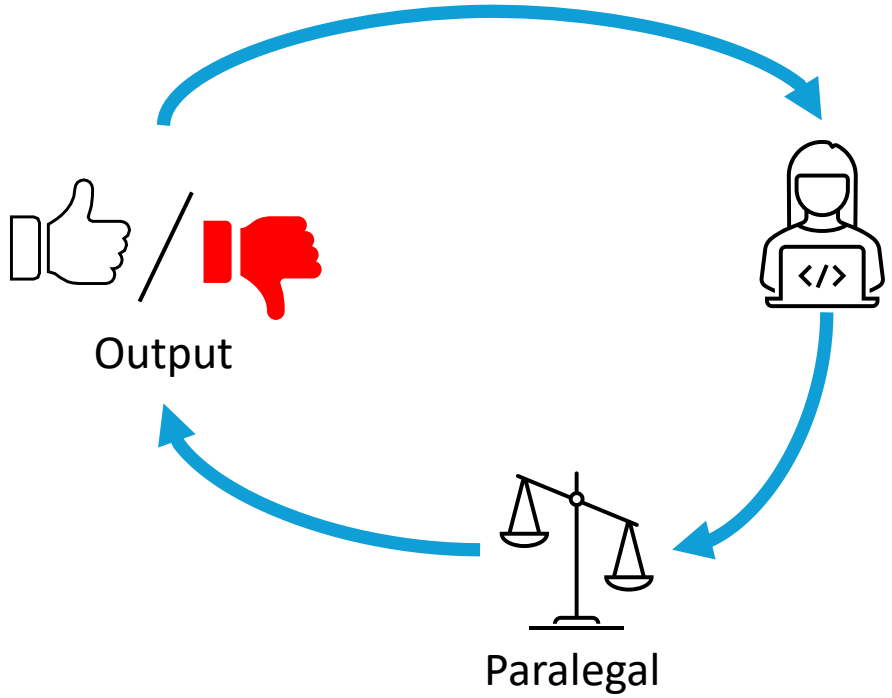
Targets unmodified Rust

Limitations:

- Static information only
- Recall and precision depend on policy

Paralegal Provides Constant, Rapid Feedback

- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation

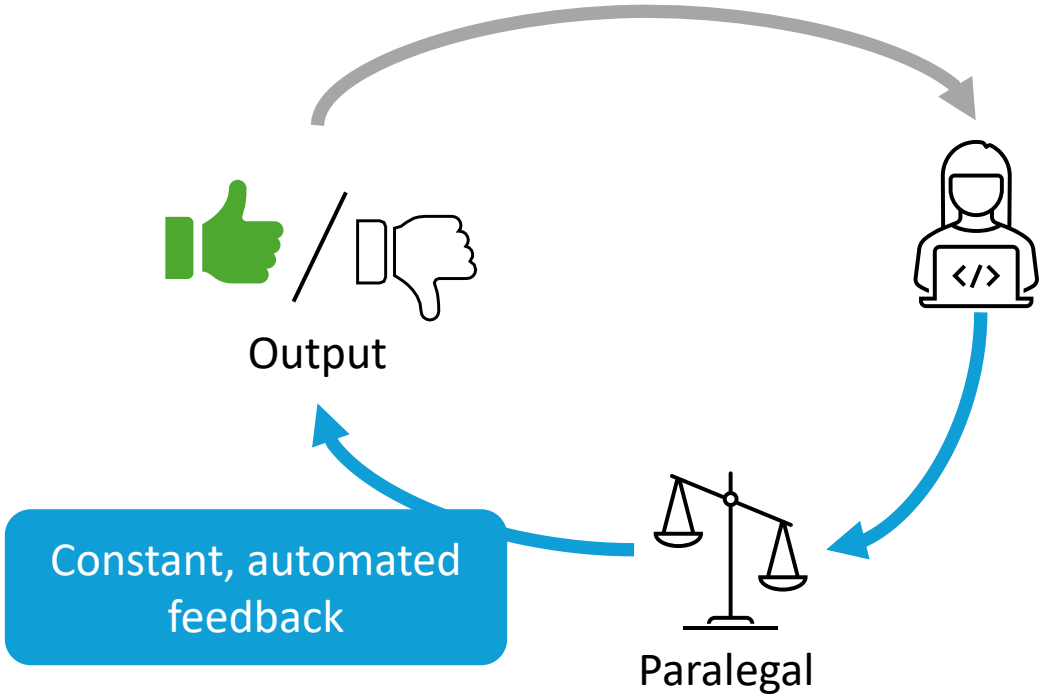


Plume

```
struct Post { ... }

struct Comment { ... }

fn delete_row(db, id, table) { ... }

fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }
+
+   // Forgot to delete Comments
+
  delete_row(db, user.id, "users");
}
```

Paralegal Provides Constant, Rapid Feedback

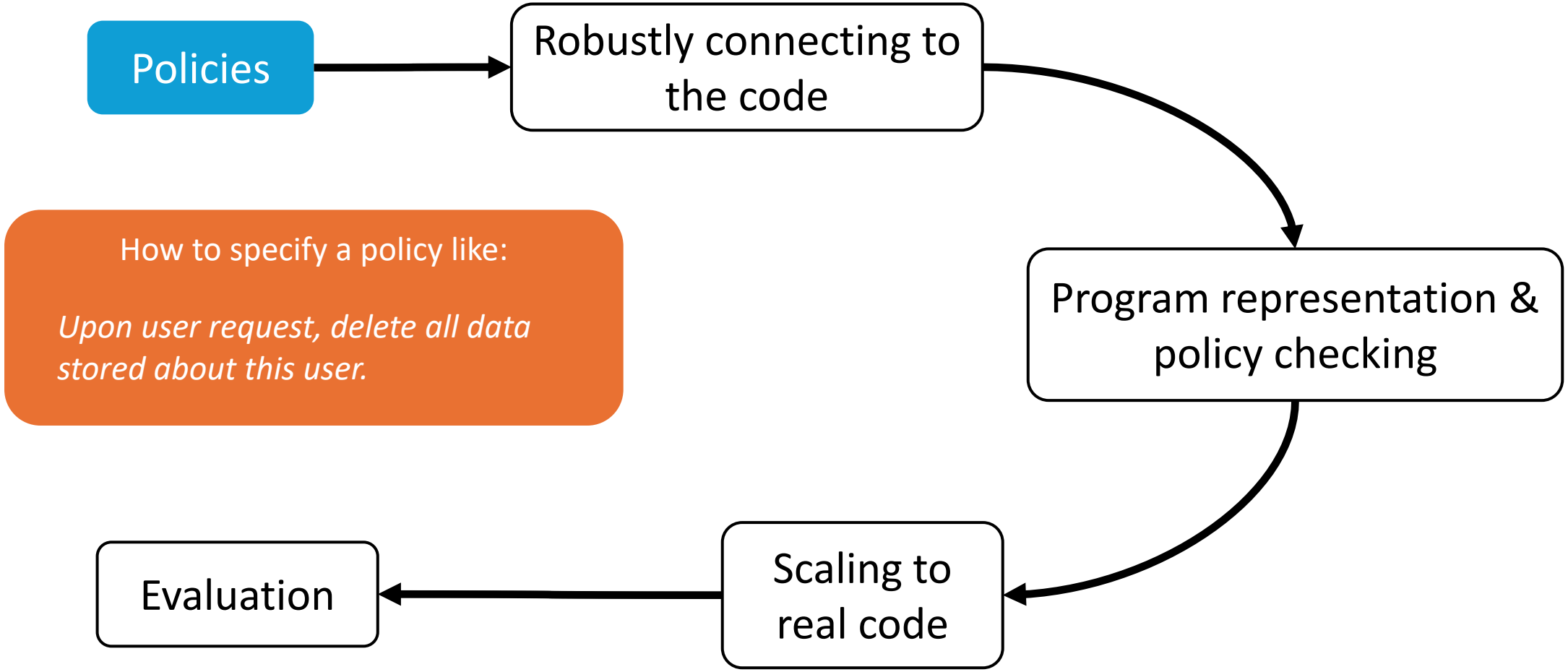
- Access Control
- Encryption-at-rest
- Data Deletion
- Purpose Limitation



Plume

```
struct Post { ... }

struct Comment { ... }

fn delete_row(db, id, table) { ... }

fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }
+ for comment in user.get_comments(db) {
+   delete_row(db, comment.id, "comments");
+ }
  delete_row(db, user.id, "users");
}
```

Roadmap

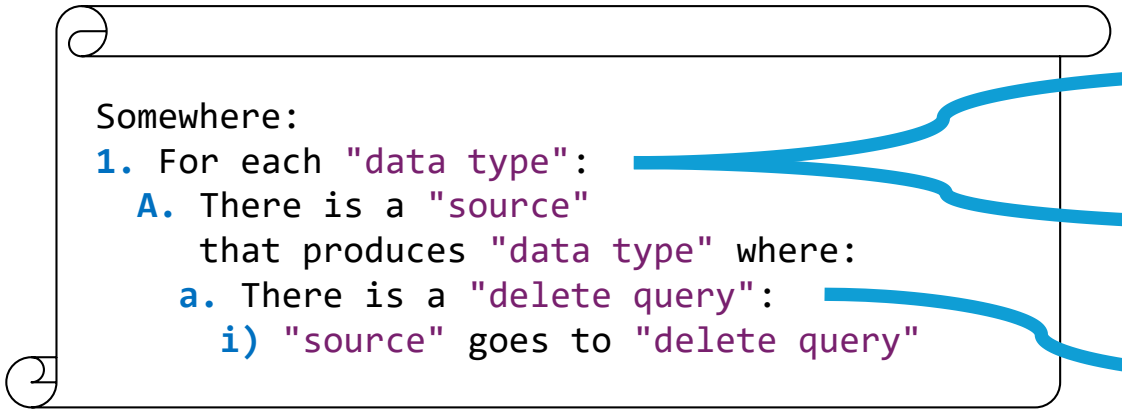

Policies
Robustly connecting to the code
Program representation & policy checking
Scaling to real code
Evaluation
How to specify a policy like:
Upon user request, delete all data stored about this user.

