Modular Information Flow through Ownership

Will Crichton
Stanford University
Stanford, USA
wcrichto@cs.stanford.edu

Marco Patrignani
University of Trento
Trento, Italy
marco.patrignani@unitn.it

Maneesh Agrawala
Pat Hanrahan
Stanford University
Stanford, USA

Abstract

Statically analyzing information flow, or how data influences other data within a program, is a challenging task in imperative languages. Analyzing pointers and mutations requires access to a program's complete source. However, programs often use pre-compiled dependencies where only type signatures are available. We demonstrate that ownership types can be used to soundly and precisely analyze information flow through function calls given only their type signature. From this insight, we built Flowistry, a system for analyzing information flow in Rust, an ownership-based language. We prove the system's soundness as a form of noninterference using the Oxide formal model of Rust. Then we empirically evaluate the precision of Flowistry, showing that modular flows are identical to whole-program flows in 94% of cases drawn from large Rust codebases. We illustrate the applicability of Flowistry by using it to implement prototypes of a program slicer and an information flow control system.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Keywords: information flow, ownership types, rust

ACM Reference Format:
Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular Information Flow through Ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519939.3523445

1 Introduction

Information flow describes how data influences other data within a program. Information flow has applications to security, such as information flow control [31], and to developer tools, such as program slicing [37]. Our goal is to build a practical system for analyzing information flow, meaning:

1. **Applicable to common language features:** the language being analyzed should support widely used features like pointers and in-place mutation.
2. **Zero configuration to run on existing code:** the analyzer must integrate with an existing language and existing unannotated programs. It must not require users to adopt a new language designed for information flow.
3. **No dynamic analysis:** to reduce integration challenges and costs, the analyzer must be purely static — no modifications to runtimes or binaries are needed.
4. **Modular over dependencies:** programs may not have source available for dependencies. The analyzer must have reasonable precision without whole-program analysis.

As a case study on the challenges imposed by these requirements, consider analyzing the information that flows to the return value in this C++ function:

```
1 // Copy elements 0 to max into a new vector
2 vector<int> copy_to(vector<int>& v, size_t max) {
3   vector<int> v2; size_t i = 0;
4   for (auto x(v.begin()); x != v.end(); ++x) {
5     if (i == max) { break; }
6     v2.push_back(*x); ++i;
7   }
8   return v2;
9 }
```

Here, a key flow is that `v2` is influenced by `v`: (1) `push_back` mutates `v2` with `*x` as input, and (2) `x` points to data within `v`. But how could an analyzer statically deduce these facts? For C++, the answer is *by looking at function implementations*. The implementation of `push_back` mutates `v2`, and the implementation of `begin` returns a pointer to data in `v`.

However, analyzing such implementations violates our fourth requirement, since these functions may only have their type signature available. In C++, given only a function's type signature, not much can be inferred about its behavior, since the type system does not contain information relevant to pointer analysis.

Our key insight is that *ownership types* can be leveraged to modularly analyze pointers and mutation using only a function's type signature. Ownership has emerged from several intersecting lines of research on linear logic [16], class-based alias management [8], and region-based memory management [18]. The fundamental law of ownership is that data

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523445

cannot be simultaneously aliased and mutated. Ownership-based type systems enforce this law by tracking which entities own which data, allowing ownership to be transferred between entities, and flagging ownership violations like mutating immutably-borrowed data.

Today, the most popular ownership-based language is Rust. Consider the information flows in this Rust implementation of `copy_to`:

```
fn copy_to(v: &Vec<i32>, max: usize) -> Vec<i32> {
  let mut v2 = Vec::new();
  for (i, x) in v.iter().enumerate() {
    if i == max { break; }
    v2.push(*x);
  }
  return v2;
}
```

Focus on the two methods `push` and `iter`. For a `Vec<i32>`, these methods have the following type signatures:

```
fn push(&mut self, value: i32);
fn iter<'a>(&'a self) -> Iter<'a, i32>;
```

To determine that `push` mutates `v2`, we leverage *mutability modifiers*. All references in Rust are either immutable (i.e. the type is `&T`) or mutable (the type is `&mut T`). Therefore `iter` does not mutate `v` because it takes `&self` as input (excepting interior mutability, discussed in Section 4.3), while `push` may mutate `v2` because it takes `&mut self` as input.

To determine that `x` points to `v`, we leverage *lifetimes*. All references in Rust are annotated with a lifetime, either explicitly (such as `'a`) or implicitly. Shared lifetimes indicate aliasing: because `&self` in `iter` has lifetime `'a`, and because the returned `Iter` structure shares that lifetime, then we can determine that `Iter` may contain pointers to `self`.

Inspired by this insight, we built Flowistry, a system for analyzing information flow in the safe subset of Rust programs. Flowistry satisfies our four design criteria: (1) Rust supports pointers and mutation, (2) Flowistry does not require any change to the Rust language or to Rust programs, (3) Flowistry is a purely static analysis, and (4) Flowistry uses ownership types to analyze function calls without needing their definition. This paper presents a theoretical and empirical investigation into Flowistry in five parts:

1. We provide a precise description of how Flowistry computes information flow by embedding its definition within Oxide [38], a formal model of Rust (Section 2).
2. We prove the soundness of our information flow analysis as a form of noninterference (Section 3).
3. We describe the implementation of Flowistry that bridges the theory of Oxide to the practicalities of Rust (Section 4).
4. We evaluate the precision of the modular analysis on a dataset of large Rust codebases, finding that modular flows are identical to whole-program flows in 94% of cases, and are on average 7% larger in the remaining cases (Section 5).
5. We demonstrate the utility of Flowistry by using it to prototype a program slicer and an IFC checker (Section 6).

We conclude by presenting related work (Section 7) and discussing future directions for Flowistry (Section 8). Due to space constraints, we omit many formal details, all auxiliary lemmas, and all proofs. The interested reader can find them in the appendix. Flowistry and our applications of it are publicly available, open-source, MIT-licensed projects at https://github.com/willcrichton/flowistry.

2 Analysis

Inspired by the dependency calculus of Abadi et al. [1], our analysis represents information flow as a set of dependencies for each variable in a given function. The analysis is flow-sensitive, computing a different dependency set at each program location, and field-sensitive, distinguishing between dependencies for fields of a data structure.

While the analysis is implemented in and for Rust, our goal here is to provide a description of it that is both concise (for clarity of communication) and precise (for amenability to proof). We therefore base our description on Oxide [38], a formal model of Rust. At a high level, Oxide provides three ingredients:

1. A syntax of Rust-like programs with expressions e and types τ.
2. A type-checker, expressed with the judgment $\Sigma; \Delta; \Gamma \vdash e : \tau \Rightarrow \Gamma'$ using the contexts Γ for types and lifetimes, Δ for type variables, and Σ for global functions.
3. An interpreter, expressed by a small-step operational semantics with the judgment $\Sigma \vdash (\sigma; e) \rightarrow (\sigma'; e')$ using σ for a runtime stack.

We extend this model by assuming that each expression in a program is automatically labeled with a unique location ℓ. Then for a given expression e, our analysis computes the set of dependencies $\kappa ::= \{\overline{\ell}\}$. Because expressions have effects on persistent memory, we further compute a *dependency context* $\Theta ::= \{\overline{p \mapsto \kappa}\}$ from memory locations p to dependencies κ. The computation of information flow is intertwined with type-checking, represented as a modified type-checking judgment (additions highlighted in red):

$$\Sigma; \Delta; \Gamma; \Theta \vdash e_\ell : \tau \bullet \kappa \Rightarrow \Gamma'; \Theta'$$

This judgment is read as, "with type contexts Σ, Δ, Γ and dependency context Θ, e at location ℓ has type τ and dependencies κ, producing a new dependency context Θ'."

Oxide is a large language — describing every feature, judgment, and inference rule would exceed our space constraints. Instead, in this section we focus on a few key rules that demonstrate the novel aspects of our system. We first lay the foundations for dealing with variables and mutation (Section 2.1), and then describe how we modularly analyze references (Section 2.2) and function calls (Section 2.3). The remaining rules can be found in the appendix.

