Static Taint Analysis in Rust

Using Rusts Ownership System for Precise Static Analysis

Emil Jørgensen Njor, Hilmar Gústafsson

Distributed Systems
ds103f21
June 10, 2021

Master's Thesis

Department of Computer Science
Aalborg University
https://www.cs.aau.dk/

Title:
Static Taint Analysis in Rust – Using Rusts Ownership System for Precise Static Analysis

Theme:
Distributed Systems

Project Period:
Spring 2021

Project Group:
ds103f21

Participant(s):
Emil Jørgensen Njor, Hilmar Gústafsson

Supervisor(s):
René Rydhof Hansen, Danny Bøgsted Poulsen

Page Numbers: 45

Date of Completion:
June 10, 2021

Abstract:

The Rust programming language employs a ownership system that tackles the aliasing issue, a significant source for imprecision in ordinary static analyses. Theoretically, a static analysis should therefore be more precise in Rust than in languages without the ownership system. We explore how the ownership system can make an analysis more precise in practice.
To do so, we formally define a subset of Mid-level Intermediate Representation (MIR), and a static taint analysis based on that formalization. We implement a tool which is based on the Rust compiler to demonstrate the formalized taint analysis.
We confirm that the ownership system makes it possible to create a more precise taint analysis due to the aliasing restrictions.

Summary

We document a master's thesis project by Emil Njor and Hilmar Gústafsson, written and worked on from the 1st of February 2021 to the 11th of June 2021.
We investigate how the ownership system of the Rust programming language could be used to improve the precision of a static taint analysis. To investigate this, we describe static analysis, specifically the theory of dataflow analysis. We also investigate the Rust programming language and its compilation process.
We find a central Intermediate Representation (IR) during the compilation process called MIR to be suitable for static analysis, and choose it as the language for which to design the static taint analysis. As no official syntax and semantics exist for MIR, we formalise a subset of the language to base the analysis on. We define a simple static taint analysis without concerning itself with the ownership system of Rust, to form a base analysis to expand upon. Afterwards, we add Rust's ownership system to the syntax and semantics, and explore its implications on the simple static taint analysis.
We confirm that the ownership system makes it possible to create a more precise taint analysis due to the aliasing restrictions.
In addition to this report, we provide an implementation of the analysis presented in this report, which can be found in its repository on GitHub.

Contents

Glossary

CFG Control Flow Graph. 8, 9, 10, 11, 17, 18, 23, 24, 25, 26, 27

HIR High-level Intermediate Representation. 33, 35

ICFG Interprocedural Control Flow Graph. 10, 17, 26

IR Intermediate Representation. ii, 2, 17

MIR Mid-level Intermediate Representation. i, ii, 2, 3, 17, 18, 19, 20, 21, 23, 24, 25, 26, 28, 30, 31, 33, 34, 35, 36, 37, 38, 42, 43

Chapter 1

Introduction

User input is the most common way to exploit programs in the world today, according to OWASP [7]. An important task for programmers is then to make sure that inputs do not cause unintended behaviour in the program or supporting systems. A common example of unintended behaviour is the case of the *SQL injection* attack.

Example 1.1 – SQL Injection

```
1  a = input();
2  b = query_database("SELECT * FROM Users WHERE Id = " + a);
```

In the first line, the program takes input from an untrusted source, it could be some publicly accessible webpage. In the second line, the program uses this input to query a database. For the majority of cases, this will work well, as users of the webpage will input their username, and the server will respond with the row corresponding to that username. However, a malicious user could input "`whatever'; DROP TABLE Users`", which would delete the entire "Users" table from the database.
The external input has not been checked for symbols which have a specific significance in SQL, such as quotes or the semicolon. This allows SQL queries to contain a sequence of statements. With the following adjustment to the program, this particular attack will no longer be possible (However, many other attacks exist — too many for us to exhaustively consider here).

Example 1.2 – SQL Injection Partly Sanitised

```
1  a = input();
2  b = sanitise(a);
3  c = query_database("SELECT * FROM Users WHERE Id = " + b);
```

In general, we want to *sanitise* external input before using it in a critical section of a program. When we sanitise input, we remove or modify parts of the input that might cause unintended behaviour if used later in the program. A large effort has been put into educating programmers to write code that do not contain vulnerabilities such as this, but despite the effort it is still prevalent [7].
Instead of relying on educating every programmer, imagine that we could make compilers automatically reject programs that exhibit these flaws. We can achieve this by using *static analysis* to implement what is called a *taint analysis*. During taint analysis we consider taint as data in a program that has been affected by external input. The analysis involves keeping track of how taint originates from *taint sources*, spreads to other variables, and finally becomes sanitised or flow into *taint sinks*.

Example 1.3 – Taint Source and Sink
In the example that we considered earlier, the taint source would be the `input()`, and the taint sink would be the `query_database()`.

If any taint flows into a taint sink there is a risk that the taint may cause unintended behaviour in the program, and the compiler should reject the program. A known limitation of static analyses is the undecidability of non-trivial properties, as stated in Rice's theorem [22]. Practically, this means that an analysis must approximate its answer.

The Rust programming language is an excellent candidate for a taint analysis, as it has an ownership system which enforces strict aliasing rules, and should improve the precision of static taint analysis. By making such language constraints, the analysis approximates less. In this report we outline our efforts to introduce such a static taint analysis into the Rust. Rust also makes a distinction between mutable and immutable bindings, which could also improve the precision of a static analysis.

The Rust language goes through several IRs when being compiled. One of them called MIR, is especially well suited for dataflow analysis. This is discussed further in Chapter 4.

We therefore propose the following problem statement: *How does the precision of static taint analysis benefit from the programming model in the Rust Programming Language?* To understand this, we:

- Define a formal syntax and semantics for a subset of MIR.
- Define a formal taint analysis based on the syntax and semantics.
- Implement a tool which runs the defined analysis on real Rust code.

To our knowledge, this is the first attempt at formalising the semantics of MIR.

The report is structured as follows. In Chapter 2 we introduce papers and work from the literature that draw similarities to this work. Chapter 3 introduces the static analysis theory necessary to design a static taint analysis. In Chapter 4 we give a deeper introduction to the Rust programming language, and its compilation process. Chapter 5 defines the basic syntax and semantics of Rust's MIR. We use this in Chapter 6 to design a simple static taint analysis for MIR. In Chapter 7 we then expand the syntax and semantics of MIR to include Rusts special ownership system, and similarly expand the analysis to also consider the ownership system. Chapter 8 highlights key components of an implementation of the presented analysis. We then discuss this work and potential future work in Chapter 9, and present our conclusions in Chapter 10