Paralegal Policies are Succinct

```
Somewhere:
1. For each "data type":
  A. There is a "source"
     that produces "data type" where:
    a. There is a "delete query":
      i) "source" goes to "delete query"
```

```
struct Post { ... }

struct Comment { ... }

fn delete_row(db, id, table) { ... }

fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }


  delete_row(db, user.id, "users");
}
```

Paralegal Policies are Succinct




```
struct Post { ... }
```

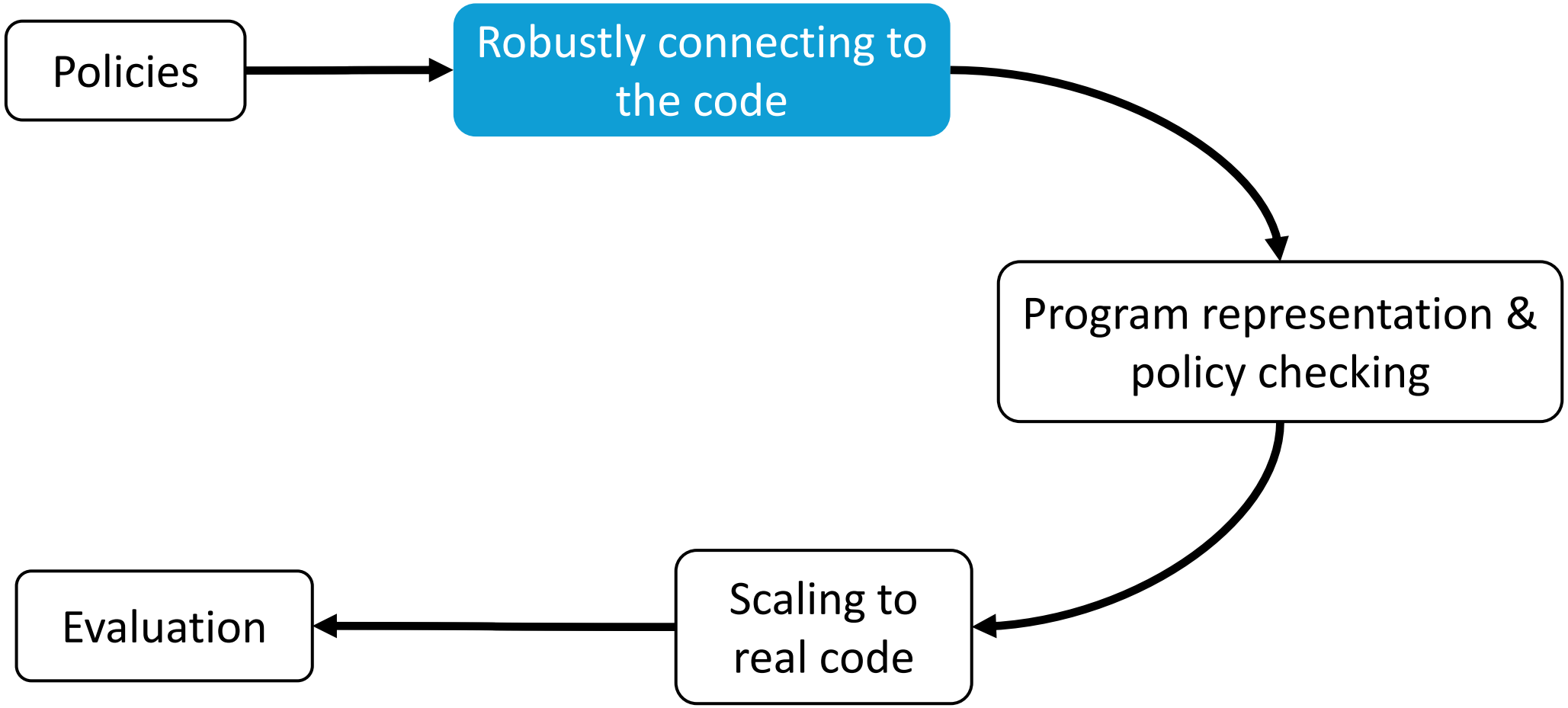
```
struct Comment { ... }
```

```
fn delete_row(db, id, table) { ... }
```

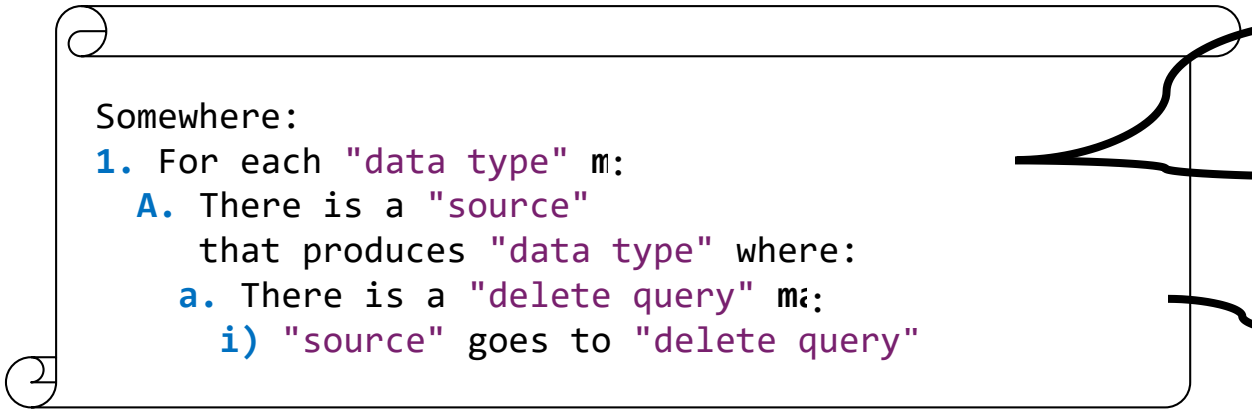
- Formal language
  - Expressive (first-order logic)
  - Unambiguous → predictable
- Natural-language-like → auditable

```
fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }


  delete_row(db, user.id, "users");
}
```

Roadmap


Policies
Robustly connecting to the code
Program representation & policy checking
Scaling to real code
Evaluation

Markers Robustly Link Code and Policy




```
struct Post { ... }

struct Comment { ... }

fn delete_row(db, id, table) { ... }

fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }


  delete_row(db, user.id, "users");
}
```

Markers: Lightweight annotations of semantically meaningful terms.