2.1 Variables and Mutation

The core of Oxide is an imperative calculus with constants and variables. The abstract syntax for these features is below:

$$\text{Variable } x \quad \text{Number } n$$
$$\text{Path } q ::= \varepsilon \mid n.q$$
$$\text{Place } \pi ::= x.q$$
$$\text{Constant } c ::= () \mid n \mid \text{true} \mid \text{false}$$
$$\text{Base Type } \tau^{\text{B}} ::= \text{unit} \mid \text{u32} \mid \text{bool}$$
$$\text{Sized Type } \tau^{\text{SI}} ::= \tau^{\text{B}} \mid (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}}) \mid \ldots$$
$$\text{Expression } e ::= c \mid \pi \mid \texttt{let}\ x : \tau_a^{\text{SI}} = e_1;\ e_2 \mid$$
$$\pi := e \mid e_1;\ e_2 \mid \ldots$$

Constants are Oxide's atomic values and also the base-case for information flow. A constant's dependency is simply itself, expressed through the T-U32 rule:

T-U32

$$\frac{}{\Sigma; \Delta; \Gamma; \Theta \vdash n_\ell : \text{u32} \bullet \{\ell\} \Rightarrow \Gamma; \Theta}$$

Variables and mutation are introduced through let-bindings and assignment expressions, respectively. For example, this (location-annotated) program mutates a field of a tuple:

$$\texttt{let}\ t : (\text{u32}, \text{u32}) = (1_{\ell_1}, 2_{\ell_2});\ t.1 := 3_{\ell_3}$$

Here, t is a variable and $t.1$ is a *place*, or a description of a specific region in memory. For information flow, the key idea is that let-bindings introduce a set of places into Θ, and then assignment expressions change a place's dependencies within Θ. In the above example, after binding t, then Θ is:

$$\Theta = \{t, t.0, t.1 \mapsto \{\ell_1, \ell_2\}\}$$

After checking "$t.1 := 3$", then ℓ_3 is added to $\Theta(t)$ and $\Theta(t.1)$, but not $\Theta(t.0)$. This is because the values of t and $t.1$ have changed, but the value of $t.0$ has not. Formally, the let-binding rule is:

T-Let

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e_1 : \tau_1^{\text{SI}} \bullet \kappa_1 \Rightarrow \Gamma_1; \Theta_1 \\ \Gamma; \Delta_1 \vdash \tau_1^{\text{SI}} \lesssim \tau_a^{\text{SI}} \Rightarrow \Gamma_1' \qquad \Theta_1' = \Theta_1[\forall \pi^{\square}[x]\ .\ \pi \mapsto \kappa_1] \\ \Sigma; \Delta; \text{gc-loans}(\Gamma_1', x : \tau_a^{\text{SI}}); \Theta_1' \vdash e_2 : \tau_2^{\text{SI}} \bullet \kappa_2 \Rightarrow \Gamma_2, x : \tau^{\text{SD}}; \Theta_2\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \texttt{let}\ x : \tau_a^{\text{SI}} = e_1;\ e_2 : \tau_2^{\text{SI}} \bullet \kappa_2 \Rightarrow \Gamma_2; \Theta_2}$$

Again, this rule (and many others) contain aspects of Oxide that are not essential for understanding information flow such as the subtyping judgment $\tau_1 \lesssim \tau_2$ or the metafunction gc-loans. For brevity we will not cover these aspects here, and instead refer the interested reader to Weiss et al. [38]. We have deemphasized (in grey) the judgments which are not important to understanding our information flow additions.

The key concept is the formula $\Theta_1[\forall \pi^{\square}[x]\ .\ \pi \mapsto \kappa_1]$. This introduces two shorthands: first, $\pi^{\square}[x]$ means "a place π with root variable x in a context $\pi^{\square}$", used to decompose a place. In T-Let, the update to Θ_1 happens for all places with a root variable x. Second, $\Theta_1[\pi \mapsto \kappa_1]$ means "set π to κ_1 in Θ_1". So this rule specifies that when checking e_2, all places within x are initialized to the dependencies κ_1 of e_1.

Next, the assignment expression rule is defined as updating all the *conflicts* of a place π:

T-Assign

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma; \Theta \vdash e : \tau^{\text{SI}} \bullet \kappa \Rightarrow \Gamma_1; \Theta_1 \\ \Gamma_1(\pi) = \tau^{\text{SX}} \qquad (\tau^{\text{SX}} = \tau^{\text{SD}} \vee \Delta; \Gamma_1 \vdash_{\text{uniq}} \pi \Rightarrow \{\ ^{\text{uniq}}\pi\}) \\ \Delta; \Gamma_1 \vdash \tau^{\text{SI}} \lesssim \tau^{\text{SX}} \Rightarrow \Gamma' \\ \Theta_2 = \Theta_1[\text{update-conflicts}\ (\Theta_1, \pi, \kappa)]\end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash \pi := e : \text{unit} \bullet \varnothing \Rightarrow \Gamma'[\pi \mapsto \tau^{\text{SI}}] \triangleright \pi; \Theta_2}$$

If you conceptualize a type as a tree and a path as a node in that tree, then a node's conflicts are its ancestors and descendants (but not siblings). Semantically, conflicts are the set of places whose value change if a given place is mutated. Recall from the previous example that $t.1$ conflicts with t and $t.1$, but not $t.0$. Formally, we say two places are disjoint ($\#$) or conflict ($\sqcap$) when:

$$x_1.q_1 \ \#\ x_2.q_2 \stackrel{\text{def}}{=} x_1 \neq x_2 \vee (\ (q_1 \text{ is not a prefix of } q_2) \wedge (q_2 \text{ is not a prefix of } q_1))$$
$$\pi_1 \sqcap \pi_2 \stackrel{\text{def}}{=} \neg(\pi_1 \ \#\ \pi_2)$$

Then to update a place's conflicts in Θ, we define the metafunction update-conflicts to add κ to all conflicting places p'. (Note that this rule is actually defined over place *expressions* p, which are explained in the next subsection.)

$$\text{update-conflicts}\ (\Theta, p, \kappa) \stackrel{\text{def}}{=}$$
$$\forall p' \mapsto \kappa_{p'} \in \Theta_{\text{cfl}}\ .\ p' \mapsto \kappa_{p'} \cup \kappa$$
$$\text{where } \Theta_{\text{cfl}} = \{p' \mapsto \kappa_{p'} \in \Theta \mid p \sqcap p'\}$$

Finally, the rule for reading places is simply to look up the place's dependencies in Θ:

T-Move

$$\frac{\Delta; \Gamma \vdash_{\text{uniq}} \pi \Rightarrow \{\ ^{\text{uniq}}\pi\} \qquad \Gamma(\pi) = \tau^{\text{SI}} \qquad \text{noncopyable}_{\Sigma}\ \tau^{\text{SI}}}{\Sigma; \Delta; \Gamma; \Theta \vdash \pi : \tau^{\text{SI}} \bullet \Theta(\pi) \Rightarrow \Gamma[\pi \mapsto \tau^{\text{SI}^\dagger}]; \Theta}$$

2.2 References

Beyond concrete places in memory, Oxide also contains references that point to places. As in Rust, these references have both a lifetime (called a "provenance") and a mutability qualifier (called an "ownership qualifier"). Their syntax is:

$$\text{Concrete Provenance } r \quad \text{Abstract Provenance } \varrho$$
$$\text{Place Expression } p ::= x \mid *p \mid p.n$$
$$\text{Provenance } \rho ::= \varrho \mid r$$
$$\text{Ownership Qual. } \omega ::= \text{shrd} \mid \text{uniq}$$
$$\text{Sized Type } \tau^{\text{SI}} ::= \ldots \mid \&\rho\ \omega\ \tau^{\text{XI}}$$
$$\text{Expression } e ::= \ldots \mid \&r\ \omega\ p \mid p := e \mid \texttt{letprov}\langle r \rangle\ e$$

Provenances are created via a **letprov** expression, and references are created via a borrow expression $\&r\ \omega\ p$ that has an initial concrete provenance r (abstract provenances are just used for types of function parameters). References are used in conjunction with place expressions p that are places whose paths contain dereferences. For example, this program creates, reborrows, and mutates a reference:

$$\begin{aligned}
&\textbf{letprov}\langle r_1, r_2, r_3, r_4\rangle\\
&\textbf{let}\ x : (\mathsf{u32}, \mathsf{u32}) = (0, 0);\\
&\textbf{let}\ y : \&r_2\ \mathsf{uniq}\ (\mathsf{u32}, \mathsf{u32}) = \&r_1\ \mathsf{uniq}\ x;\\
&\textbf{let}\ z : \&r_4\ \mathsf{uniq}\ \mathsf{u32} = \&r_3\ \mathsf{uniq}\ (*y).1;\\
&*z := 1_\ell
\end{aligned}$$

Consider the information flow induced by $*z := 1_\ell$. We need to compute all places that z could point-to, in this case $x.1$, so ℓ can be added to the conflicts of $x.1$. Essentially, we must perform a *pointer analysis* [33].