Chapter 2

Related Work

A lot of work has already been done on analysis of Rust. We mention the larger contributions. MIRI [3] is an experimental interpreter for Rust's MIR. It can detect certain cases of undefined behaviour in unsafe Rust. As such it is a dynamic analysis contrary to the static analysis approach which we take.
MIRAI [2] is a static analysis tool for Rust. While it supports many types of analysis including taint analysis, the tool is external to the Rust compiler, and thus require a non-trivial setup process to analyse Rust programs.
In the paper "A few billion lines of code later", employees at Coverity, a company built around a static and dynamic analysis tool, describe the importance of analysis tools being easy to use for the end user in order to be widely adopted [13]. The same point is made by engineers at Google [23].
Prusti [8] is another static analysis tool for Rust. By default, it verifies the absence of panics by verifying that the "unreachable!()" and "panic!()" statements are unreachable. An additional option to enable overflow checking for integers is available. Otherwise, Prusti relies on annotations inserted by the programmer to verify preconditions, postconditions and loop invariants. The Prusti tool is also not a part of the Rust compiler, and must therefore be run as a separate tool. The easiest way to do so is to use the "Prusti Assistant" extension for the Visual Studio Code text editor.
In their paper "Systems Programming in Rust: Beyond Safety", Balasubramanian et al. discuss the advantages of Rust that go beyond its safety guarantees. One of their main points is that Rust allows for efficient static information flow analysis, due to its ownership model preventing pointer aliasing issues [12]. While aliasing is not an issue in safe rust, as soon as unsafe code blocks are involved, aliasing becomes an issue again. This was recognized by Ralf Jung et al. in their paper "Stacked Borrows: An aliasing model for Rust", where they propose a semantics for Rust that makes Rust programs using unsafe code undefined behaviour if they break safe aliasing rules [16].
Static taint analyses have already been proposed in other programming languages than Rust. In [19] a taint analysis for the Java programming language is proposed. Another static taint analysis is Pysa developed by Facebook for the Python programming language [9].
To be able to design any analysis, it is necessary to know the semantics of the underlying programming language. While no official semantics for Rust or MIR exist, some effort has been put into formalising the languages. Oxide [26] is one such attempt at formalising Rust, in which the authors formalise a programming language, which they claim captures the essence of Rust. Krust [25] is an attempt at formalising a subset of Rust. The semantics of the subset is implemented in K, an executable semantic framework, which provides a formal interpreter and verification tools for the language.

Chapter 3

Static Analysis

This chapter describes the theory behind static program analysis. As taint analysis is concerned with the flow of taint through a program, we focus on dataflow analysis, a type of static analysis which considers how the state of a program changes through its statements. To motivate the use of static analysis, we consider the following quote by Edsger Dijkstra:

> "Program testing can be used to show the presence of bugs, but never to show their absence!" [6]

Example 3.1 – Buggy Program
To demonstrate the principle, the following program will successfully run on all input except for 7:

```
1  main (input){
2      if(input == 7){
3          0 / 0
4      }
5      ...
6  }
```

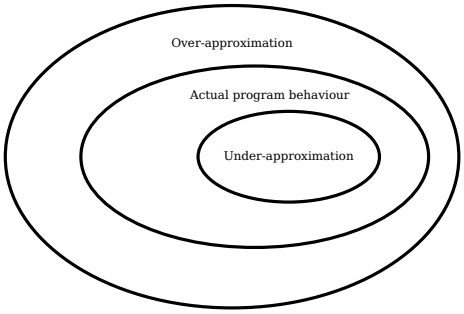
If the input range of this program is the set of all natural numbers $\mathbb{N}$, then we have an infinite amount of inputs to reason about. This makes it clear that we cannot use program testing to exhaustively reason about every run of a program. To do so, we need to turn to static program analysis.
Static analysis is, in contrast to dynamic program analysis, the method of reasoning about programs without running them [20]. It has uses ranging from optimizing compilers for producing efficient code, to programs that can automatically detect errors in programs. While a dynamic analysis can only reason about the observed execution of programs, static analysis allows reasoning about all possible executions over a program [15]. A static analysis is often used in the form of a program analyser, a program that takes other programs as its input, and outputs the result of the analysis.
Rice's theorem from 1953 has important implications for the usefulness of static analysis.

Theorem 3.1 – Rice's Theorem
In all Turing-complete languages, all non-trivial semantic properties of programs are undecidable [22].

A trivial semantic property of a program is a property which is either true for every program or false for every program. As a result, we are not able to automatically reason about the behaviour of programs without introducing approximations. The results of these approximations are that static analyses will either be able to show the presence of some property, or answer maybe.
A static analysis that over-approximates actual program behaviour will be able to show that a property is not in the actual program behaviour. In other words, an over-approximating static analysis will never report false negative results, it may however, as a consequence of Theorem 3.1, produce false positive results. On the other hand, a static analysis that under-approximates actual program behaviour will be able to show that

a property is in the actual program behaviour. This means that an under-approximating static analysis will never produce false positive results, and may produce false negative results.

Example* 3.2 – *Over-Approximations and Under-Approximations



We include a sketch of how over-approximations and under-approximations work. The middle ellipse is a representation of the actual behaviour of a program. An over-approximation will completely envelop the behaviour of the program, and also include behaviour not actually in the program. Dually, an under-approximation will be completely enveloped by the behaviour of the actual program, but not capture all behaviour in the program.

In practice, many static analyses are not sound, as sound analyses tend to either over-approximate or under-approximate program behaviour past usefulness. An unsound static analysis is not useless, as it can still be used to locate bugs in a program.

Definition* 3.1 – *Soundness
We say that a static analysis is sound if it either completely over-approximates or completely under-approximates the actual program behaviour.

As stated in the beginning of the chapter, we focus on the dataflow analysis approach to static analysis. However, before we delve into that topic, we introduce the mathematical concept of lattices, which is necessary to understand further topics.

3.1 Lattice Theory

Lattice theory is grounded in *order theory*, specifically the application of *partial orders* to sets. Most definitions here are based on definitions by Møller and Schwartzbach [20].

Definition* 3.2 – *Partially Ordered Set
A *partially ordered set* is a set S equipped with a binary relation $\sqsubseteq$ where the following conditions are satisfied:

Reflexivity: $\forall x \in S : x \sqsubseteq x$
Transitivity: $\forall x, y, z \in S : x \sqsubseteq y \land y \sqsubseteq z \implies x \sqsubseteq z$
Anti-symmetry: $\forall x, y \in S : x \sqsubseteq y \land y \sqsubseteq x \implies x = y$ [21]

For partially ordered sets we introduce the concepts of an *upper bound* and a *lower bound*.

Definition* 3.3 – *Upper and Lower Bounds
Let $X \subseteq S$.
$y \in S$ is an upper bound of X written $X \sqsubseteq y$ if $\forall x \in X, x \sqsubseteq y$.
$y \in S$ is a lower bound of X written $y \sqsubseteq X$ if $\forall x \in X, y \sqsubseteq x$.

Note that we define what the relation means when applied to an element and a set, whereas the previous definition was only concerned with the relation between two elements in a set.

We also introduce the *least upper bound* and the *greatest lower bound*.

Definition* 3.4 – *Least Upper Bound and Greatest Lower Bound
We denote the set of all upper bounds for X as $Up(X)$ and the set of all lower bounds for X as $Lw(X)$
The least upper bound of X written $\bigsqcup X$, is an element $lup \in Up(X)$ where $lup \sqsubseteq Up(X)$.
The greatest lower bound of X written $\bigsqcap X$ as an element $glw \in Lw(X)$ where $Lw(X) \sqsubseteq glw$.