Markers Robustly Link Code and Policy

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

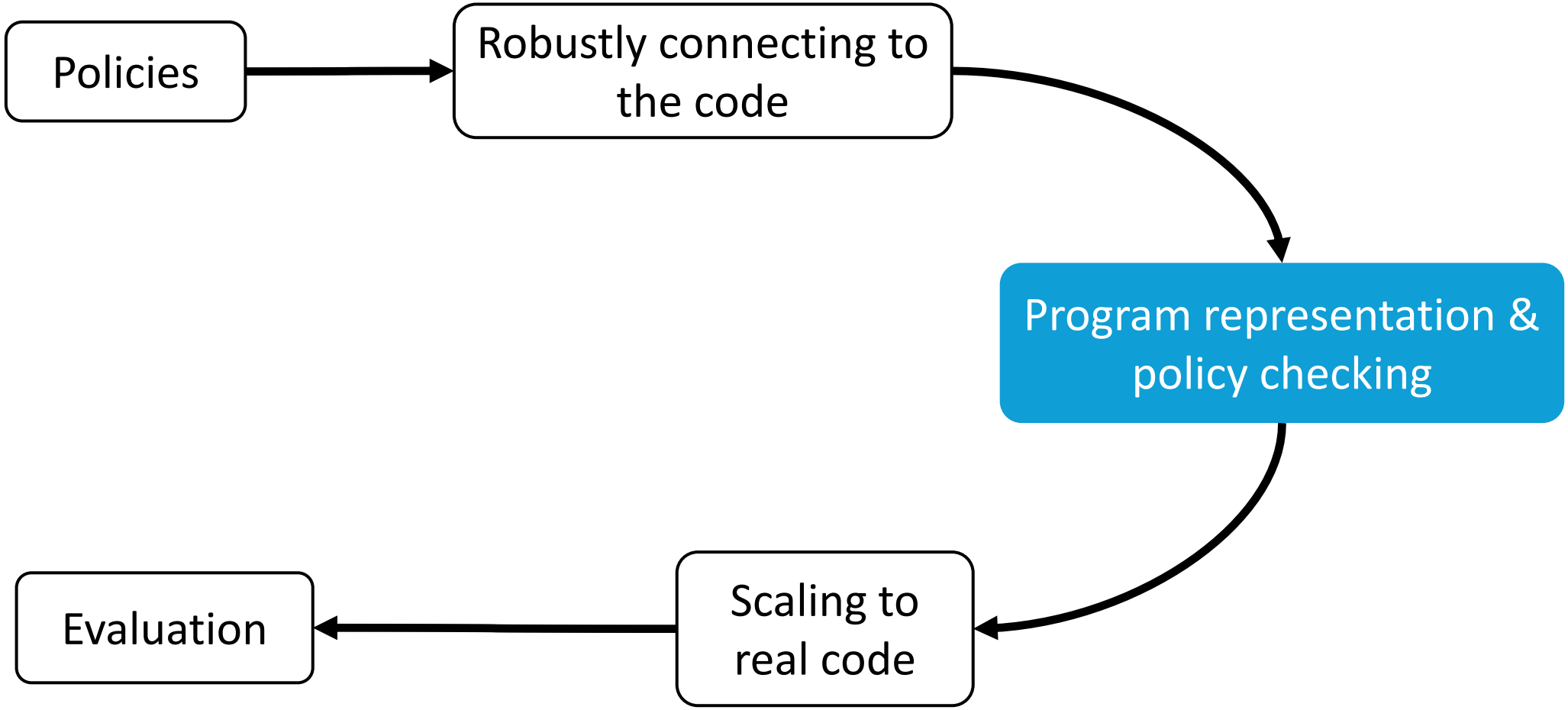
Dataflow dependency

Paralegal

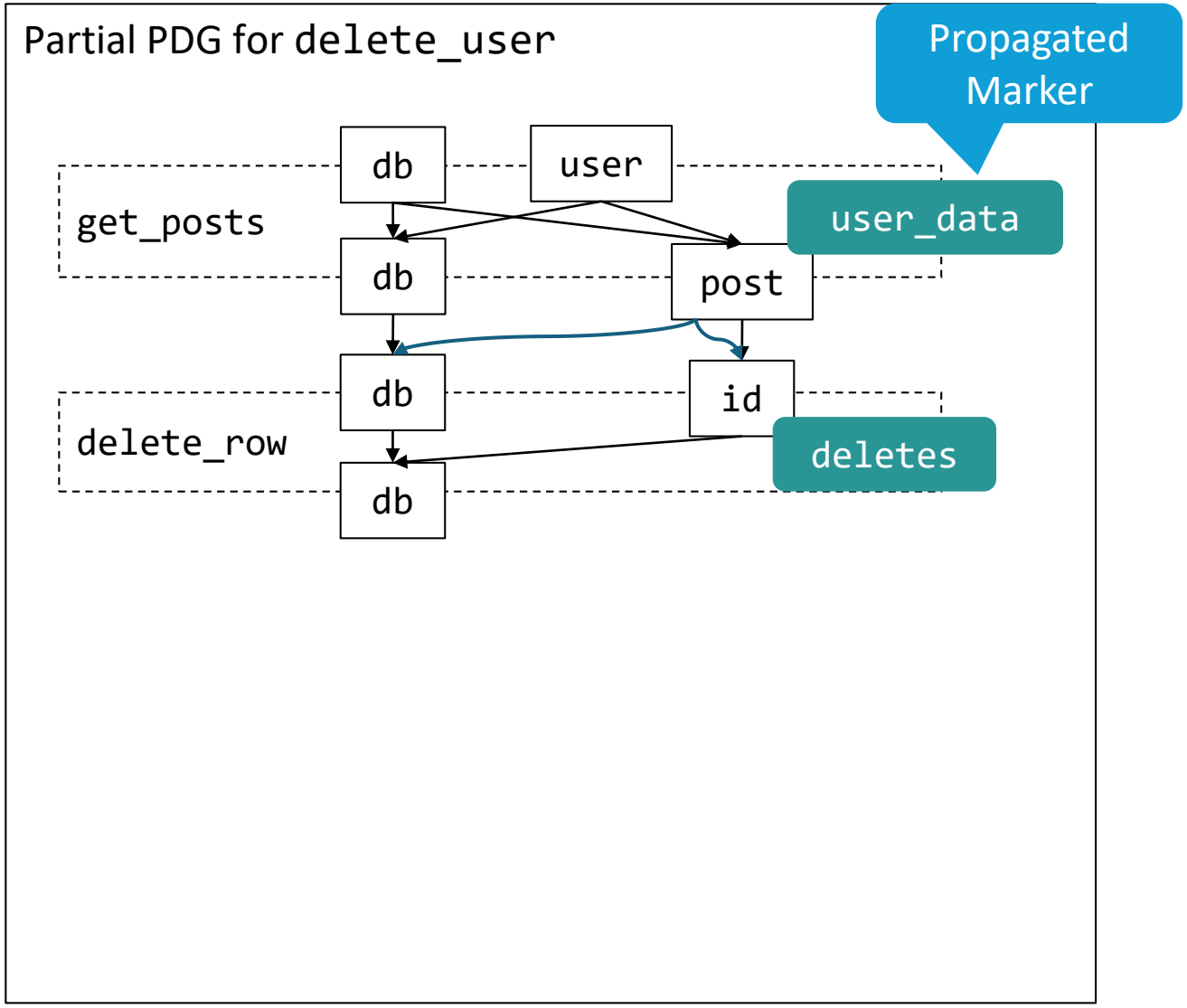
```
'Deletion Policy' not satisfied.
No entrypoints satisfied rule 1.A

Entrypoint `delete_user'
  Did not satisfy Rule 1.A
    There is a "source"
    that produces "data type"

  for the "data type" type: `Comment` (Rule 1)
```

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

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

#[paralegal::marker(deletes, argument=[id])]
fn delete_row(db, id, table) { ... }


fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }



  delete_row(db, user.id, "users");
}
```

Roadmap


Policies
Robustly connecting to the code
Program representation & policy checking
Scaling to real code
Evaluation

PDGs Represent Dependencies



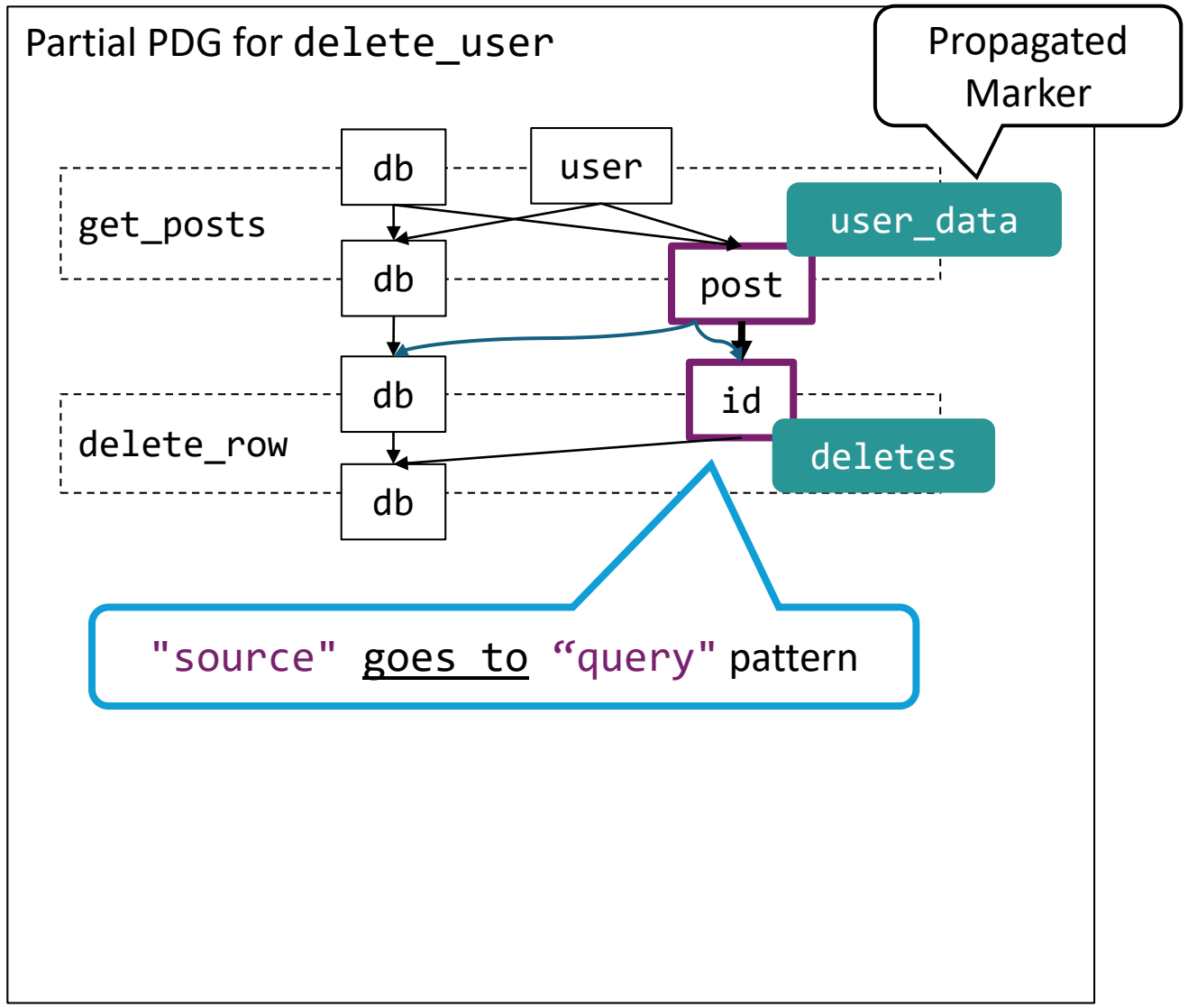
Program **D**ependence **G**raph

Nodes represent variables at a point in the program execution

Edges represent data → and control ⇢ dependencies

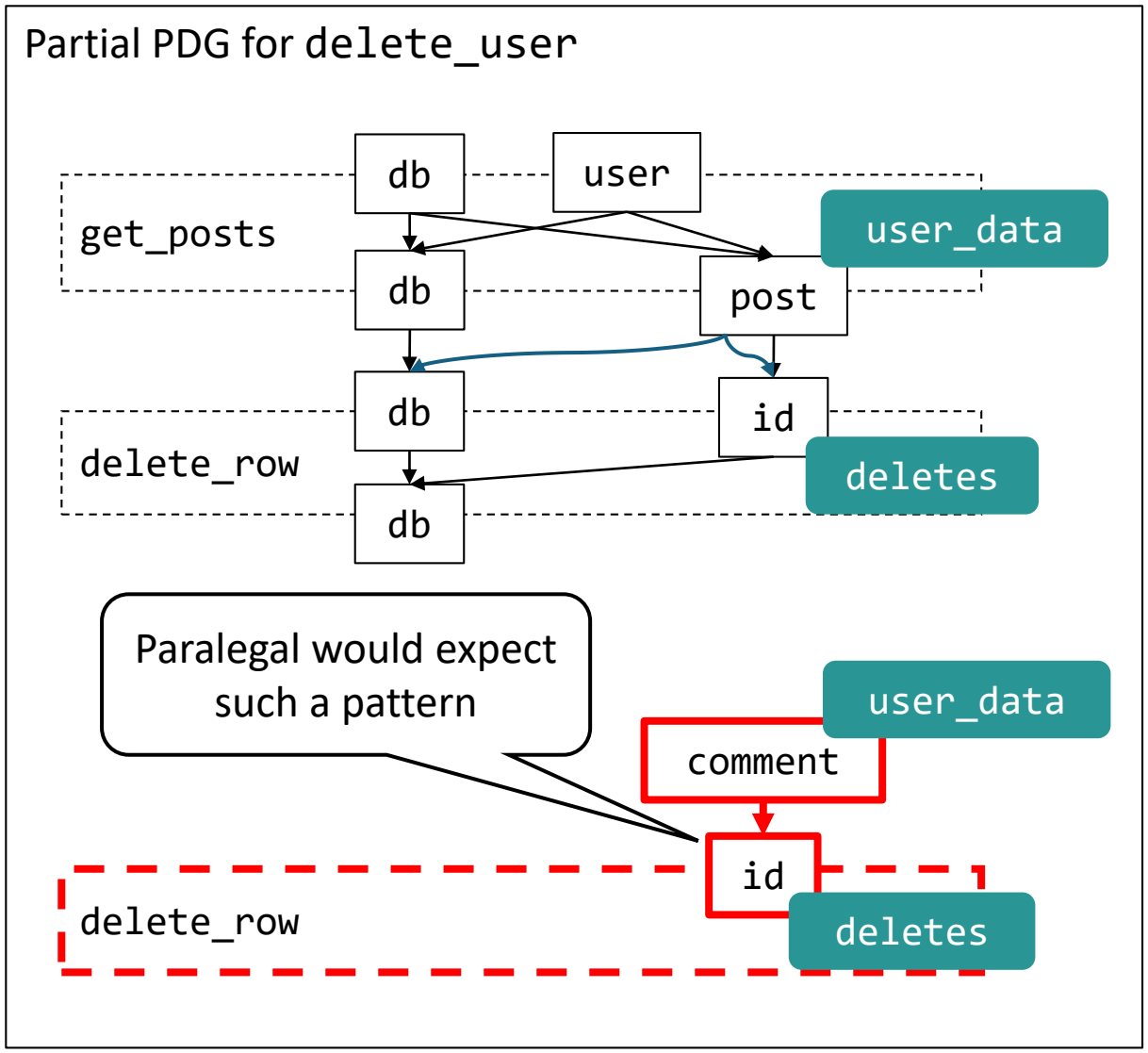
```
fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }



  delete_row(db, user.id, "users");
}
```

Graph Queries Check Policies



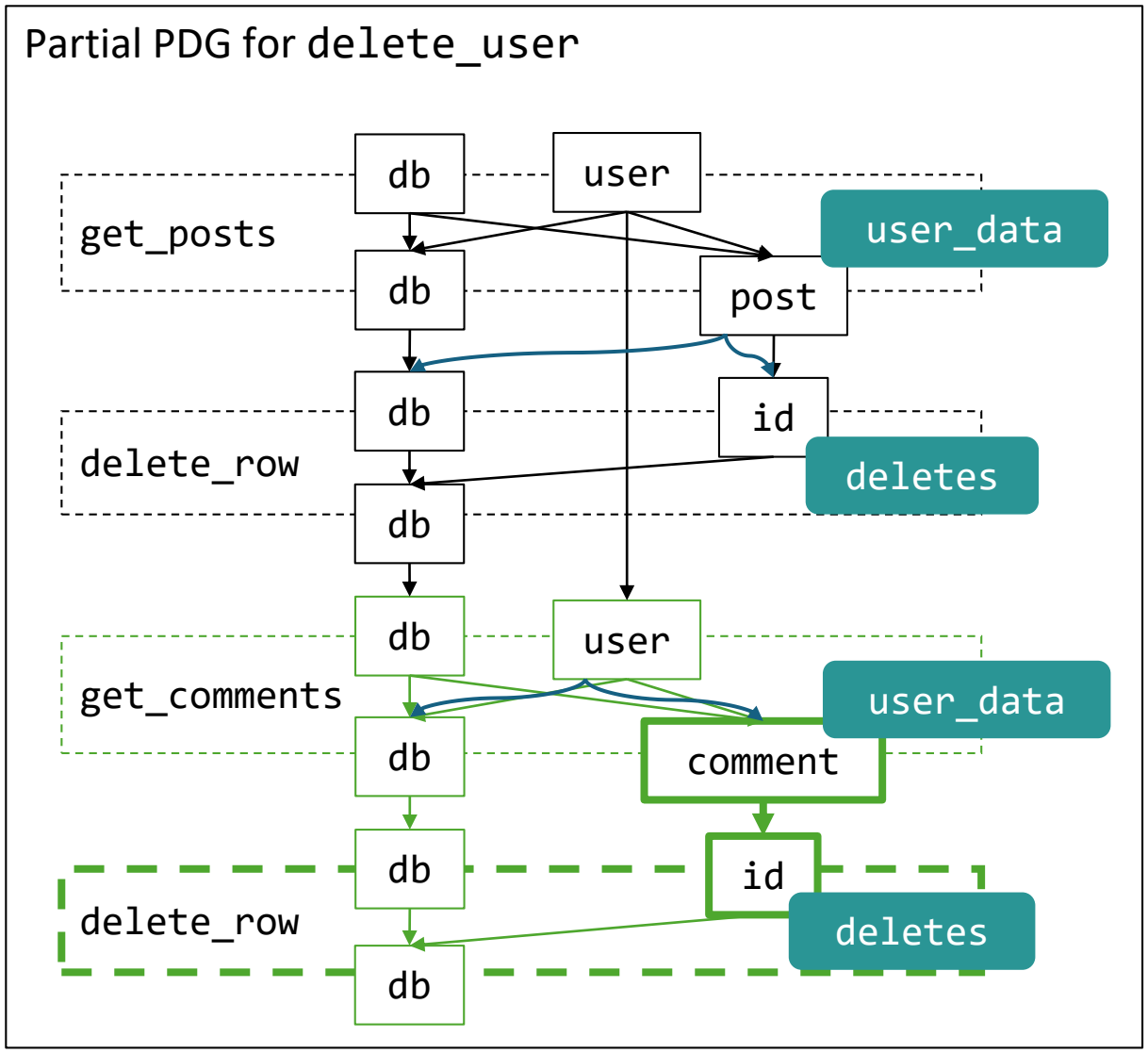