The key idea is that Oxide already does a pointer analysis! Performing one is an essential task in ensuring ownership-safety. All we have to do is extract the relevant information with Oxide's existing judgments. This is represented by the information flow extension to the reference-mutation rule:

$$\textsc{T-AssignDeref}\quad \frac{\begin{array}{c}\Sigma;\Delta;\Gamma;\Theta \vdash e : \tau_n^{\mathrm{SI}} \bullet \kappa \Rightarrow \Gamma_1;\Theta_1 \qquad \Delta;\Gamma_1 \vdash_{\mathsf{uniq}} p : \tau_o^{\mathrm{SI}}\\ \Delta;\Gamma_1 \vdash_{\mathsf{uniq}} p \Rightarrow \{\bar{l}\} \qquad \Delta;\Gamma_1 \vdash \tau_n^{\mathrm{SI}} \lesssim \tau_o^{\mathrm{SI}} \Rightarrow \Gamma'\\ \Theta_2 = \Theta_1[\forall\ {}^{\omega}p' \in \{\bar{l}\}\ .\ \text{update-conflicts}\ (\Theta_1, p', \kappa)]\end{array}}{\Sigma;\Delta;\Gamma;\Theta \vdash p := e : \mathsf{unit} \bullet \varnothing \Rightarrow \Gamma' \triangleright p;\Theta_2}$$

Here, the important concept is Oxide's ownership safety judgment: $\Delta;\Gamma \vdash_\omega p \Rightarrow \{\bar{l}\}$, read as "in the contexts Δ and Γ, p can be used ω-ly and points to a loan in $\{\bar{l}\}$." A loan $l ::= {}^{\omega}p$ is a place expression with an ownership-qualifier. In Oxide, this judgment is used to ensure that a place is used safely at a given level of mutability. For instance, in the example at the top of this column, if $*z := 1$ was replaced with $x.1 := 1$, then this would violate ownership-safety because x is already borrowed by y and z.

In the example as written, the ownership-safety judgment for $*z$ would compute the loan set:

$$\{\bar{l}\} = \{\,{}^{\mathsf{uniq}}(*z), {}^{\mathsf{uniq}}(*y).1, {}^{\mathsf{uniq}}x.1\}$$

Note that $x.1$ is in the loan set of $*z$. That suggests the loan set can be used as a pointer analysis. The complete details of computing the loan set can be found in Weiss et al. [38, p. 12], but the summary for this example is:

1. Checking the borrow expression "$\&r_1\ \mathsf{uniq}\ x$" gets the loan set for x, which is just $\{{}^{\mathsf{uniq}}x\}$, and so sets $\Gamma(r_1) = \{{}^{\mathsf{uniq}}x\}$.
2. Checking the assignment "$y = \&r_1\ \mathsf{uniq}\ x$" requires that $\&r_1\ \mathsf{uniq}\ (\mathsf{u32}, \mathsf{u32})$ is a subtype of $\&r_2\ \mathsf{uniq}\ (\mathsf{u32}, \mathsf{u32})$, which requires that r_1 "outlives" r_2, denoted $r_1 :> r_2$.
3. The constraint $r_1 :> r_2$ adds $\Gamma(r_1)$ to $\Gamma(r_2)$, so $\Gamma(r_2) = \{{}^{\mathsf{uniq}}x\}$.
4. Checking "$\&r_3\ \mathsf{uniq}\ (*y).1$" gets the loan set for $(*y).1$, which is:

$$\{{}^{\mathsf{uniq}}p.1 \mid {}^{\mathsf{uniq}}p \in \Gamma(r_2)\} \cup \{{}^{\mathsf{uniq}}(*y).1\} = \{{}^{\mathsf{uniq}}x.1, {}^{\mathsf{uniq}}(*y).1\}$$

   That is, the loans for r_2 are looked up in Γ (to get $\{x\}$), and then the additional projection _.1 is added on-top of each loan (to get $\{x.1\}$).
5. Then $\Gamma(r_4) = \Gamma(r_3)$ because $r_3 :> r_4$.
6. Finally, the loan set for $*z$ is:

$$\Gamma(r_4) \cup \{{}^{\mathsf{uniq}}(*z)\} = \{{}^{\mathsf{uniq}}x.1, {}^{\mathsf{uniq}}(*y).1, {}^{\mathsf{uniq}}(*z)\}$$

Applying this concept to the T-AssignDeref rule, we compute information flow for reference-mutation as: when mutating p with loans $\{\bar{l}\}$, add κ_e to all the conflicts for every loan ${}^{\mathsf{uniq}}p' \in \{\bar{l}\}$.

2.3 Function Calls

Finally, we examine how to modularly compute information flow through function calls, starting with syntax:

$$\begin{aligned}
&\text{Type Var } \alpha \qquad \text{Frame Var } \varphi\\
&\text{Expression } e ::= \ldots \mid f\langle\overline{\Phi}, \overline{\rho}, \overline{\tau}\rangle(\pi)\\
&\text{Global Entry } \varepsilon ::= \textbf{fn}\ f\langle\overline{\varphi}, \overline{\varrho}, \overline{\alpha}, \overline{\varrho_1 : \varrho_2}\rangle(x : \tau_a^{\mathrm{SI}}) \longrightarrow \tau_r^{\mathrm{SI}}\ \{\ e\ \}\\
&\text{Global Env. } \Sigma ::= \bullet \mid \Sigma, \varepsilon
\end{aligned}$$

Oxide functions are parameterized by frame variables φ (for closures), abstract provenances ϱ (for provenance polymorphism), and type variables α (for type polymorphism). Unlike Oxide, we restrict to functions with one argument for simplicity in the formalism. Calling a function f requires an argument π and any type-level parameters Φ, ρ and τ.

The key question is: without inspecting its definition, what is the *most precise* assumption we can make about a function's information flow while still being sound? By "precise" we mean "if the analysis says there is a flow, then the flow actually exists", and by "sound" we mean "if a flow actually exists, then the analysis says that flow exists." For example consider this program:

$$\begin{aligned}
&\textbf{fn}\ \mathsf{f}\langle\varrho_1, \varrho_2\rangle(x : (\&\varrho_1\ \mathsf{uniq}\ \mathsf{u32}, \&\varrho_2\ \mathsf{shrd}\ \mathsf{u32}))\{\ ???\ \}\\
&\textbf{let}\ x : \mathsf{u32} = 1_{\ell_1};\ \textbf{let}\ y : \mathsf{u32} = 2_{\ell_2};\\
&\textbf{letprov}\langle r_1, r_2\rangle\ \textbf{let}\ t : (\&r_1\ \mathsf{uniq}\ \mathsf{u32}, \&r_2\ \mathsf{shrd}\ \mathsf{u32})\\
&\quad = (\&r_1\ \mathsf{uniq}\ x, \&r_2\ \mathsf{shrd}\ y);\\
&\mathsf{f}\langle r_1, r_2\rangle(t)
\end{aligned}$$

First, what can $\mathsf{f}(t)$ mutate? Any data behind a shared reference is immutable, so only $*t.0$ could possibly be mutated, not $*t.1$. More generally, the argument's *transitive mutable references* must be assumed to be mutated.

Second, what are the inputs to the mutation of $*t.0$? This could theoretically be any possible value in the input, so both $*t.0$ and $*t.1$. More generally, every *transitively readable place* from the argument must be assumed to be to be an input to the mutation. So in this example, a modular analysis of the

information flow from calling cp would add $\{\ell_1, \ell_2\}$ to $\Theta(x)$ but not $\Theta(y)$.