Not every partially ordered set has a greatest lower bound, or a least upper bound. For a pair of elements (x, y), we refer to the least upper bound as the *join* with the infix notation $x \sqcup y$, equivalent to $\bigsqcup\{x, y\}$.
For a similar pair of elements (x, y), we refer to the greatest lower bound as the *meet* with the infix notation $x \sqcap y$, equivalent to $\bigsqcap\{x, y\}$.
Equipped with these concepts, we define a *Lattice*.

Definition* 3.5 – *Lattice
A *lattice* is a partially ordered set in which $x \sqcup y$ and $x \sqcap y$ exist $\forall x, y \in S$.

We also define two lattices, namely a *bounded lattice* and a *complete lattice*.

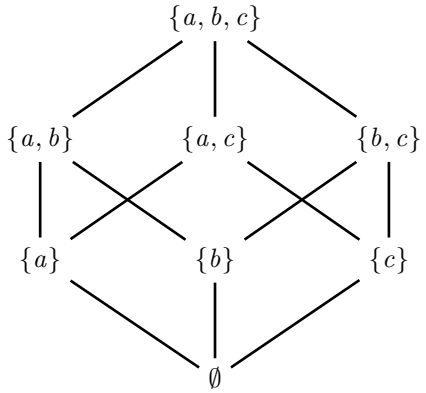
Definition* 3.6 – *Bounded Lattice
A *bounded lattice* is a lattice which has a unique largest element, usually called *"top"* and denoted $\top$, and a unique smallest element, usually called *"bottom"* and denoted $\bot$.

Definition* 3.7 – *Complete Lattice
A *complete lattice* is a partially ordered set in which $\bigsqcup X$ and $\bigsqcap X$ exist $\forall X \subseteq S$. Every finite non-empty lattice is complete, and every complete lattice is also a bounded lattice.

A commonly used lattice is the *powerset lattice*.

Definition* 3.8 – *Powerset Lattice
For a set A we can define a complete lattice $(P(A), \subseteq)$, where $\bot = \emptyset, \top = A, x \sqcup y = x \cup y$, and $x \sqcap y = x \cap y$

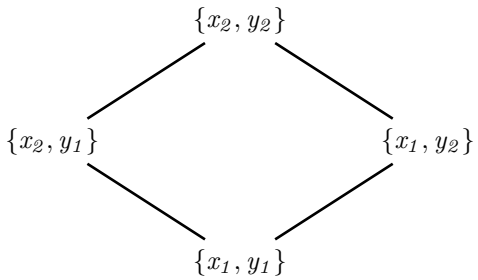
Example* 3.3 – *Powerset Lattice
For the set $\{a, b, c\}$, the powerset lattice, denoted as $\mathcal{P}(\{a, b, c\})$, or the tuple $(\{a, b, c\}, \subseteq)$ would look as follows.



Another construct that is commonly used is the *product lattice*.

Definition* 3.9 – *Product Lattice
If $L_1, L_2, \ldots, L_n$ are complete lattices, then so is $L_1 \times L_2 \times \cdots \times L_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in L_i\}$, which we call a *product lattice*.
In a product lattice $\sqsubseteq$ is defined component wise: $(x_1, x_2, \ldots, x_n) \sqsubseteq (x_1^{'}, x_2^{'}, \ldots x_n^{'}) \iff \forall i = 1, 2, \ldots, n : x_i \sqsubseteq x_i^{'}$.

Example* 3.4 – *Product Lattice
Let L_1 be the lattice with elements $\{x_1, x_2\}$ where $x_1 \sqsubseteq x_2$. Similarly, let L_2 be the lattice with elements $\{y_1, y_2\}$ where $y_1 \sqsubseteq y_2$. The product lattice given by $L_1 \times L_2$ is visualised below.



In general product lattices can be challenging to visualise, especially as they grow past $n \leq 3$.

The product of n identical lattices can be written as P^n.
A *map lattice* is a lattice that maps from a set to a complete lattice.

Definition* 3.10 – *Map Lattice
Let A be a set and L a complete lattice. $A \to L$ is then a map lattice defined as:
$A \to L = \{[a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots] | A = \{a_1, a_2, \ldots\} \land x_1, x_2, \cdots \in L\}$
In a map lattice $\sqsubseteq$ is defined as: $f \sqsubseteq g \iff \forall a_i \in A : f(a_i) \sqsubseteq g(a_i)$ where $f, g \in A \to L$

3.1.1 Functions and Fixed Points

During dataflow analysis, a lattice represents an abstract domain from which we wish infer abstract states at certain parts of a program. Usually this lattice will have the most precise information as $\bot$ and the least precise information as $\top$. Whenever control flow merges, we use the least upper bound to get the most precise information that we can infer after the merge. To ensure that dataflow analyses eventually terminate, all functions must be monotone. A function over lattices is monotone if the output of the function does not get less precise from more precise input.

Definition* 3.11 – *Monotone Function
A function f is monotone, or *order preserving*, if

$$\forall x, y \in L : x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$$

To analyse a program, we establish an equation system for each variable in each state in the program. Informally, the equation system for a variable x in a state s describes the constraints that we put on the variable x immediately after state s.

Definition* 3.12 – *Equation System
Let L be a complete lattice. An equation system over L is of the form:

$$
\begin{aligned}
x_1 &= f_1(x_1, \ldots, x_n) \\
x_2 &= f_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &= f_n(x_1, \ldots, x_n)
\end{aligned}
$$

Here $x_1, \ldots, x_n$ are variables and $f_1, \ldots, f_n : L^n \to L$ are functions, called constraint functions. A solution to an equation system provides a value from L to each variable such that all equations are satisfied.

We can combine the n functions from definition 3.12 into a single $f : L^n \to L^n$:

$$f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n))$$

A solution to this equation system can be found by finding a *fixed point* of the function.

Definition* 3.13 – *Fixed Point
A fixed point for a function f is an input x for which $x = f(x)$.

We get the most precise solution by finding the least fixed point.

Definition* 3.14 – *Least Fixed Point
For a lattice L, the least fixed point is a fixed point x where $x \sqsubseteq y$ for every fixed point y of f.

For a program with only a single control flow, such fixed points are easy to find. However, with the introduction of iterative control structures and branching, this becomes much more complex. We know, however, from Kleene's Theorem [18], that it does exist. The theorem also shows how the least fixed point can be computed. It states:

Theorem* 3.2 – *Kleene's Theorem
In a complete lattice L with finite height, every monotone function $f : L \to L$ has a unique least fixed point denoted $lfp(f)$ defined as:

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

Fixed point algorithms employ these principles, some of which are described in [20] and [21]. Equipped with our new knowledge about lattices we can move on to the topic of dataflow analysis.