```
#[paralegal::marker(user_data)]
struct Post { ... }

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

#[paralegal::marker(deletes, argument=[id])]
fn delete_row(db, id, table) { ... }


fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }



  delete_row(db, user.id, "users");
}
```

Graph Queries Check Policies



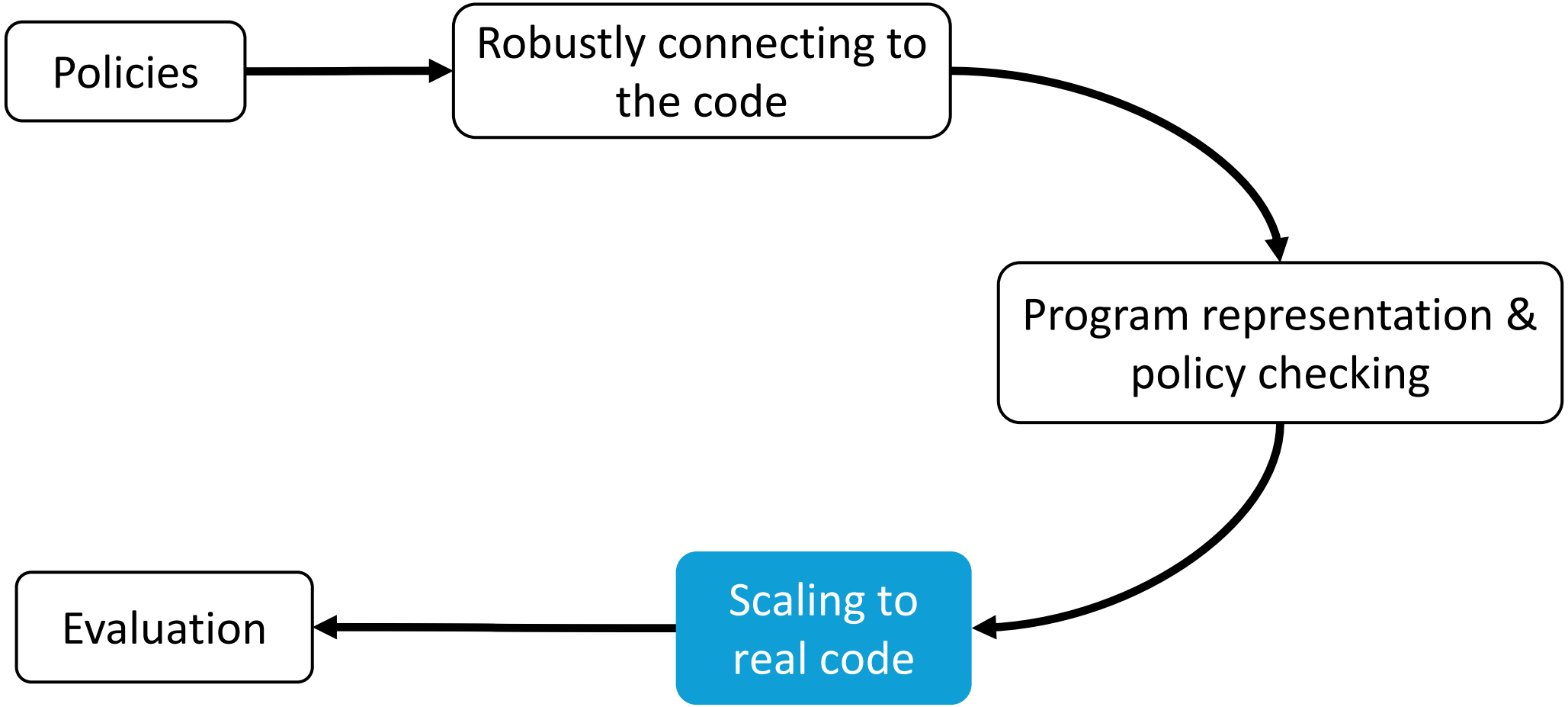

```rust
#[paralegal::marker(user_data)]
struct Post { ... }

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

#[paralegal::marker(deletes, argument=[id])]
fn delete_row(db, id, table) { ... }


fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }



  delete_row(db, user.id, "users");
}
```

Graph Queries Check Policies



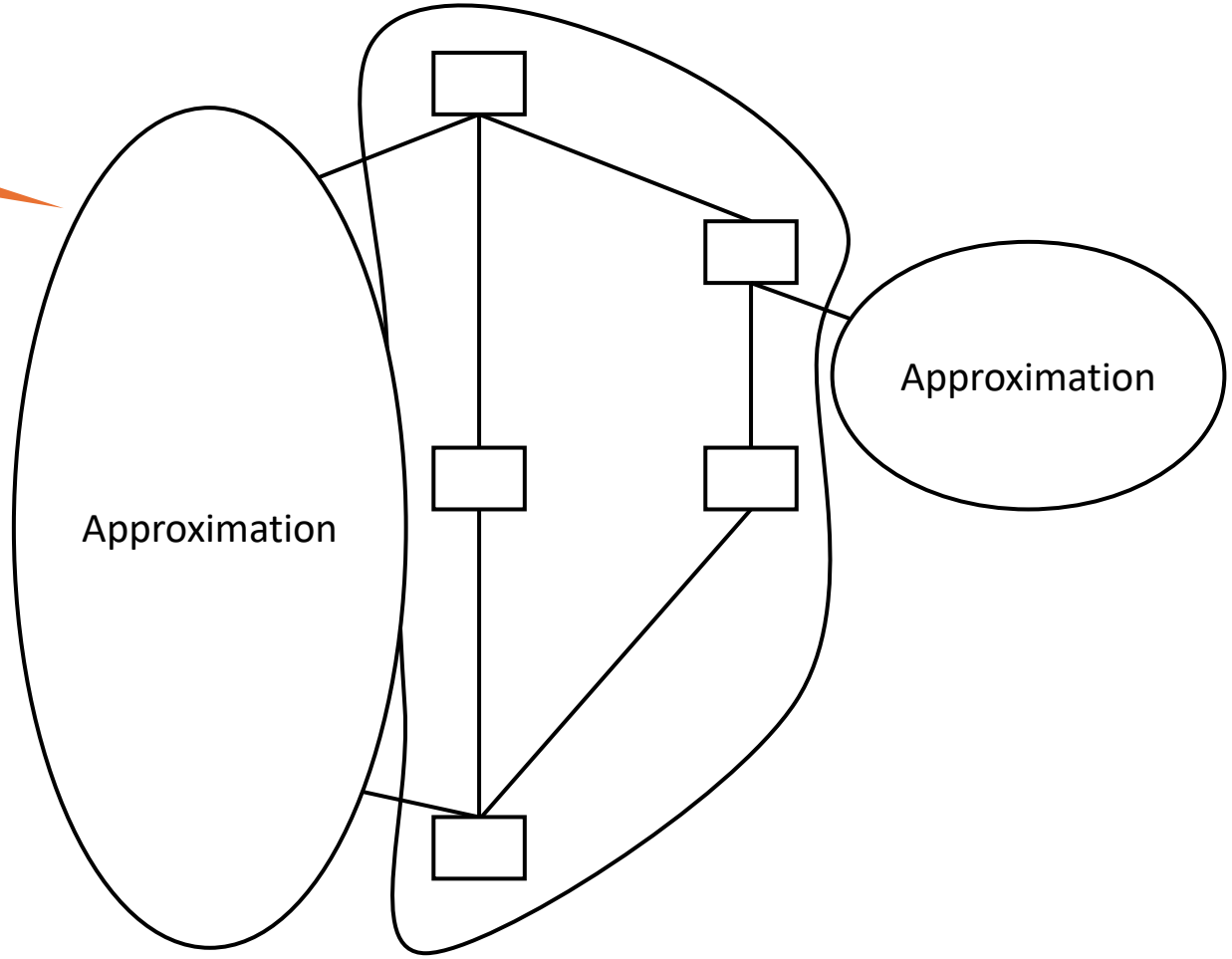

```
#[paralegal::marker(user_data)]
struct Post { ... }

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

#[paralegal::marker(deletes, argument=[id])]
fn delete_row(db, id, table) { ... }