To formalize these concepts, we first need to describe the transitive references of a place. The ω-refs(p, τ) meta-function computes a place expression for every reference accessible from p. If $\omega = \text{uniq}$ then this just includes unique references, otherwise it includes unique and shared ones.

$$\omega\text{-refs}(p, \tau^{\text{B}}) = \varnothing$$
$$\omega\text{-refs}(p, (\tau_1^{\text{SI}}, \ldots, \tau_n^{\text{SI}})) = \bigcup_i \omega\text{-refs}(p.i, \tau_i^{\text{SI}})$$
$$\omega\text{-refs}(p, \&\rho\, \omega'\, \tau^{\text{XI}}) = \begin{cases} \{*p\} \cup \omega\text{-refs}(*p, \tau^{\text{XI}}) & \text{if } \omega \lesssim \omega' \\ \varnothing & \text{otherwise} \end{cases}$$

Here, $\omega \lesssim \omega'$ means "a loan at ω can be used as a loan at ω'", defined as uniq $\not\lesssim$ shrd and $\omega \lesssim \omega'$ otherwise. Then ω-loans$(p, \tau, \Delta, \Gamma)$ can be defined as the set of concrete places accessible from those transitive references:

$$\omega\text{-loans}(p, \tau, \Delta, \Gamma) \stackrel{\text{def}}{=} \bigcup_{p_1 \in \omega\text{-refs}(p,\tau)} \{p_2 \mid {}^{\omega}p_2 \in \{\bar{l}\}\} \quad \text{where } \Delta; \Gamma \vdash_{\omega} p_1 \Rightarrow \{\bar{l}\}$$

Finally, the function application rule can be revised to include information flow as follows:

T-App

$$\frac{\begin{array}{c}\overline{\Sigma; \Delta; \Gamma \vdash \Phi} \qquad \overline{\Delta; \Gamma \vdash \rho} \qquad \overline{\Sigma; \Delta; \Gamma \vdash \tau^{\text{SI}}} \\ \Sigma(f) = \texttt{fn}\, f\langle \overline{\varphi}, \overline{\varrho}, \overline{\alpha}, \overline{\varrho_1 : \varrho_2}\rangle(x : \tau_a^{\text{SI}}) \to \tau_r^{\text{SI}}\ \{\ e\ \} \\ \Sigma; \Delta; \Gamma; \Theta \vdash \pi : \tau_a^{\text{SI}}\overline{[\Phi/\varphi][\rho/\varrho][\tau^{\text{SI}}/\alpha]} \bullet \kappa \Rightarrow \Gamma_1; \Theta \\ \Delta; \Gamma_1 \vdash \varrho_2\overline{[\rho/\varrho]} :> \varrho_1\overline{[\rho/\varrho]} \Rightarrow \Gamma_2 \\ \kappa_{\text{arg}} = \kappa \cup \bigcup_{p \in \text{shrd-loans}(\pi, \tau_a^{\text{SI}}, \Delta, \Gamma_2)} \Theta(p) \\ \Theta' = \Theta[\forall p \in \text{uniq-loans}(\pi, \tau_a^{\text{SI}}, \Delta, \Gamma_2)\ . \\ \qquad \text{update-conflicts}\,(\Theta, p, \kappa_{\text{arg}})] \end{array}}{\Sigma; \Delta; \Gamma; \Theta \vdash f\langle \overline{\Phi}, \overline{\rho}, \overline{\tau^{\text{SI}}}\rangle(\pi) : \tau_r^{\text{SI}}\overline{[\Phi/\varphi][\rho/\varrho][\tau^{\text{SI}}/\alpha]} \bullet \kappa_{\text{arg}} \Rightarrow \Gamma_2; \Theta'}$$

The collective dependencies of the input π are collected into κ_{arg}, and then every unique reference is updated with κ_{arg}. Additionally, the function's return value is assumed to be influenced by any input, and so has dependencies κ_{arg}.

Note that this rule does not depend on the body e of the function f, only its type signature in Σ. This is the key to the modular approximation. Additionally, it means that this analysis can trivially handle higher-order functions. If f were a parameter to the function being analyzed, then no control-flow analysis is needed to guess its definition.

3 Soundness

To characterize the correctness of our analysis, we seek to prove its *soundness*: if a true information flow exists in a program, then the analysis computes that flow. The standard soundness theorem for information flow systems is *noninterference* [17]. At a high level, noninterference states that for a given program and its dependencies, and for any two execution contexts, if the dependencies are equal between contexts, then the program will execute with the same observable behavior in both cases. For our analysis, we focus just on values produced by the program, instead of other behaviors like termination or timing.

To formally define noninterference within Oxide, we first need to explore Oxide's operational semantics. Oxide programs are executed in the context of a stack of frames that map variables to values:

$$\text{Stack } \sigma ::= \bullet \mid \sigma \natural \varsigma$$
$$\text{Stack Frame } \varsigma ::= \bullet \mid \varsigma, x \mapsto v$$

For example, in the empty stack $\bullet$, the expression "**let** x : u32 = 1; x := 2" would first add $x \mapsto 1$ to the stack. Then executing $x := 2$ would update $\sigma(x) = 2$. More generally, we use the shorthand $\sigma(p)$ to mean "reduce p to a concrete location π, then look up the value of π in σ."

The key ingredient for noninterference is the equivalence of dependencies between stacks. That is, for two stacks σ_1 and σ_2 and a set of dependencies κ in a context Θ, we say those stacks are *the same up to* κ if all p with $\Theta(p) \subseteq \kappa$ are the same between stacks. Formally, the dependencies of κ and equivalence of heaps are defined as:

$$\text{deps}(\Theta, \kappa) \stackrel{\text{def}}{=} \{p \mid p \mapsto \kappa_p \in \Theta \wedge \kappa_p \subseteq \kappa\}$$
$$\sigma_1 \sim_P \sigma_2 \stackrel{\text{def}}{=} \forall p \in P\ .\ \sigma_1(p) = \sigma_2(p)$$
$$\sigma_1 \sim_{\kappa}^{\Theta} \sigma_2 \stackrel{\text{def}}{=} \sigma_1 \sim_{\text{deps}(\Theta,\kappa)} \sigma_2$$

Then we define noninterference as follows:

Theorem 3.1 (Noninterference). *Let e such that:*

$$\Sigma; \bullet; \Gamma; \Theta \vdash e : \tau \bullet \kappa \Rightarrow \Gamma'; \Theta'$$

For $i \in \{1, 2\}$, let σ_i such that:

$$\Sigma \vdash \sigma_i : \Gamma \quad \textit{and} \quad \Sigma \vdash (\sigma_i; e) \xrightarrow{*} (\sigma_i'; v_i)$$

Then:

(a) $\sigma_1 \sim_{\kappa}^{\Theta} \sigma_2 \implies v_1 = v_2$
(b) $\forall p \mapsto \kappa_p \in \Theta'\ .\ \sigma_1 \sim_{\kappa_p}^{\Theta} \sigma_2 \implies \sigma_1'(p) = \sigma_2'(p)$

This theorem states that given a well-typed expression e and corresponding stacks σ_i, then its output v_i should be equal if the expression's dependencies κ are initially equal. Moreover, for any place expression p, if its dependencies in the output context Θ' are initially equal then the stack value will be the same after execution.

Note that the context Δ is required to be empty because an expression e can only evaluate if it does not contain abstract type or provenance variables. The judgment $\Sigma \vdash \sigma_i : \Gamma$ means "the stack σ_i is well-typed under Σ and Γ". That is, for all places π in Γ, then $\pi \in \sigma$ and $\sigma(\pi)$ has type $\Gamma(\pi)$.