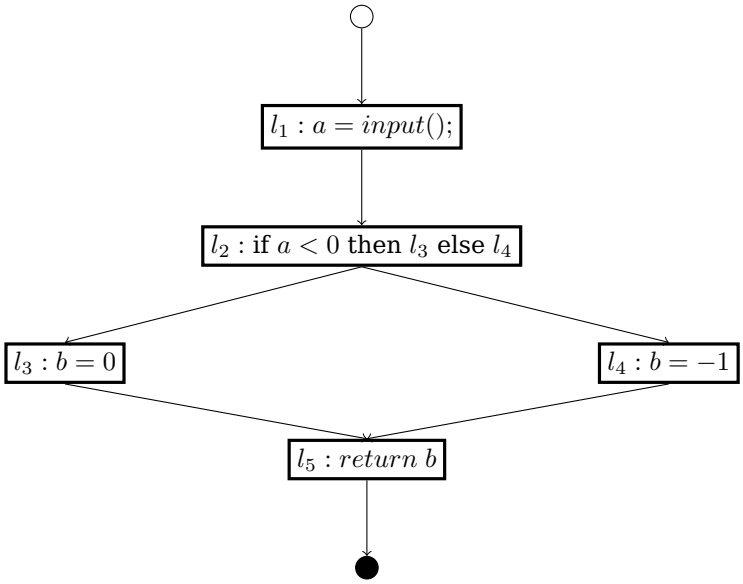
3.2 Dataflow Analysis

Dataflow analysis is a type of static analysis that, as the name implies, is concerned with the flow of data through a program. To conduct such an analysis, it is necessary to first establish an abstraction of program states. In this report we adopt the approach from [20] and represent programs using Control Flow Graphs (CFGs).

Definition* 3.15 – *Control Flow Graph (CFG)
A control flow graph is a directed graph $G = (V, E)$ where V are statements in a program and E represent possible flow of control.

Note that some literature composes several statements, without diverging control flow, into what is called *basic blocks*, and uses those as nodes V in the control flow graph. It is straightforward to combine statements without diverging control flow into basic blocks, by simply concatenating the operations of the statements. We label the nodes V in the CFG to be able to refer to a single node, and use the notation l_n as a label for a node. The set of all labels in a cfg is called $Labels$, and we have that $l_n \in Labels$. We furthermore require that that no two different CFG nodes have the same l_n. Another common addition to a CFG which we adopt is the *initial* and *terminal* node. The initial node has a directed edge into the first statement of the program, and all statements which terminate the program have a directed edge into the terminal node.

Example 3.5 – Control Flow Graph



In this example, the initial node is drawn as a circle with white filling, and the terminal node is drawn as a circle with black filling.

We define two functions $PRED(l)$ and $SUCC(l)$ which respectively output the immediate predecessors and successors to the input node. Immediate predecessors and successors are nodes that can be found by following a single edge. We give an example of the uses of these functions on the CFG from above.

Example 3.6 – PRED and SUCC Functions
For l_2 we have that $PRED(l_2) = \{l_1\}$ and $SUCC(l_2) = \{l_3, l_4\}$.
For l_5 which have $PRED(l_5) = \{l_3, l_4\}$ and $SUCC(l_5) = \{terminal\}$.

Dataflow analyses can be expressed in the form of a *monotone framework*. A monotone framework consists of a complete lattice with finite height and a set of transfer functions. The complete lattice may be fixed for a given analysis, or parameterized by the programs being analysed [20]. One common parameterized lattice is the powerset lattice over all variables in a program.
The transfer functions, written f_t, are functions that specify how a CFG node takes its $input$ and transforms it into its $output$. Over lattices, transfer functions are formally defined as: $f_t : L \to L$.
Combining a CFG and a monotone framework gives us an analysis with the following equations

$$input(l_n) = \bigsqcup_{l'_n \in PRED(l_n)} output(l'_n)$$
$$output(l_n) = f_t(input(l_n))$$

Where f_t is the transfer function for the statement corresponding to the CFG node.
To be able to start an analysis, it is also necessary to define an initial value, which will be the input to the initial node. This initial value will typically be the $\bot$ value of the lattice.
One way to differentiate dataflow analyses are forward analyses and backward analyses. Forward analyses that start from the beginning of a CFG and move towards the end, and backwards analyses that start at the end of a CFG and move towards the start. To convert a forwards analysis into a backwards analysis it suffices to invert each directed edge in the CFG, and exchange the initial and terminal nodes.
A second way to differentiate dataflow analyses are may and must analyses. The term is typically used in relation to powerset lattices, but it is possible to extend the idea to other lattices. A may analysis will identify properties that may be fulfilled at a certain program point, such as a binding maybe being tainted. As such it corresponds to an over-approximating analysis. Dually, a must analysis will identify properties that must be fulfilled at a certain program point, such as a binding must be tainted. As such it corresponds to an under-approximating analysis.

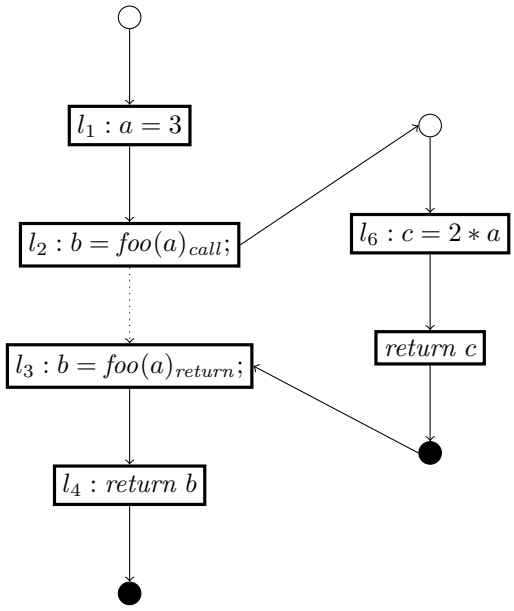
3.2.1 Interprocedural Dataflow Analysis

An analysis which is only applied to a single function is an ***Intra****procedural Dataflow Analysis*. Most programs, however, consist of more than a single function. An analysis which follows the flow of data past function boundaries is an ***Inter****procedural Dataflow Analysis*, which begins with the concept of Interprocedural Control Flow Graphs (ICFGs).
We consider dataflow analysis from the perspective of context-sensitivity, which affects both the runtime performance of an analysis and its precision.

Context-Insensitive Analysis

When an analysis does not differentiate between different calls to the same function, we call it *context-insensitive*. This approach is less precise, since it will unify the results of all calls to a given function. For example, consider the code in Example 3.8. While the first call to `process` submits a tainted argument, and therefore receives tainted output, the second call submits an untainted value. In a context-insensitive analysis, we would join the results of both calls to be the same, e.g., as tainted in a taint analysis.
We give an example of how to implement such an analysis as follows. In an ICFG, we first construct CFGs for each function in a program. We represent a function call as two separate nodes, one is the *call* node, and one the *return* node. All nodes that would be in $PRED$ of a non split function call CFG node will now be in $PRED(call)$, and nodes that would have been in $SUCC$ are now in $SUCC(return)$. The call node will have a directed edge to the initial node of the called function's CFG, and the terminal node of that CFG will have a directed edge back to the corresponding return node.

Example 3.7 – Interprocedural Control Flow Graph



If the same function is called from two different places in the program, this approach will join the two call values and the two return values. As a consequence, one function call will get a returned state that's not only based on its own call, but also all other calls of the same function in the program [21].