fn delete_user(user, db) {
  for post in user.get_posts(db) {
    delete_row(db, post.id, "posts");
  }
+ for comment in user.get_comments(db) {
+   delete_row(db, comment.id, "comments");
+ }
  delete_row(db, user.id, "users");
}
```

Roadmap

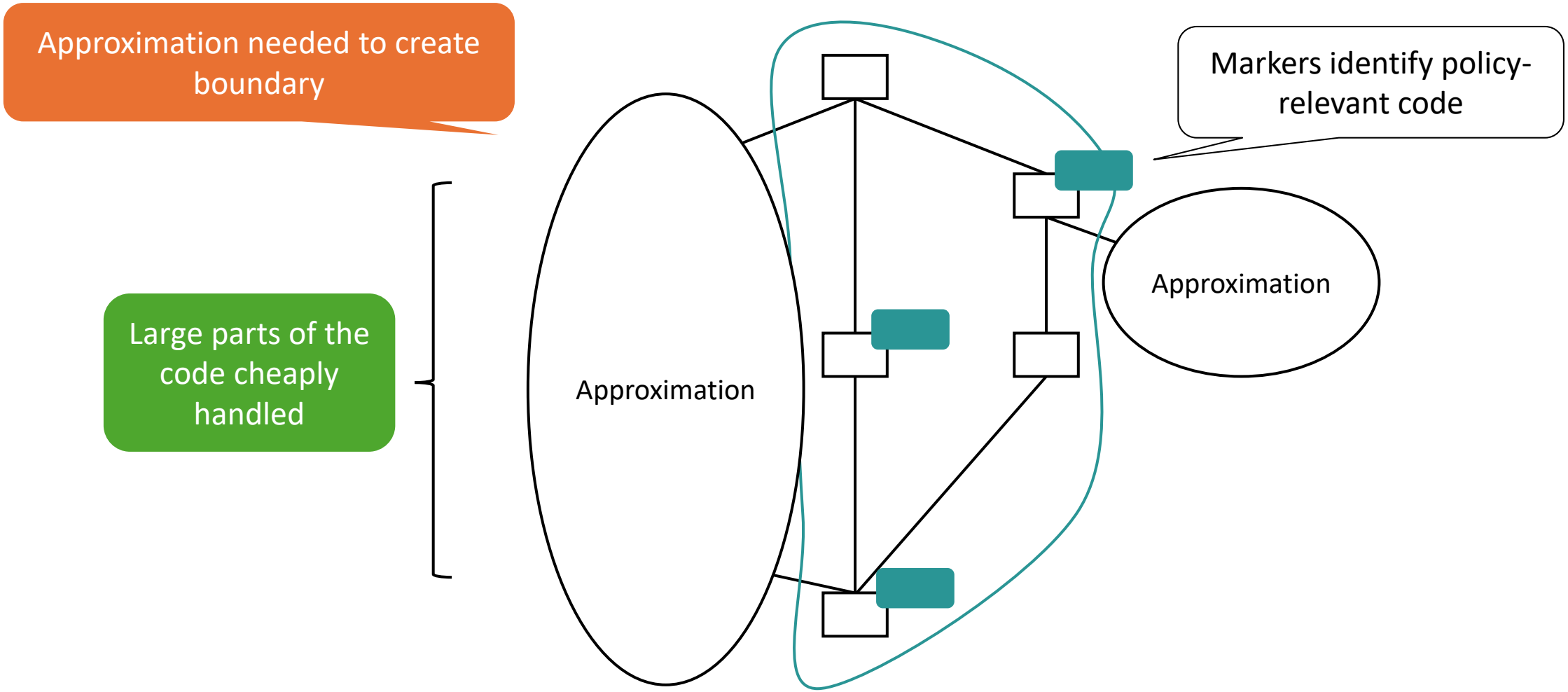

Policies
Robustly connecting to the code
Program representation & policy checking
Scaling to real code
Evaluation

Pruning the Call Tree Helps Scaling

Approximation needed to create boundary



Markers Identify Policy-Relevant Code



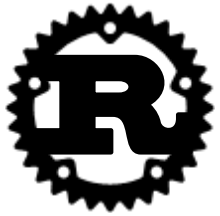
Rust Types Allow Approximating Functions

To approximate the PDG for a function Paralegal needs to know:

- How data flows through the function
- points-to analysis)

So Far: Approximation imprecise. Dependencies missed or many fake ones

Key insight: Leverage Rust types for a **precise** approximation



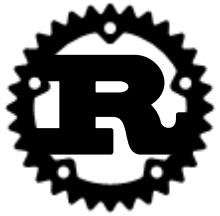
Rust Types Allow Approximating Functions

To approximate the PDG for a function Paralegal needs to know:

- How data flows through the function
- points-to analysis)

So Far: Approximation imprecise.
Dependencies missed or many fake ones

Key insight: Leverage Rust types for a **precise** approximation



Conservative: All passed references are modified

Determination from type signature (**mut**):

- `db` might be mutated
- `table` cannot

```
fn delete_row(db: &mut Database, id: i32, table: &str)
```

Rust Types Allow Approximating Functions

To approximate the PDG for a function
Paralegal needs to know:

- How data flows through th
- function

points-to analysis)

types for a
imation

- Enables Code Pruning Optimization
- Closed-source code
- FFI
- Difficult-to-analyze libraries

Conserva

Type signat
determinati

le cannot

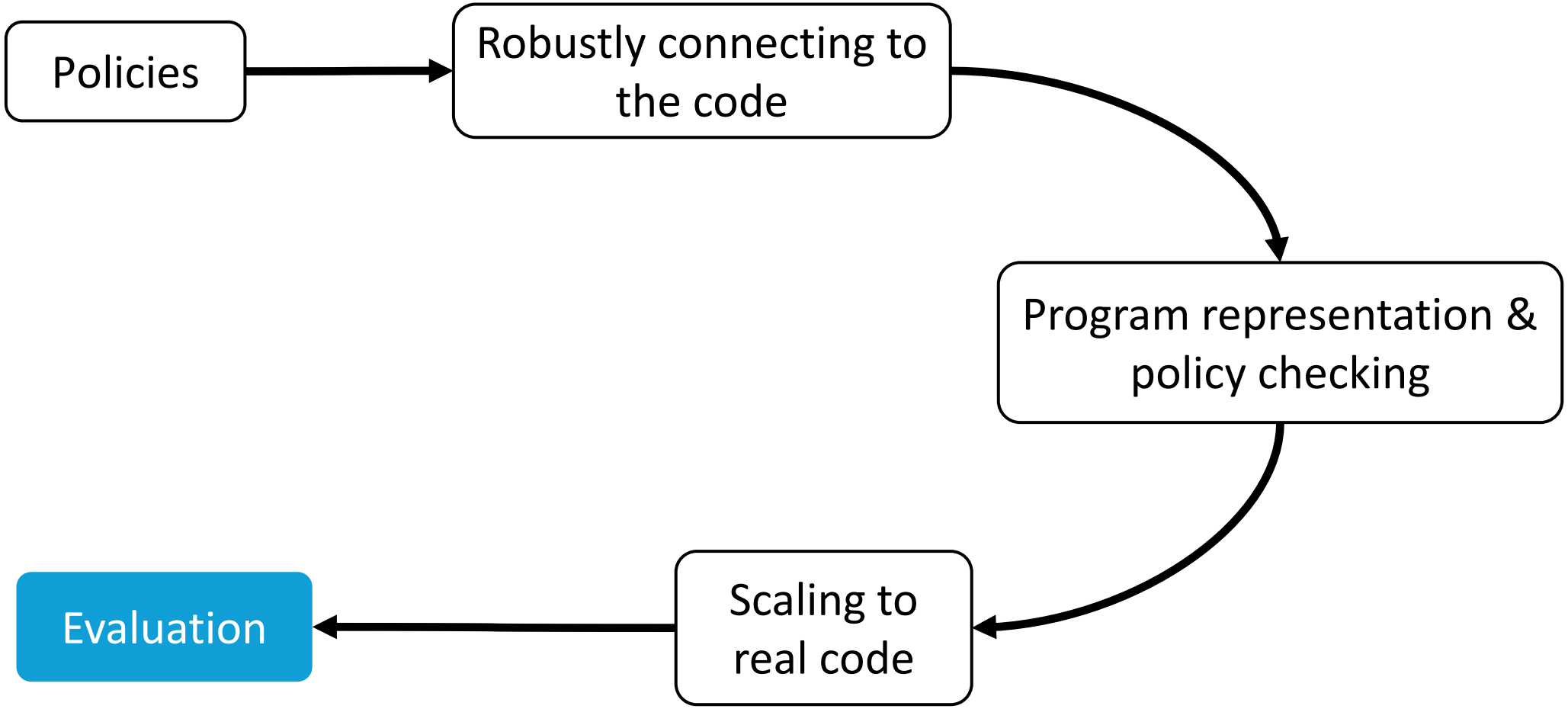
Limitation: Misses dependencies when lifetimes or mutability inaccurate