The proof of Theorem 3.1, found in the appendix, guarantees that we can soundly compute information flow for Oxide programs.

```
1  fn get_count(
2    h: &mut HashMap<String, u32>,
3    k: String
4  ) -> u32 {
5    if !h.contains_key(&k) {
6      h.insert(k, 0); 0
7    } else {
8      *h.get(&k).unwrap()
9    }
10 }
```

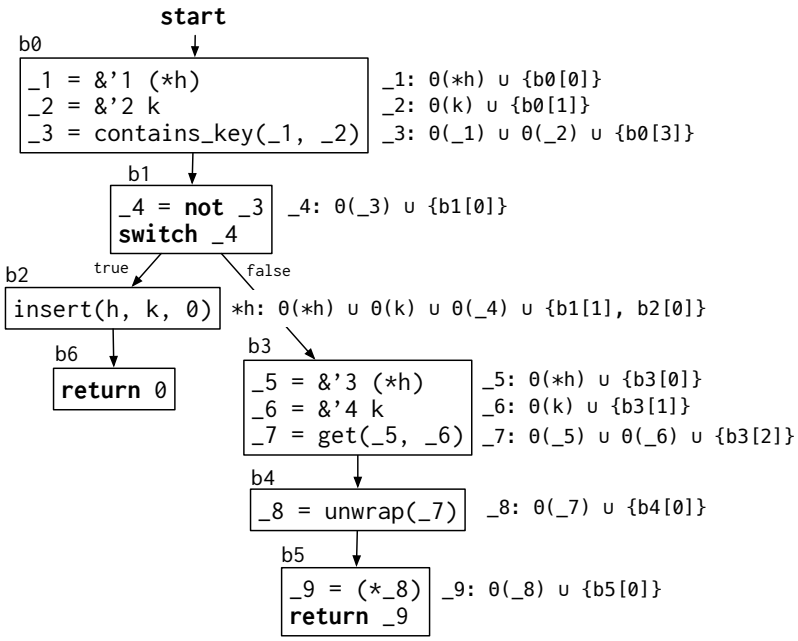

Figure 1. Example of how Flowistry computes information flow. On the left is a Rust function `get_count` that finds a value in a hash map for a given key, and inserts 0 if none exists. On the right `get_count` is lowered into Rust's MIR control-flow graph, annotated with information flow. Each rectangle is a basic block, named at the top. Arrows indicate control flow (panics omitted). Beside each instruction is the result of the information flow analysis, which maps place expressions to locations in the CFG (akin to Θ in Section 2). For example, the `insert` function call adds dependencies to `*h` because it is assumed to be mutated, since it is a mutable reference. Additionally, the `switch` instructions and `_4` variable are added as dependencies to `h` because the call to `insert` is control-dependent on the switch.

4 Implementation

Our formal model provides a sound theoretical basis for analyzing information flow in Oxide. However, Rust is a more complex language than Oxide, and the Rust compiler uses many intermediate representations beyond its surface syntax. Therefore in this section, we describe the key details of how our system, Flowistry, bridges theory to practice. Specifically:

1. Rust computes lifetime-related information on a control-flow graph (CFG) program representation, not the high-level AST. So we translate our analysis to work for CFGs (Section 4.1).
2. Rust does not compute the loan set for lifetimes directly like in Oxide. So we must reconstruct the loan sets given the information exported by Rust (Section 4.2).
3. Rust contains escape hatches for ownership-unsafe code that cannot be analyzed using our analysis. So we describe the situations in which our analysis is unsound for Rust programs (Section 4.3).

4.1 Analyzing Control-Flow Graphs

The Rust compiler lowers programs into a "mid-level representation", or MIR, that represents programs as a control-flow graph. Essentially, expressions are flattened into sequences of instructions (basic blocks) which terminate in instructions that can jump to other blocks, like a branch or function call. Figure 1 shows an example CFG and its information flow.

To implement the modular information flow analysis for MIR, we reused standard static analysis techniques for CFGs, i.e., a flow-sensitive, forward dataflow analysis pass where:

- At each instruction, we maintain a mapping from place expressions to a set of locations in the CFG on which the place is dependent, comparable to Θ in Section 2.
- A transfer function updates Θ for each instruction, e.g. $p := e$ follows the same rules as in T-AssignDeref by adding the dependencies of e to all conflicts of aliases of p.
- The input Θ^{in} to a basic block is the join of each of the output Θ_i^{out} for each incoming edge, i.e. $\Theta^{\text{in}} = \bigvee_i \Theta_i^{\text{out}}$. The join operation is key-wise set union, or more precisely:

$$\Theta_1 \vee \Theta_2 \stackrel{\text{def}}{=} \{x \mapsto \Theta_1(x) \cup \Theta_2(x) \mid x \in \Theta_1 \vee x \in \Theta_2\}$$

- We iterate this analysis to a fixpoint, which we are guaranteed to reach because $\langle \Theta, \vee \rangle$ forms a join-semilattice.

To handle indirect information flows via control flow, such as the dependence of `h` on `contains_key` in Figure 1, we compute the control-dependence between instructions. We define control-dependence following Ferrante et al. [15]: an instruction Y is control-dependent on X if there exists a

directed path P from X to Y such that any Z in P is post-dominated by Y, and X is not post-dominated by Y. An instruction X is post-dominated by Y if Y is on every path from X to a `return` node. We compute control-dependencies by generating the post-dominator tree and frontier of the CFG using the algorithms of Cooper et al. [9] and Cytron et al. [11], respectively.

Besides a return, the only other control-flow path out of a function in Rust is a panic. For example, each function call in Figure 1 actually has an implicit edge to a panic node (not depicted). Unlike exceptions in other languages, panics are designed to indicate unrecoverable failure. Therefore we exclude panics from our control-dependence analysis.

4.2 Computing Loan Sets from Lifetimes

To verify ownership-safety (perform "borrow-checking"), the Rust compiler does not explicitly build the loan sets of lifetimes (or provenances in Oxide terminology). The borrow checking algorithm performs a sort of flow-sensitive dataflow analysis that determines the range of code during which a lifetime is valid, and then checks for conflicts e.g. in overlapping lifetimes (see the non-lexical lifetimes RFC [24]).

However, Rust's borrow checker relies on the same fundamental language feature as Oxide to verify ownership-safety: outlives-constraints. For a given Rust function, Rust can output the set of outlives-constraints between all lifetimes in the function. These lifetimes are generated in the same manner as in Oxide, such as from inferred subtyping requirements or user-provided outlives-constraints. Then given these constraints, we compute loan sets via a process similar to the ownership-safety judgment described in Section 2.2. In short, for all instances of borrow expressions $\&r\,\omega\,p$ in the MIR program, we initialize $\Gamma(r) = \{p\}$. Then we propagate loans via $\Gamma(r) = \bigcup_{r' :> r} \Gamma(r')$ until Γ reaches a fixpoint.

4.3 Handling Ownership-Unsafe Code

Rust has a concept of *raw pointers* whose behavior is comparable to pointers in C. For a type `T`, an immutable reference has type `&T`, while an immutable raw pointer has type `*const T`. Raw pointers are not subject to ownership restrictions, and they can only be used in blocks of code demarcated as `unsafe`. They are primarily used to interoperate with other languages like C, and to implement primitives that cannot be proved as ownership-safe via Rust's rules.

Our pointer and mutation analysis fundamentally relies on ownership-safety for soundness. We do not try to analyze information flowing directly through unsafe code, as it would be subject to the same difficulties of C++ in Section 1. While this limits the applicability of our analysis, empirical studies have shown that most Rust code does not (directly) use unsafe blocks [2, 14]. We further discuss the impact and potential mitigations of this limitation in Section 8.

5 Evaluation

Section 3 established that our analysis is *sound*. The next question is whether it is *precise*: how many spurious flows are included by our analysis? We evaluate two directions:

1. What if the analysis had *more* information? If we could analyze the definitions of called functions, how much more precise are whole-program flows vs. modular flows?
2. What if the analysis had *less* information? If Rust's type system was more like C++, i.e. lacking ownership, then how much less precise do the modular flows become?

To answer these questions, we created three modifications to Flowistry:

- Whole-program: the analysis recursively analyzes information flow within the definitions of called functions. For example, if calling a function `f(&mut x, y)` where `f` does not actually modify `x`, then the Whole-program analysis will not register a flow from `y` to `x`.
- Mut-blind: the analysis does not distinguish between mutable and immutable references. For example, if calling a function `f(&x)`, then the analysis assumes that `x` can be modified.
- Ref-blind: the analysis does not use lifetimes to reason about references, and rather assumes all references of the same type can alias. For example, if a function takes as input `f(x: &mut i32, y: &mut i32)` then `x` and `y` are assumed to be aliases.

The Whole-program modification represents the most precise information flow analysis we can feasibly implement. The Mut-blind and Ref-blind modifications represent an ablation of the precision provided by ownership types. Each modification can be combined with the others, representing $2^3 = 8$ possible conditions for evaluation.

To better understand Whole-program, say we are analyzing the information flow for an expression `f(&mut x, y)` where `f` is defined as `f(a, b) { (*a).1 = b; }`. After analyzing the implementation of `f`, we translate flows to parameters of `f` into flows on arguments of the call to `f`. So the flow `b` $\rightarrow$ `(*a).1` is translated into `y` $\rightarrow$ `x.1`. Additionally, if the definition of `f` is not available, then we fall back to the modular analysis. Importantly, due to the architecture of the Rust compiler, the only available definitions are those *within the package being analyzed*. Therefore even with Whole-program, we cannot recurse into e.g. the standard library.