Example 3.8 – Context-Sensitivity Problem
An extended version of Example 1.1, we introduce the following:

- `input_t`: This is the initial input received from an external source. It is tainted.
- `input_u`: This is the sanitized version of `input_t`. It is untainted.
- `process`: This function returns a processed version of its input without removing possible SQL injection attacks. The taint status of the output depends on the taint status of the input.

```
1 input_t = input();
2 input_u = sanitize(input_t);
3
4 processed_t = process(input_t);
5 processed_u = process(input_u);
6
7 users = query_database("SELECT * FROM Users WHERE Id = " + processed_u);
8
```

Context-Sensitive Analysis

When an analysis differentiates between function calls based on a given context, we call it *context-sensitive*. There are two classical approaches to making an analysis context-sensitive, the *call strings* and *functional approach* [24].
They are both based on using a more expressive lattice than for context-insensitive dataflow analyses,

$$Nodes \to Contexts \to States_{\perp}$$

where *Nodes* is the set of CFG nodes, *Contexts* is the set of call contexts. $States$ is the lattice representing program states, and $States_{\perp}$ is the augmented $States$ lattice, where $\perp$ signifies that the state is not reachable from the program entry.
As a reminder, the context-insensitive analysis would typically ignore contexts, getting the following lattice.

$$Nodes \to States$$

In cases where the $States$ lattice already models reachability, it is not necessary to lift the lattice with $\perp$ to signify unreachable locations [20].

Call Strings With Call Strings, we define $Contexts = Calls^{\leq k}$, where $Calls$ is the set of all call stacks, and $Calls^{\leq k}$ is the set of all tuples whose length is less than or equal to k. We distinguish between function calls by their call stack, whose maximum length is determined by k. This approach makes it possible to set an upper bound to the depth of an analysis, thus also constraining the runtime cost of analysing a program, and avoiding issues which may arise when analysing recursive function calls. However, it will analyse calls to the same function more than necessary, in cases where the call stack is different, even if the input to the functions is the same.

Functional Approach With the Functional Approach, we define $Contexts = States_{\perp}$, where we distinguish function calls by the abstract state of their input instead of the call stack. This approach can become quite costly, depending on the size of the possible combinations of the input state. However, it does not need to recompute function calls where the input state is the same, and it can continue to an arbitrary depth.

Chapter 4

Rust

The Rust programming language is an increasingly popular language, designed to be fast, efficient, and safe. Design on the language began in 2006, with the first stable version being released on 2015. The high level goals of the language are to provide performance comparable to systems programming languages like C and C++, but without sacrificing the safety of higher-level languages, like Java and Python.
This is a difficult problem to solve, because the systems programming languages need to be able to control how memory is managed explicitly in order to be performant, which is easy to do incorrectly. The higher-level languages sacrifice this control for convenience, using tools such as garbage collectors to automate memory management.
Consider, as an example, how heap memory is handled in C and Java. In C we can allocate memory on the heap by using the `malloc` or `calloc` functions. The data on the heap can then be reclaimed with `free`. These functions are easily misused, e.g. by calling `free` twice on the same pointer, which is undefined behaviour, and forgetting to free memory would be wasting that memory.
Java, and many other high level programming languages, use a different approach. Here the programmer does not have to manually allocate and deallocate heap memory, but it is reclaimed at runtime using a *garbage collector*. While this makes it easier to write programs that correctly handle memory, it also impacts program performance, due to the operational overhead of determining which memory is no longer in use.
In Rust, this problem is addressed by the ownership system, which requires that the code is written in such a way that the compiler can infer when memory should be allocated, and released to the allocator. If the code does not satisfy the requirements, it will not compile.

4.1 Tooling

Rust comes with official package managers, which we briefly describe in the following subsections.

4.1.1 Cargo

Cargo is the name of Rust's package manager. It can be used to create new packages, download dependencies and compile packages. More advanced features supported by Cargo are publishing packages and running tests. Packages created and used by cargo are known as *crates* [1].
Cargo is designed to be the central tool which developers use when working with crates. When programs have the `cargo` prefix, Cargo will forward function calls to them as if they were called without the prefix, providing information on the crate in the context that it was called. Tools like Clippy [4] work in this way.

4.1.2 Rustup

While Cargo is used to manage crates, Rustup is used to manage Rust tools, like Cargo and the Rust compiler, and any other components which have been added to the toolchain. Among other things, Rustup makes it possible to have fine-grained control over the version of Rust used to compile crates. To understand how this affects development, we explain the versioning used in the Rust ecosystem.

New stable and beta versions of Rust are released every six weeks, and a nightly version is released every night, based on the previous day's `master` branch in the central git repository. The beta versions are based on the latest nightly every six weeks, and the stable versions are based on the previous beta.
Features introduced in nightly versions of the compiler must be explicitly enabled in the source code, as shown in Listing 4.1.

Example 4.1 – Enabling Features

```
#![feature(box_patterns)]
```

Listing 4.1: Enabling the `box_patterns` feature.

In order to develop any sort of tooling for Rust, developers must use nightly versions, although they can pin the version to a certain date if they wish.
Rustup allows developers to set the version specifically for a given project, or as a global default.

4.2 Language Basics

The Rust language itself is unique in many ways, and we strive to summarize the key points in the following subsections.

4.2.1 Syntax

Rust syntax is mostly in line with the C family of languages. The ubiquitous Hello, World! program can be written as follows:

Example 4.2 – Hello World

```
fn main() {
    println!("Hello, world!");
}
```

While curly brackes indicate scope, and statements end with a semicolon, Rust differs from the C family of languages in that its function definitions do not start with the return type. Typically, in fact, the types are written after the identifiers, as demonstrated in Listings 4.2, 4.3, and 4.4.

Example 4.3 – Struct Definition

```
struct Point {
    x: f32,
    y: f32,
}
```

Listing 4.2: The type definition for a point in two-dimensional space.

Example 4.4 – Enum Definition

```
enum Either<L, R> {
    Left(L),
    Right(R)
}
```

Listing 4.3: A Generic Rust enum.

In Listing 4.3 we demonstrate Rust's `enum` type with generics. The generics are indicated by the angle brackets, and are two anonymous types.
Enums and Structs are the two central types available in Rust. Enums are accessed via pattern matching, whereas struct fields can be accessed directly.

Example 4.5 – Function Definition

```
fn middle(p1: Point, p2: Point) -> Point {
    Point {
        x: (p1.x + p2.x) / 2,
        y: (p1.y + p2.y) / 2,
    }
}
```

Listing 4.4: Computing the point between two points in two-dimensional space.

In Listing 4.4, we compute and return a new point, placed in between the two points provided. This may look strange to those used to seeing the `return` keyword used instead. While it is possible to return values in that way, it is considered idiomatic Rust to return by ending the function with an expression, which is implicitly interpreted by the compiler as a return statement.

4.2.2 Functions

Rust completely separates the definition of behavior from the definition of data. To implement behavior for an enum or a struct, we add an `impl` block, as shown in Listing 4.5. Functions which take `self` are methods, and have access to the fields of the struct on which they are called. Functions which otherwise are declared in `impl` blocks but without taking `self` are associated methods, and are available on the static type.

Example 4.6 – Methods and Associated Functions

```
impl Point {
    fn add(self, other: Point) -> Point {
        Self::add_inner(self, other)
    }

    fn add_inner(p1: Point, p2: Point) -> Point {
        Point {
            x: (p1.x, p2.x),
            y: (p1.y, p2.y),
        }
    }
}
```

Listing 4.5: Associating behavior with the point defined in Listing 4.2.

Otherwise, functions are declared independently of a type, as shown in Listing 4.4.

4.2.3 Traits

In Rust, behavior is shared using Traits, which are similar to Haskell's type classes, or C++ concepts. As an example, a trait could be defined for implementing a method that generates a string from an instance of a type.