```
fn delete_row(db: &mut Database, id: i32, table: &str)
```

Implementation

- Paralegal has a prototype (15k LoC)
- Rust compiler plugin, operating on MIR
- Leverages Flowistry (Crichton et al. PLDI 2022)

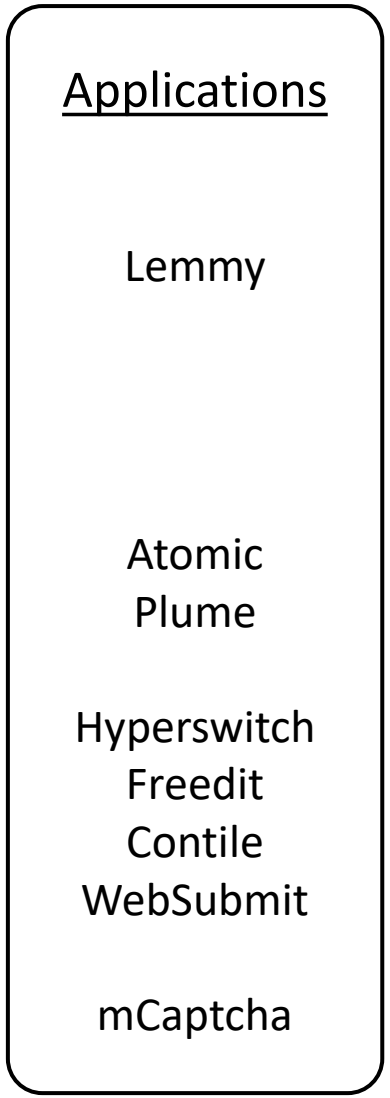
Roadmap


Policies
Robustly connecting to the code
Program representation & policy checking
Scaling to real code
Evaluation

Evaluation

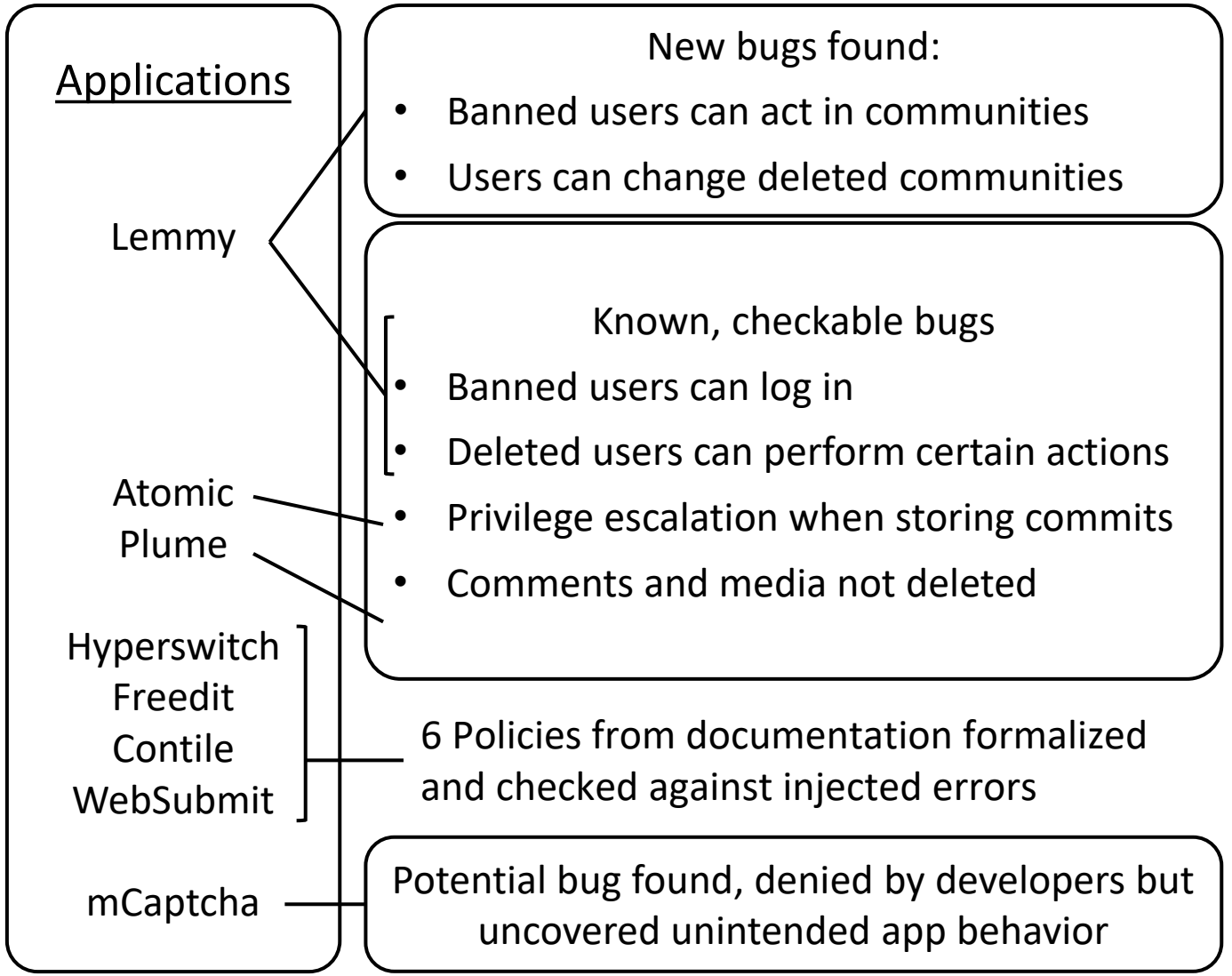
1. Does Paralegal find privacy bugs?

11 Policies formalized



Paralegal Finds Privacy Bugs

1. Does Paralegal find privacy bugs?



11 Policies formalized

Paralegal finds real, semantically subtle bugs

Evaluation

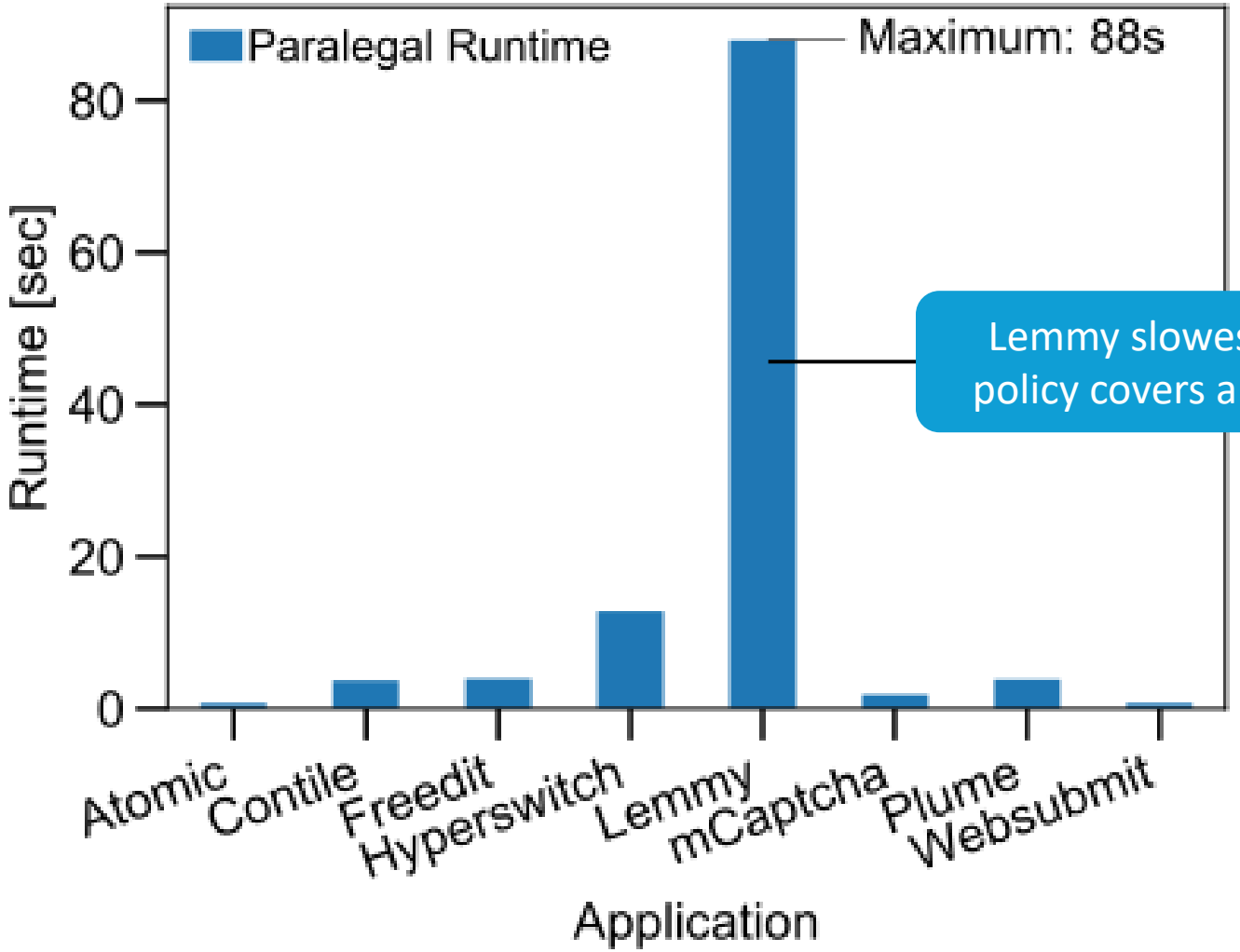
1. Does Paralegal find privacy bugs?
2. Is Paralegal fast enough to be practical?

Is Paralegal Fast Enough?

Setup: All dependencies analyzed (except stdlib)

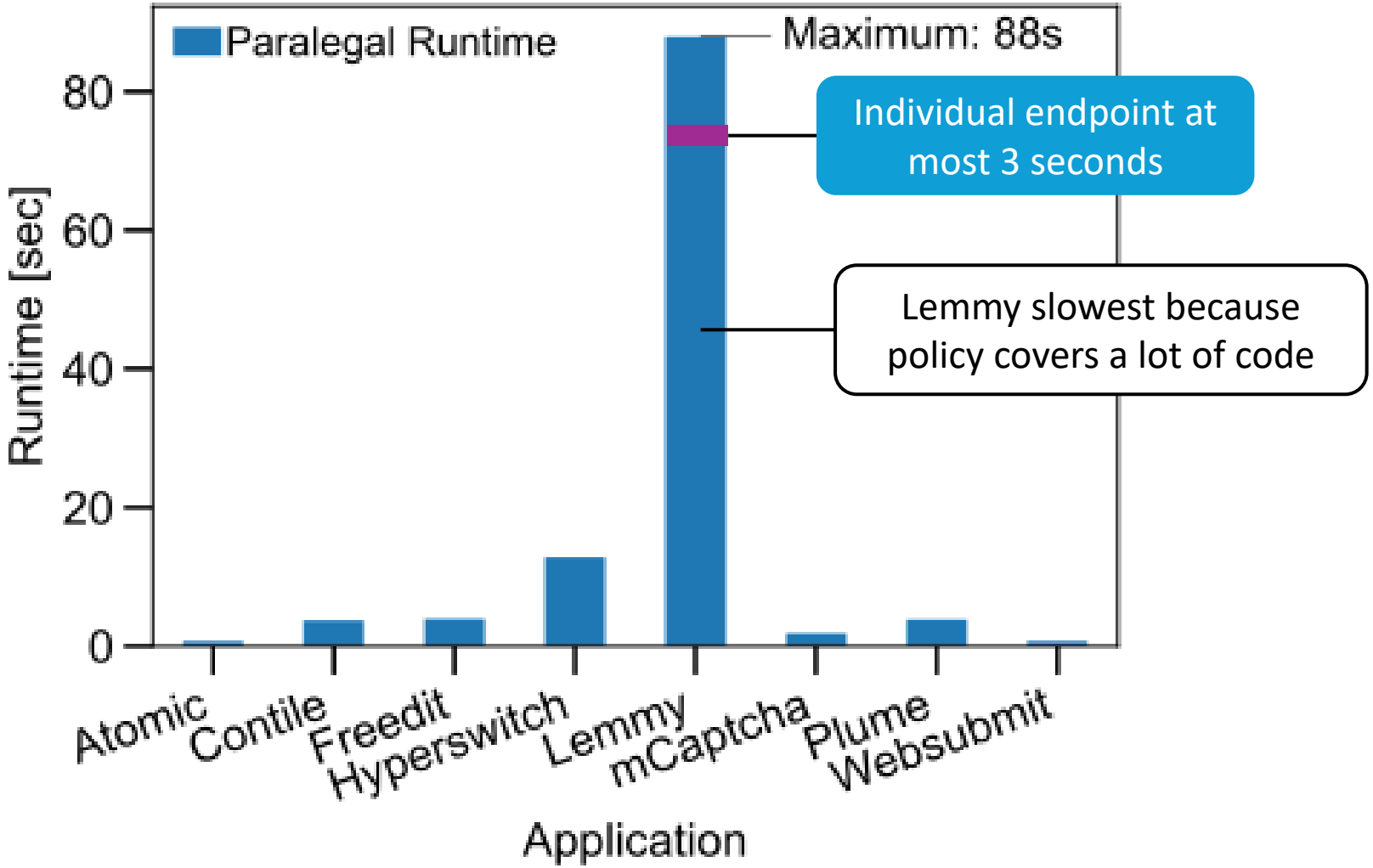
Performance Scales With Coverage

Setup: All dependencies analyzed (except stdlib)



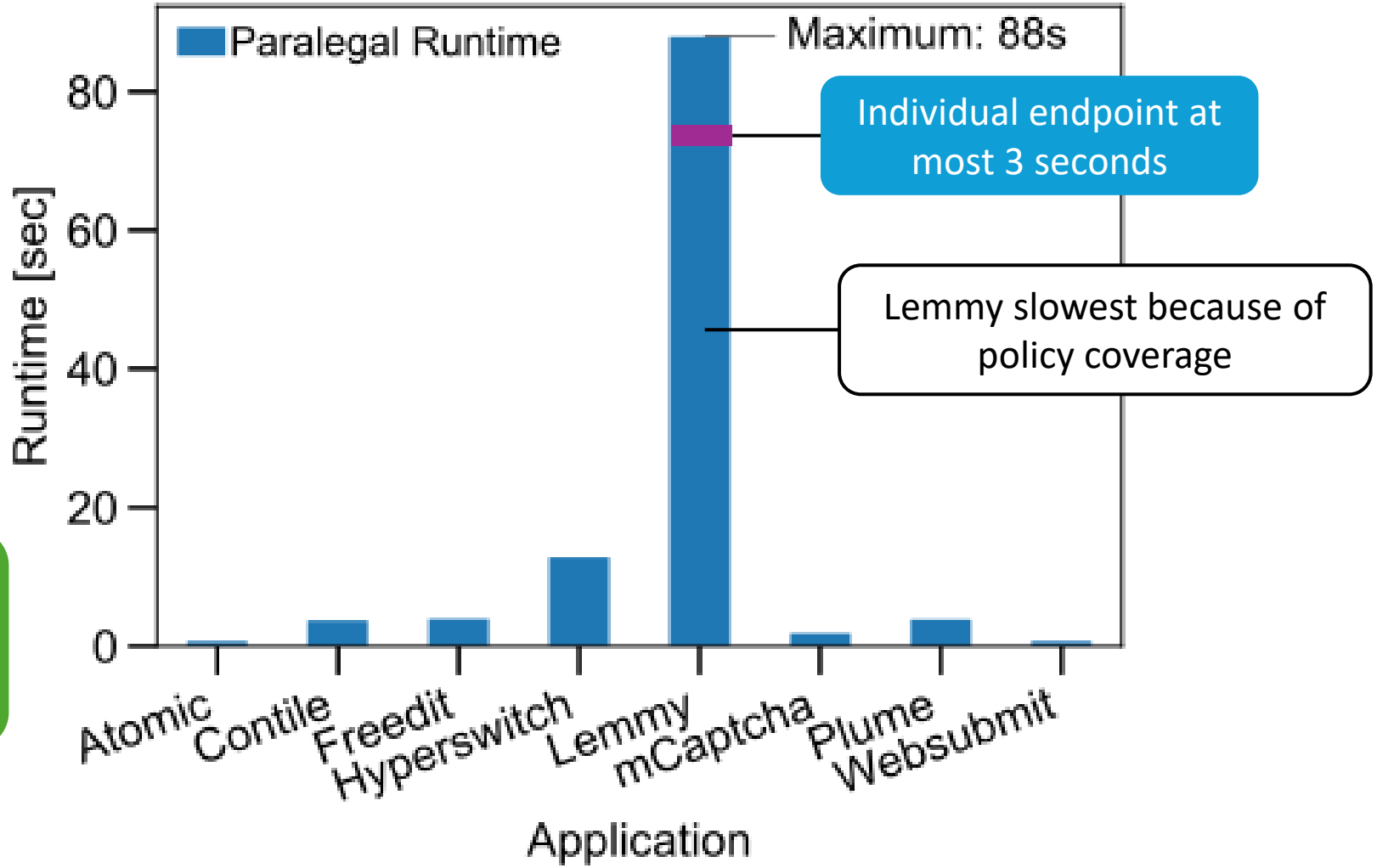
Performance Scales With Coverage

Setup: All dependencies analyzed (except stdlib)



Performance Scales With Coverage

Setup: All dependencies analyzed (except stdlib)



When including all dependencies, Paralegal's analysis is fast enough for CI

Impact of Optimizations

1. Does Paralegal find privacy bugs?
2. Is Paralegal fast enough to be practical?
3. What is the impact of Paralegal's optimizations?

Call Tree Pruning Is Necessary

1. Does Paralegal find privacy bugs?
2. Is Paralegal fast enough to be practical?
3. What is the impact of Paralegal's optimizations?

For two apps (Lemmy and Plume) optimization is necessary for analysis run out of memory after ~20min