With these three modifications, we compare the number of flows computed from a dataset of Rust projects (Section 5.1) to quantitatively (Section 5.2) and qualitatively (Section 5.3) evaluate the precision of our analysis.

5.1 Dataset

To empirically compare these modifications, we curated a dataset of Rust packages (or "crates") to analyze. We had two selection criteria:

Table 1. Dataset of crates used to evaluate information flow precision, ordered in increasing number of variables analyzed. Each project often contains many crates, so a sub-crate is specified where applicable, and the root crate is analyzed otherwise. Metrics displayed are LOC (lines of code), number of variables, number of functions, and the average number of MIR instructions per function (size of CFG).

Project	Crate	Purpose	LOC	# Vars	# Funcs	Avg. Instrs/Func
rayon		Data parallelism library	15,524	10,607	1,079	16.6
Rocket	core/lib	Web backend framework	10,688	12,040	741	25.5
rustls	rustls	TLS implementation	16,866	23,407	868	42.4
sccache		Distributed build cache	23,202	23,987	643	62.1
nalgebra		Numerics library	31,951	35,886	1,785	26.7
image		Image processing library	20,722	39,077	1,096	56.8
hyper		HTTP server	15,082	44,900	790	82.9
rg3d		3D game engine	54,426	59,590	3,448	25.7
rav1e		Video encoder	50,294	76,749	931	115.4
RustPython	vm	Python interpreter	47,927	97,637	3,315	51.0
		Total:	286,682	435,979	14,696	

1. To mitigate the single-crate limitation of Whole-program, we preferred large crates so as to see a greater impact from the Whole-program modification. We only considered crates with over 10,000 lines of code as measured by the `cloc` utility [12].
2. To control for code styles specific to individual applications, we wanted crates from a wide range of domains.

After a manual review of large crates in the Rust ecosystem, we selected 10 crates, shown in Table 1. We built each crate with as many feature flags enabled as would work on our Ubuntu 16.04 machine. Details like the specific flags and commit hashes can be found in the appendix.

For each crate, we ran the information flow analysis on every function in the crate, repeated under each of the 8 conditions. Within a function, for each local variable x, we compute the size of $\Theta(x)$ at the exit of the CFG — in terms of program slicing, we compute the size of the variable's backward slice at the function's return instructions. The resulting dataset then has four independent variables (crate, function, condition, variable name) and one dependent variable (size of dependency set) for a total of 3,487,832 data points.

Our main goal in this evaluation is to analyze precision, not performance. Our baseline implementation is reasonably optimized — the median per-function execution time was 370.24μs. But Whole-program is designed to be as precise as possible, so its naive recursion is sometimes extremely slow. For example, when analyzing the `GameEngine::render` function of the `rg3d` crate (with thousands of functions in its call graph), the modular analysis takes 0.13s while the recursive analysis takes 23.18s, a 178× slowdown. Future work could compare our modular analysis to whole-program analyses across the precision/performance spectrum, such as in the extensive literature on context-sensitivity [33].

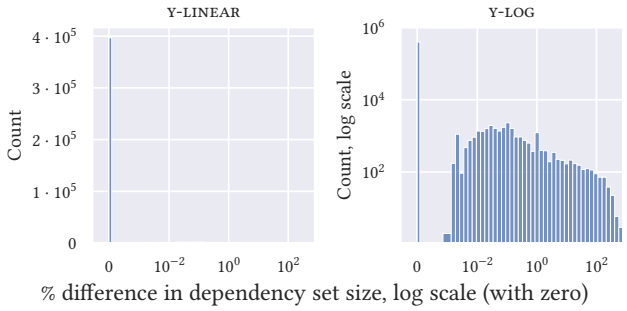

Figure 2. Distribution in differences of dependency set size between Whole-program and Modular analyses. The x-axis is a log-scale with 0 added for comparison. Most sets are the same, so 0 dominates (left). A log-scale (right) shows the tail more clearly.

5.2 Quantitative Results

We observed no meaningful patterns from the interaction of modifications — for example, in a linear regression of the interaction of Mut-blind and Ref-blind against the size of the dependency set, each condition is individually statistically significant ($p < 0.001$) while their interaction is not ($p = 0.337$). So to simplify our presentation, we focus only on four conditions: three for each modification individually active with the rest disabled, and one for all modifications disabled, referred to as Modular.

5.2.1 Whole-program. For Whole-program, we compare against Modular to answer our first evaluation question: how much more precise is a whole-program analysis than a modular one? To quantify precision, we compare the *percentage increase in size* of dependency sets for a given variable between two conditions. For instance, if Whole-program computes $|\Theta(x)| = 2$ and Modular computes

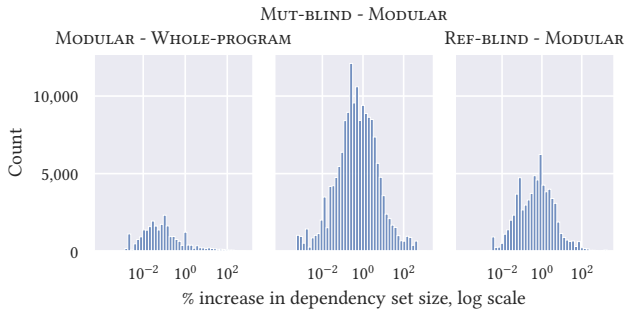

Figure 3. Distribution in differences between Modular and each alternative condition, with zeros excluded to highlight the shape of each distribution. Mut-blind and Ref-blind both reduce the precision more often and more severely than Modular does vs. Whole-program.

$|\Theta(x)| = 5$ for some x, the difference is $(5-2)/2 = 1.5 = 150\%$.

Figure 2 shows a histogram of the differences between Whole-program and Modular for all variables. In 94% of all cases, the Whole-program and Modular conditions produce the same result and hence have a difference of 0. In the remaining 6% of cases with a non-zero difference, visually enhanced with a log-scale in Figure 2-right, the metric follows a right-tailed log-normal distribution. We can summarize the log-normal by computing its median, which is 7%. This means that within the 6% of non-zero cases, the median difference is an increase in size by 7%. Thus, the modular approximation does not significantly increase the size of dependency sets in the vast majority of cases.

5.2.2 Mut-blind and Ref-blind.

Next, we address our second evaluation question: how much less precise is an analysis with weaker assumptions about the program than the Modular analysis? For this question, we compare the size of dependency sets between the Mut-blind and Ref-blind conditions versus Modular. Figure 3 shows the corresponding histograms of differences, with the Whole-program vs. Modular histogram included for comparison.

First, the Mut-blind and Ref-blind modifications reduce the precision of the analysis more often and with a greater magnitude than Modular does vs. Whole-program. 39% of Mut-blind cases and 17% of Ref-blind cases have a non-zero difference. Of those cases, the median difference in size is 50% for Mut-blind and 56% for Ref-blind.

Therefore, the information from ownership types is valuable in increasing the precision of our information flow analysis. Dependency sets are often larger without access to information about mutability or lifetimes.

5.3 Qualitative Results

The statistics convey a sense of how often each condition influences precision. But it is equally valuable to understand the kind of code that leads to such differences. For each condition, we manually inspected a sample of cases with non-zero differences vs. Modular.

5.3.1 Modularity.

One common source of imprecision in modular flows is when functions take a mutable reference as input for the purposes of passing the mutable permission off to an element of the input.

```
fn crop<I: GenericImageView>(
  image: &mut I, x: u32, y: u32,
  width: u32, height: u32
) -> SubImage<&mut I> {
  let (x, y, width, height) =
    crop_dimms(image, x, y, width, height);
  SubImage::new(image, x, y, width, height)
}
```

For example, the function `crop` from the `image` crate returns a mutable view on an image. No data is mutated, only the mutable permission is passed from whole image to sub-image. However, a modular analysis on the `image` input would assume that `image` is mutated by `crop`.