Example 4.7 – Trait Definition

```
trait ToString {
  fn to_string(&self) -> String
}
```

The trait could then be implemented for the boolean type.

Example 4.8 – Trait Implementation

```
impl ToString for bool {
  fn to_string(&self) -> String {
    if *self {"True"} else {"False"}
  }
}

```

The trait could afterwards be implemented for other types, and used to generically create functions that are implemented for many types. For example we could create a function that prints the `to_string` of any type implementing the `ToString` trait.

Example 4.9 – Trait Consumption

```
fn print_to_string<T: ToString>(t: &T){
  println!("The to_string is: {}", t.to_string())
}

```

For more information on traits we refer to [10].

4.2.4 Generics

Rust supports generic functions, and allows for shared behavior via traits. There are two ways generics are handled in Rust. The first way uses monomorphization, which generates a concrete implementation of the function for each type it is used with. While this improves runtime performance, it does so at the cost of compilation time and executable size. Listing 4.6 demonstrates three ways to write a generic function of this kind.

Example 4.10 – Generic Function Definitions

```
/// Simple generic definition
fn gobble(f: impl Fn(&str)) {
    f("Hello, world!");
}

/// Alternative syntax.
fn gobble<F: Fn(&str)>(f: F) { ... }

/// Third variant. Useful when the definition grows large.
fn gobble<F>(f: F)
where
  F: Fn(&str) { ... }
```

Listing 4.6: Three variants of defining a generic function which is generic over functions that accept a string.

The second way uses trait objects to call the methods, which is less performant at runtime, but takes less time to compile and generally results in smaller executables. Since a concrete function is not generated for each of these at compile time, we must wrap the trait object with a pointer, to make sure that it has a known size at compile time. We use the `Box` type for this purpose in Listing 4.7.

Example 4.11 – Generic with Trait Objects

```
fn gobble(f: Box<dyn Fn(&str)>) {
    f("Hello, world!")
}
```

Listing 4.7: A function which will remain generic at runtime. Can be rewritten in the same style as the other two variants in Listing 4.6.

4.2.5 Ownership

As mentioned in the beginning of the chapter, Rust has an approach to memory management which is different from the classical approach of C or C++, where memory is managed manually through functions like `malloc` and `free`, or by letting the garbage collector manage the resources.
In Rust, values have owners, and once the owner goes out of scope, the value is deallocated.

Example 4.12 – Heap Data Owned by Binding

```
{
    let a = String::from("content"); // The String data on the heap is created
                                     // and set to be owned by binding a
} // The scope in which a is active is now over, and a is now no longer valid
  //  At the same time the string data on the heap is freed
```

In cases where the value needs to outlive a given scope, it can be *moved* to a new owner in a different scope. This happens, for example, when a value is passed as a parameter to a function, or on assignment, with the caveat that some values are so cheaply copied that a move is never necessary.

Example 4.13 – Transferring Ownership

```
{
    let a = String::from("content");
    let b = a; // The ownership of the String is transferred from a to b

    println!("The content is: {}", a); // Trying to use a after the transferring the
                                       // ownership will give a compile time error
}
```

This would quickly become burdensome in cases where a value needs to be passed to a function, and then used again later. For that case, we have references, which can be immutable, or read-only, or mutable, meaning that the owner of the reference has direct access to the underlying value as well.
When a reference is taken, it is known as borrowing in Rust, and is a key interaction in the ownership model. When a value is borrowed, the owner is unable to modify it. A value can only be borrowed mutably if no other borrow is valid for the duration of the borrow. A value can be borrowed immutably multiple times, as there is no possibility of a race condition when the underlying cannot be modified.
These rules make it possible to infer when resources can be deallocated, and guarantee memory safety, but also make it impossible to create structures that are not tree-formed.
For example, it would be impossible to create a linked list with these conditions. That is because the ownership system is an over-approximation of memory safety, and there are cases where completely valid and safe code

can be written, which does not satisfy its conditions. This is why much of the standard library is implemented using the `unsafe` keyword, which allows the programmer, among other things, to bypass these rules.
In practice, this means that memory bugs still happen in Rust, but that the ownership system prevents them from originating in safe Rust.

4.2.6 Lifetimes

The lifetime of a reference determines the duration it is valid for. Each reference in Rust has a lifetime, and the compiler infers constraints for these lifetimes based on how they are used. The compiler will emit an error if the lifetime constraints cannot be satisfied.
Sometimes the compiler requires hints for the relationship between the input and output lifetimes of a function. It will always require annotations on type definitions.
This is important for two reasons. First, it allows the compiler to ensure that a reference never outlives the underlying value, and second, it allows the compiler to ensure that there never is an overlap of mutable access to a value.

4.2.7 Dataflow Analysis

The ownership system is implemented using dataflow analysis [5]. There are several traits and types with generic methods which make it possible to define a custom dataflow analysis using the types available in the compiler. These traits and types are defined for MIR, which is one of the intermediate representations available during compilation, which we describe in the following chapter.

4.3 Compilation Pipeline

The Rust compiler goes through several phases, each of which requires the source code to be represented with a different IR. This is because the different kinds of analyses lend themselves to different levels of abstraction. On the one end we have the abstract syntax tree, which is created during parsing, and on the other end, we have the machine code, ready to be executed. One of the intermediate representations is MIR, which is a CFG representation of the code. It is generated per function, and therefore needs to be combined into an ICFG to be used in an interprocedural taint analysis.

Chapter 5

Mid-Level Intermediate Representation

MIR is a stage in the Rust compilation pipeline designed for dataflow analysis, and is structured as a CFG. The most basic unit in MIR is the function, which is composed of basic blocks. Each basic block contains a series of statements and ends with a single terminator. While statements and terminators both affect program state, terminators alone determine the next basic block to execute. To avoid the confusion between statements as described in the theory in Chapter 3, and statements as they are defined in MIR, we refer to the set of MIR statements and terminators as *instructions*. Whenever we refer to statements in this chapter, we refer to MIR statements.

5.1 MIR Syntax

In reality, MIR is a collection of data structures which are processed directly in-memory, and represented as text only for the convenience of the programmer. When printing the textual representation of MIR, the compiler issues a warning, stating that the format is unstable and subject to change. Therefore, any syntactical definition is, by nature, only valid for a given version of the programming language, with no guarantees for forward or backward compatibility. Despite this limitation, we define a limited subset of the MIR syntax, for the purpose of designing a taint analysis. The subset closely resembles the textual representation output by the compiler, with a few caveats. Notably, we exclude types, references, closures, generators, and any operators or operations which require those items. The definition is as follows, beginning with the most granular units:

$$\begin{aligned} z &\in \mathbb{Z} \\ i &\in \mathsf{Ident} \end{aligned}$$

z is a literal integer value. i is any valid function identifier in a MIR program, which are all strings satisfying the regular expression: $[\texttt{a-zA-Z}]^+$. While by our definition z, can only store integer values, any other type can be encoded into an integer representation.
Our MIR subset contains binary and unary operations.