Optimization reduces analysis time by 35% on average, more for large apps

Comparison With Other Tools

1. Does Paralegal find privacy bugs?
2. Is Paralegal fast enough to be practical?
3. What is the impact of Paralegal's optimizations?
4. How does Paralegal compare to the related work?

For two apps (Lemmy and Plume) optimization is necessary for analysis to terminate within 15 minutes

Optimization reduces analysis time by 35% on average, more for large apps

Paralegal is More Expressive and Reliable Than Similar Approaches

1. Does Paralegal find privacy bugs?
2. Is Paralegal fast enough to be practical?
3. What is the impact of Paralegal's optimizations?
4. How does Paralegal compare to the related work?

For two apps (Lemmy and Plume) optimization is necessary for analysis to terminate within 15 minutes

Analysis time reduced by 35% on average, more for large apps

IFC can only express **6 out of 11** of our policies and requires source-code changes.
CodeQL can express them but only finds bugs in **4 out of 8** we tried.

In the paper:

1. Markers make policies robust.
2. Paralegal can be used interactively.

Conclusion

Paralegal: a practical, automated tool for checking privacy policies in Rust.

- Auditable, flexible policy language
- Robust separation of code and policy with markers
- Scales to real-world application sizes
- Type-based approximation for libraries

Paralegal finds real and semantically subtle bugs in real-world applications.

Code available at
https://github.com/brownsys/paralegal

Find us around the conference.


Carolyn


Will


Mithi


Malte


Justus