Another common case is when a value only depends on a subset of a function's inputs. The modular approximation assumes all inputs are relevant to all possible mutations, but this is naturally not always the case.

```
fn solve_lower_triangular_with_diag_mut<R2,C2,S2>(
  &self, b: &mut Matrix<N, R2, C2, S2>, diag: N,
) -> bool {
  if diag.is_zero() { return false; }
  // logic mutating b...
  true
}
```

For example, this function from `nalgebra` returns a boolean whose value solely depends on the argument `diag`. However, a modular analysis of a call to this function would assume that `self` and `b` is relevant to the return value as well.

5.3.2 Mutability.

The reason Mut-blind is less precise than Modular is quite simple — many functions take immutable references as inputs, and so many more mutations have to be assumed.

```
fn read_until<R, F>(io: &mut R, func: F)
  -> io::Result<Vec<u8>>
  where R: Read, F: Fn(&[u8]) -> bool
{
  let mut buf = vec![0; 8192]; let mut pos = 0;
  loop {
    let n = io.read(&mut buf[pos..])?; pos += n;
    if func(&buf[..pos]) { break; }
    // ...
  }
}
```

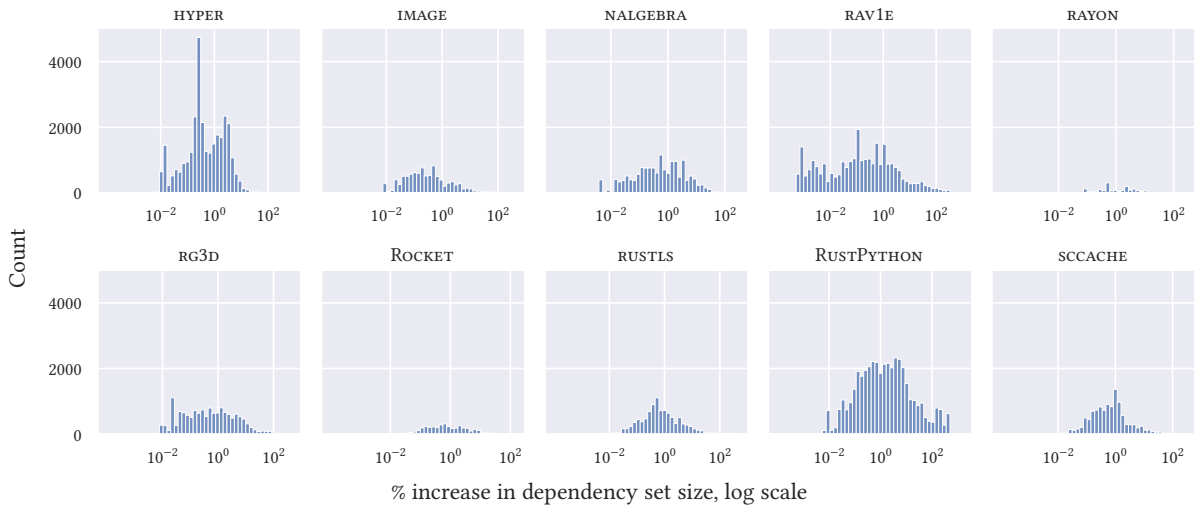

Figure 4. Distribution of non-zero differences between Modular and Mut-blind, broken down by crate.

For instance, this function from `hyper` repeatedly calls an input function `func` with segments of an input buffer. Without a control-flow analysis, it is impossible to know what functions `read_until` will be called with. And so Mut-blind must always assume that `func` could mutate `buf`. However, Modular can rely on the immutability of shared references and deduce that `func` could not mutate `buf`.

5.3.3 Lifetimes. Without lifetimes, our analysis has to make more conservative assumptions about objects that could possibly alias. We observed many cases in the Ref-blind condition where two references shared different lifetimes but the same type, and so had to be classified as aliases.

```
fn link_child_with_parent_component(
  parent: &mut FbxComponent,
  child: &mut FbxComponent,
  child_handle: Handle<FbxComponent>,
) { match parent {
  FbxComponent::Model(model) => {
    model.geoms.push(child_handle),
  },
  // ..
}}
```

For example, the `link_child_with_parent_component` function in `rg3d` takes mutable references to a `parent` and `child`. These references are guaranteed not to alias by the rules of ownership, but a naive pointer analysis must assume they could, so modifying `parent` could modify `child`.

5.4 Threats to Validity

Finally, we address the issue: how meaningful are the results above? How likely would they generalize to arbitrary code rather than just our selected dataset? We discuss a few threats to validity below.

Are the results due to only a few crates? If differences between techniques only arose in a small number of situations that happen to be in our dataset, then our technique would not be as generally applicable. To determine the variation between crates, we generated a histogram of non-zero differences for the Modular vs. Mut-blind comparison, broken down by crate in Figure 4.

As expected, the larger code bases (e.g. rav1e and RustPython) have more non-zero differences than smaller codebases — in general the correlation between non-zero differences and total number of variables analyzed is strong, $R^2 = 0.79$. However variation also exists for crates with roughly the same number of variables like `image` and `hyper`. Mut-blind reduces precision for variables in `hyper` more often than `image`. A qualitative inspection of the respective codebases suggests this may be because `hyper` simply makes greater use of immutable references in its API.

These findings suggest that the impact of ownership types and the modular approximation likely do vary with code style, but a broader trend is still observable across all code.

Would Whole-program be more precise with access to dependencies? A limitation of our whole-program analysis is our inability to access function definitions outside the current crate. Without this limitation, it may be that the Modular analysis would be significantly worse than Whole-program. So for each variable analyzed by Whole-program, we additionally computed whether the information flow for that variable involved a function call across a crate boundary.

Overall 96% of cases reached at least one crate boundary, suggesting that this limitation does occur quite often in practice. However, the impact of the limitation is less clear. Of the 96% of cases that hit a crate boundary, 6.6% had a non-zero difference between Modular and Whole-program. Of the

```
let mut f: File =
  File::create(path: "file.txt").unwrap();
let contents: &[u8; 11] = b"Hello world";
f.write_all(buf: contents).unwrap();

let meta: Metadata = f.metadata().unwrap();
println!("{:?}", f);
println!("{:?}", meta);
```

```
// let start = Instant::now();
run_expensive_calculation();
// let elapsed = start.elapsed();
// println!("Elapsed: {}s", elapsed.as_secs())
```

(a) A program slicer integrated into VSCode. Above, the user selects a slicing criterion like the variable `f`. Then the slicer highlights the criterion in green, and fades out lines that are not part of the backward slice on `f`. For example, `write_all` mutates the file so it is in the slice, while `metadata` reads the file so it is not in the slice. Below, the user can manipulate aspects of a program such as commenting out code related to timing. The user computes a forward slice on `start`, adds this slice to their selection (in blue), then tells the IDE to comments out all lines in the selection.

```
use ifc_traits::{insecure_print, Secure};
struct Password(&'static str);
impl Secure for Password {}

fn main() {
  let password = Password(include_str!("secret"));
  if password.0 == "hello" {
    insecure_print!("Hello world!");
  }
}
```

```
~/C/f/e/password ❯❯❯ cargo ifc
ERROR: insecure flow in main.rs from data at 6:6-6:14:
  password
to data at 8:4-8:36:
  <in macro expansion>
at the instruction 8:4-8:36:
  insecure_print!("Hello world!");
```

(b) An IFC checker. Above, the `ifc_traits` library exports a `Secure` for users to mark sensitive data, like `Password`, and insecure operations like `insecure_print`. Below, a compiler plugin invoked on the program checks for information flows from data with a type implementing `Secure` to insecure operations. Here `insecure_print` is conditionally executed based on a read from `PASSWORD`, so this flow is flagged.

Figure 5. Two applications of information flow built using Flowistry.

4% that did not hit a crate boundary, 0.6% had a non-zero difference. One would expect that Whole-program would be the most precise when the whole program is available (no boundary), but instead it was much closer to Modular.

Ultimately it is not clear how much more precise Whole-program would be given access to all a crate's dependencies, but it would not necessarily be a significant improvement over the benchmark presented.

Is ownership actually important for precision? The finding that Ref-blind only makes a difference in 17% of cases may seem surprisingly small. For instance, Shapiro and Horwitz [32] found in a empirical study of slices on C programs that "using a pointer analysis with an average points-to set size twice as large as a more precise pointer analysis led to an increase of about 70% in the size of [slices]."