$$\begin{aligned} bop &\in \mathsf{BinOps} = \{\mathsf{Add}, \mathsf{Sub}, \mathsf{Mul}, \mathsf{Div}, \mathsf{Rem}, \mathsf{BitXor}, \mathsf{BitAnd}, \mathsf{BitOr}, \mathsf{Shl}, \mathsf{Shr}, \mathsf{Eq}, \mathsf{Lt}, \mathsf{Le}, \mathsf{Ne}, \mathsf{Ge}, \mathsf{Gt}\} \\ uop &\in \mathsf{UnOps} = \{\mathsf{Not}, \mathsf{Neg}\} \end{aligned}$$

Additionally, some operations are defined on Rvalues, which is a term from C++, and represents values which cannot be accessed directly through a memory location.
In MIR, a Place is what in C++ would be referred to as an Lvalue. Places store locations, through which values can be read and modified.

$$
\begin{array}{rcl}
P \in \mathsf{Places} & ::= & \{\_n \mid n \in \mathbb{N}_0\} \\
\\
O \in \mathsf{Operands} & ::= & P \mid \texttt{const } z \\
\\
R \in \mathsf{Rvals} & ::= & O \\
& \mid & bop(O_1, O_2) \\
& \mid & uop(O) \\
\\
\beta \in \mathsf{Bid} & ::= & \{\texttt{bb}n \mid n \in \mathbb{N}_0\}
\end{array}
$$

We only support a single statement, namely assigning values to places. The terminators are more varied, as they impact the control flow of the program, which is more relevant for our analysis.

$$
\begin{array}{rcl}
S \in \mathsf{Stmt} & ::= & P \texttt{ = } R\texttt{;} \\
\\
X \in \mathsf{Targets} & ::= & z\texttt{: } \beta \\
\\
T \in \mathsf{Tmnt} & ::= & \texttt{goto -> } \beta\texttt{;} \\
& \mid & \texttt{switchInt(}O\texttt{) -> [}[X,]^*\texttt{ otherwise: } \beta\texttt{];} \\
& \mid & \texttt{assert(}O\texttt{, } z\texttt{) -> } \beta\texttt{;} \\
& \mid & P\texttt{ = } i\texttt{(}O_0, \ldots, O_n\texttt{) -> } \beta\texttt{;} \\
& \mid & i\texttt{(}O_0, \ldots, O_n\texttt{);} \\
& \mid & \texttt{return;}
\end{array}
$$

To summarize, a MIR program consists of one or more functions, which in turn contain basic blocks.

$$
\begin{array}{rcl}
\{S^*T\} \in \mathsf{Body} & ::= & \mathsf{Stmt}^* \times \mathsf{Tmnt} \\
\\
B \in \mathsf{Block} & ::= & \beta\texttt{: } \{S^*T\} \\
\\
F \in \mathsf{Fn} & ::= & \texttt{fn } i\texttt{(}O^*\texttt{) \{}B^+\texttt{\}} \\
\\
M \in \mathsf{Mir} & = & F^+
\end{array}
$$

In real MIR, basic blocks also contain a number of place declarations where it is declared if a place is mutable or immutable. For simplicity we ignore this and just assume that all places used in a program have been declared in advance.
These definitions provide the formal foundation on which we build our taint analysis. We now move on to formally describing the semantics of MIR.

5.2 MIR Semantics

In this section we formally introduce the operational semantics of the subset of MIR presented in Section 5.1. When executing a MIR program, the special `main` function is the entry point of the program. This means that the first instruction to be executed is the first instruction of the main function.
To define the formal semantics of MIR, we establish a way to evaluate a place to its value. Without functions this would be straightforward, as we could simply map each place to a value. However, with functions we can have more than one place with the same name which complicates matters a little. To get around that, we first introduce an infinite set of locations. Locations can be thought of as places that are unique across functions.

$$\xi \in \mathsf{Locations}$$

We then map places to locations in an environment, e.

$$e \in \mathsf{Environment} = \mathsf{Places} \to \mathsf{Locations}$$

And then map locations to $\mathbb{Z}_\perp$ in a Store, ζ. $\perp$ represents that the location has not been initialised.

$$\zeta \in \mathsf{Store} = \mathsf{Locations} \to \mathbb{Z}_\perp$$

This approach will allow us to map Places to different locations based on which functions they are used in. For this to work we require that $\zeta \circ e : \mathsf{Places} \to \mathbb{Z}_\perp$ is a total function.
Locations in MIR can be either mutable or immutable. We previously assume that all places have been declared in advance of a MIR program. At its declaration, the place, which corresponds to a unique location, has been declared mutable or immutable. Therefore the mutability declaration is connected to a location. In the case that a location is immutable, it can only be assigned a new value if it is $\perp$. We introduce a function called *LocationType* to tell us if a location is mutable or immutable.

$$\mathit{LocationType} = \mathsf{Locations} \to \{\mathit{mut}, \mathit{imut}\}$$

Throughout this report, the semantic rules use the mut property to determine if we are allowed to assign to a Location. The imut property is not often shown in the semantic rules, as trying to assign to an immutable location is not allowed in MIR.
We furthermore introduce a prg function to get a $fun \in \mathsf{Function}$ for an i.

$$prg \in \mathsf{Program} = \mathsf{Ident} \to \mathsf{Function}$$

And a fun function to get the instructions related to a β.

$$fun \in \mathsf{Function} = \mathsf{Bid} \to \mathsf{Body}$$

Furthermore we introduce a call stack to keep track of the stack of function calls in a program. Bid is the basic block that is to be executed following the next return instruction. Places is the place that the returned value must be written to. Environment is the environment of the current function. Function is the function to get the bodies of blocks inside the current function.

$$cs \in \mathsf{CallStack} = (\mathsf{Bid} \times \mathsf{Places} \times \mathsf{Environment} \times \mathsf{Function})^*$$

We have two types of configurations in our semantics. One configuration for a statement:

$$\mathsf{Conf}_S = \mathsf{Stmt} \times \mathsf{Store}$$

And another configuration for a basic block:

$$\mathsf{Conf}_B = \mathsf{Body} \times \mathsf{CallStack} \times \mathsf{Store}$$

We describe one step of executing a statement as the partial order:

$$\to_S \subseteq \mathsf{Conf}_S \times \mathsf{Store}$$

And one step of executing a basic block is:

$$\to_B \subseteq \mathsf{Conf}_B \times \mathsf{Conf}_B$$

Sections 5.2.1 and 5.2.2 describe how to evaluate Operands O and Rvalues R, which appear in many instructions. Afterwards, we describe the semantic rules for all MIR instructions in Section 5.2.3.

5.2.1 Operands

An operand can either be a place P or a constant `const` z. We introduce the Oeval function to evaluate O:

$$\mathit{Oeval}(O, \zeta, e) = \begin{cases} z, & \text{if } O = \mathit{const}\ z \\ \zeta \circ e(P), & \text{if } O = P \end{cases}$$

Note that in our subset of MIR, we do not consider the `move` syntax, which in the full MIR is another option for O.

5.2.2 Rvalues

The R right hand side can take on three different forms. It can be a *use*, a *binary operator* or a *unary operator*. We introduce the $Reval$ function to evaluate R:

$$Reval(R, \zeta, e) = \begin{cases} Oeval(O, \zeta, e) & \textit{if } R = O \\ \mathbf{bop}(O_1, O_2) & \textit{if } R = bop(O_1, O_2) \\ \mathbf{uop}(O) & \textit{if } R = uop(O) \end{cases}$$

Both the binary operators and the unary operators are fairly standard operators, so we do not elaborate further on them. If any input to a binary or unary operator is $\perp$, i.e., uninitialized, then the operation is undefined.