A limitation of our ablation is that the analyzed programs were written to satisfy Rust's ownership safety rules. Disabling lifetimes does not change the structure of the programs to become more C-like — Rust generally encourages a code style with fewer aliases to avoid dealing with lifetimes. A fairer comparison would be to implement an application idiomatically in both Rust and Rust-but-without-feature-X, but such an evaluation is not practical. It is therefore likely that our results understate the true impact of ownership types on precision given this limitation.

6 Applications

We have demonstrated that ownership can be leveraged to build an information flow analysis that is static, modular, sound, and precise. Our hope is that this analysis can serve as a building block for future static analyses. To bootstrap this future, we have used Flowistry to implement prototypes of a program slicer and an IFC checker, shown in Figure 5.

The program slicer in Figure 5a is a VSCode extension that fades out all lines of code that are not relevant to the user's selection, i.e. not part of the modular slice. Rather than present a slice of the entire program like in prior slicing tools, we can use Flowistry's modular analysis to present lightweight slices of just within a given function. Users can apply the slicer for comprehension tasks such as reducing the scope of a bug, or for refactoring tasks such as removing an aspect of a program like timing or logging.

The IFC checker in Figure 5b is a Rust library and compiler plugin. It provides the user a library with the traits `Secure` and `Insecure` to indicate the relative security of data types and operations. Then the compiler plugin uses Flowistry to determine whether information flows from `Insecure` variables to `Secure` variables. Users can apply the IFC checker to catch sensitive data leaks in an application. This prototype is purely intraprocedural, but future work could build an interprocedural analysis by using Flowistry's output as procedure summaries in a larger information flow graph.

7 Related Work

Our work draws on three core concepts: information flow, modular static analysis, and ownership types.

Information flow. Information flow has been historically studied in the context of security, such as ensuring low-security users of a program cannot infer anything about its high-security internals. Security-focused information flow analyses have been developed for Java [25], Javascript [4], OCaml [27], Haskell [34], and many other languages.

Each analysis satisfies some, but not all, of our requirements from Section 1. For instance, the JFlow [25] and Flow-Caml [27] languages required adding features to the base language, violating our second requirement. Some methods like that of Austin and Flanagan [4] for Javascript rely on dynamic analysis, violating our third requirement. And Haskell only supports effects like mutation through monads, violating our first requirement.

Nonetheless, we draw significant inspiration from mechanisms in prior work. Our analysis resembles the slicing calculus of Abadi et al. [1]. The use of lifetimes for modular analysis of functions is comparable to security annotations in Flow-Caml [27]. The CFG analysis draws on techniques used in program slicers, such as the LLVM dg slicer [7].

Modular static analysis. The key technique to making static analysis modular (or "compositional" or "separate") is symbolically summarizing a function's behavior, so that the summary can be used without the function's implementation. Starting from Rountev et al. [29] and Cousot and Cousot [10], one approach has been to design a system of "procedure summaries" understood by the static analyzer and distinct from the language being analyzed. This approach has been widely applied for static analysis of null pointer dereferences [40], pointer aliases [13], data dependencies [35], and other properties.

Another approach, like ours, is to leverage the language's type system to summarize behavior. Tang and Jouvelot [36] showed that an effect system could be used for a modular control-flow analysis. Later work in Haskell used its powerful type system and monadic effects to embed many forms of information flow control into the language [6, 23, 30, 34].

Ownership types. Rust and Oxide's conceptions of ownership derive from Clarke et al. [8] and Grossman et al. [18]. For instance, the Cyclone language of Grossman et al. uses regions to restrict where a pointer can point-to, and uses region variables to express relationships between regions in a function's input and output types. A lifetime is similar in that it annotates the types of pointers, but differs in how it is analyzed.

Recent works have demonstrated innovative applications of Rust's type system for modular program analysis. Astrauskas et al. [3] embed Rust programs into a separation logic to verify pre/post conditions about functions. Jung et al. [20] use Rust's ownership-based guarantees to implement more aggressive program optimizations.

Closer to our domain, Balasubramanian et al. [5] implemented a prototype IFC system for Rust by lowering programs to LLVM and verifying them with SMACK [28], although their system is hard to contrast with ours given the high-level description in their paper. Njor and Gústafsson [26] implemented a static taint analysis for Rust, although it is not field-sensitive, alias-sensitive, or modular.

8 Discussion

Looking forward, two interesting avenues for future work on Flowistry are improving its precision and addressing soundness in unsafe code. For instance, the lifetime-based pointer analysis is sound but imprecise in some respects. Lifetimes often lose information about part-whole relationships. Consider the function that returns a mutable pointer to a specific index in a vector:

```
fn get_mut<'a>(&'a mut self, i: usize)
  -> Option<&'a mut T>;
```

These lifetimes indicate only that the return value points to *something* in the input vector. The expressions `v.get_mut(i)` and `v.get_mut(i + 1)` are considered aliases even though they are not. Future work could integrate Flowistry with verification tools like Prusti [3] to use abstract interpretation for a more precise pointer analysis in such cases.

Additionally, Rust has many libraries built on unsafe code that can lose annotations essential to information flow, such as interior mutability. For example, shared-memory concurrency in Rust looks like this:

```
1 let n: Arc<Mutex<i32>> = Arc::new(Mutex::new(0));
2 let n2: Arc<Mutex<i32>> = Arc::clone(&n);
3 *n2.lock().unwrap() = 1;
```

`Arc::clone` does not share a lifetime between its input and output, so a lifetime-based pointer analysis therefore cannot deduce that `n2` is an alias of `n`, and Flowistry would not recognize that the mutation on line 3 affects `n`. Future work can explore how unsafe libraries could be annotated with the necessary metadata needed to analyze information flow, similar to how RustBelt [21] identifies the pre/post-conditions needed to ensure type safety within unsafe code.

Overall, we are excited by the possibilities created by having a practical information flow analysis that can run today on any Rust program. Many exciting systems for tasks like debugging [22], example generation [19], and program repair [39] rely on information flow in some form, and we hope that Flowistry can support the development of these tools.

Acknowledgments

This work was partially supported by the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019).

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (nov 2020), 27 pages. https://doi.org/10.1145/3428204

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (oct 2019), 30 pages. https://doi.org/10.1145/3360573

[4] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (Dublin, Ireland) *(PLAS '09)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1554339.1554353

[5] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 156–161. https://doi.org/10.1145/3102980.3103006

[6] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/2784731.2784758

[7] Marek Chalupa. 2016. *Slicing of LLVM bitcode*. Master's thesis. Masaryk University.

[8] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

[9] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[10] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, Heidelberg, 159–178.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[12] Al Danial. 2021. cloc: Count Lines of Code. https://github.com/AlDanial/cloc

[13] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 567–577. https://doi.org/10.1145/1993498.1993565

[14] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[16] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

[17] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/sp.1982.10014

[18] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

[19] Andrew Head, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. 2018. Interactive Extraction of Examples from Existing Code. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173659

[20] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. https://doi.org/10.1145/3371109

[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

[22] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

[23] Peng Li and S. Zdancewic. 2006. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 12 pp.–16. https://doi.org/10.1109/CSFW.2006.13

[24] Niko Matsakis. 2017. Non-lexical lifetimes. https://rust-lang.github.io/rfcs/2094-nll.html

[25] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

[26] Emil Jørgensen Njor and Hilmar Gústafsson. 2021. *Static Taint Analysis in Rust*. Master's thesis. Aalborg University.

[27] François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (jan 2003), 117–158. https://doi.org/10.1145/596980.596983

[28] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[29] Atanas Rountev, Barbara G. Ryder, and William Landi. 1999. Data-Flow Analysis of Program Fragments. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 235–252.

[30] Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-Weight Information-Flow Security in Haskell. In *Proceedings of*

the First ACM SIGPLAN Symposium on Haskell (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1411286.1411289

[31] A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[32] Marc Shapiro and Susan Horwitz. 1997. The Effects of the Precision of Pointer Analysis. In *Proceedings of the 4th International Symposium on Static Analysis (SAS '97)*. Springer-Verlag, Berlin, Heidelberg, 16–34.

[33] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[34] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2034675.2034688

[35] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 83–95. https://doi.org/10.1145/2676726.2676997

[36] Yan Mei Tang and Pierre Jouvelot. 1994. Separate abstract interpretation for control-flow analysis. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 224–243. https://doi.org/10.1007/3-540-57887-0_98

[37] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449.

[38] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The Essence of Rust. arXiv:arXiv:1903.00982v3

[39] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[40] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1328438.1328467