5.2.3 Instructions

This section contains the semantic inference rules of all instructions in our MIR subset. In the panicking call rule, some elements in the call stack are not available, as the rule does not have a Place to store the returned value or a Basic Block to transfer control to after executing the panicking function. In these cases we use the "_" symbol to denote a missing value.

$$[Ass]\ \frac{\langle \texttt{P = R}, \zeta\rangle \rightarrow_S \zeta[e(P) \mapsto Reval(R)] \qquad (LocationType(e(P)) = mut \vee \zeta \circ e(P) = \perp)}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \rightarrow_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R)]\rangle}$$

$$[Goto]\ \frac{}{prg \vdash \langle \texttt{\{goto -> } \beta\texttt{;\}}, (\beta', P', e', fun') :: cs, \zeta\rangle \rightarrow_B \langle fun'(\beta), (\beta', P', e', fun') :: cs, \zeta\rangle}$$

$$[SwitchInt_1]\ \frac{z_i = Oeval(O, \zeta, e'') \qquad i \in \{1, \ldots, n\}}{\begin{array}{l} prg \vdash \langle \texttt{\{SwitchInt(}O\texttt{) -> [}z_1 : \beta_1, \ldots, z_n : \beta_n, \texttt{ otherwise: } \beta'\texttt{];\}}, \\ \quad (\texttt{bb}n'', P'', e'', fun'') :: cs, \zeta\rangle \rightarrow_B \langle fun''(\beta_i), (\beta'', P'', e'', fun'') :: cs, \zeta\rangle \end{array}}$$

$$[SwitchInt_2]\ \frac{\forall z | z \in \{z_1, \ldots, z_n\} : z \neq Oeval(O, \zeta, e'')}{\begin{array}{l} prg \vdash \langle \texttt{\{SwitchInt(}O\texttt{) -> [}z_1 : \beta_1, \ldots, z_n : \beta_n, \texttt{ otherwise: } \beta'\texttt{];\}}, \\ \quad (\texttt{bb}n'', P'', e'', fun'') :: cs, \zeta\rangle \rightarrow_B \langle fun''(\beta'), (\beta'', P'', e'', fun'') :: cs, \zeta\rangle \end{array}}$$

$$[Assert]\ \frac{z = Oeval(O, \zeta, e')}{\begin{array}{l} prg \vdash \langle \texttt{\{Assert(}O\texttt{,}z\texttt{) -> } \beta\texttt{;\}}, \\ \quad (\beta', P', e', fun') :: cs, \zeta\rangle \rightarrow_B \langle fun'(\beta), (\beta', P', e', fun') :: cs, \zeta\rangle \end{array}}$$

$$[Call_{Normal}]\ \frac{\begin{array}{c} \xi_j \text{ fresh} \qquad e' = [\_j \mapsto \xi_j, \ldots] \qquad \zeta' = \zeta[\xi_j \mapsto Oeval(O_j, \zeta, e), \ldots] \\ j \in \{0, \ldots, n\} \qquad (LocationType(e(P)) = mut \vee \zeta \circ e(P) = \perp) \end{array}}{\begin{array}{l} prg \vdash \langle \texttt{\{}P\texttt{ = }i\texttt{(}O_0, \ldots, O_n\texttt{) -> } \beta\texttt{;\}}, (\beta', P', e', fun') :: cs, \zeta\rangle \rightarrow_B \\ \quad \langle prg(i)(\texttt{bb0}), (\beta, P, e, prg(i)) :: (\beta', P', e', fun') :: cs, \zeta'\rangle \end{array}}$$

$$[Call_{Panic}]\ \frac{\xi_j \text{ fresh} \qquad e' = [\_j \mapsto \xi_j, \ldots] \qquad \zeta' = \zeta[\xi_j \mapsto Oeval(O_j, \zeta, e), \ldots] \qquad j \in \{0, \ldots, n\}}{prg \vdash \langle \texttt{\{}i\texttt{(}O_0, \ldots, O_n\texttt{);\}}, (\beta', P', e', fun') :: cs, \zeta\rangle \rightarrow_B \langle prg(i)(\texttt{bb0}), (\_, \_, e, prg(i)) :: (\beta', P', e', fun') :: cs, \zeta'\rangle}$$

$$[Return]\ \frac{\zeta' = \zeta[e'(P) \mapsto e(\_0)]}{prg \vdash \langle \texttt{\{Return;\}}, (\beta, P, e, fun) :: (\beta', P', e', fun') :: cs, \zeta \rangle \rightarrow_B \langle fun'(\beta), (\beta', P', e', fun') :: cs, \zeta' \rangle}$$

Chapter 6

Static Taint Analysis

With Chapter 3 introducing the theory required for a static dataflow analysis, Chapter 4 introducing the Rust programming language, and Chapter 5 defining the syntax and semantics of MIR, we are equipped with the tools needed to design a static taint analysis in Rust.
We present a static taint analysis, which for any given function marked as a sink, ensures that none of its possible inputs have been tainted. It is a forward analysis, since tainting is a property of the past, and it is an over-approximation, because it returns the set of places which may be tainted. As described in Chapter 3, we need to define a complete lattice of finite height, and a set of transfer functions to set up a monotone framework. This monotone framework can then be instantiated with a CFG and an initial value to create an analysis. We use the functional approach described in Section 3.2.1 to create a context-sensitive interprocedural analysis. We start by defining the lattice which we use.

6.1 Lattice

We begin with a lattice containing two elements, t, short for "tainted" and u, short for "untainted". We furthermore define that $u \sqsubseteq t$, representing the fact that we have the most precise information as the bottom element, and the less precise information as the top element. The bottom element means that the place is definitely untainted, and the top element means that the place may be tainted. This is visualised in Figure 6.1. We call this lattice $Taint$. We use this lattice to represent the abstract state of a MIR place at a certain node

Figure 6.1: The lattice used for each place for each CFG node in a program.

in a CFG. To do this we create the map lattice from all places P in the program to the $Taint$ lattice. However, in MIR any number of functions can have places with the same name. This is especially noteworthy in the example of `_0` which is the place that holds the return value in every function. Because of this we need some way to differentiate between places in different functions. The solution is to prepend the function in which a place is located to the name of the place. We separate the function name and the place name by ':'. We denote the set of these places with prepended function calls as $Places_{Analysis}$.

> ***Example 6.1 – Places with Prepended Function Calls***
> The return place `_0` of a function *foo* will be called `foo:_0`.

We create the $State$ lattice:

$$State = Places_{Analysis} \rightarrow Taint$$

This lattice represents the abstract state of all places at a CFG node. To represent the state of all places at all CFG nodes we create a map lattice from the CFG node labels to the $State$ lattice:

$$States = Labels \to State$$

Finally, we expand the lattice to be able to use the functional approach to context-sensitivity.

$$States = Labels \to State_{\bot} \to State_{\bot}$$

As this is a complete lattice with a finite height, it satisfies the requirement for a monotone framework. With the lattice defined, we move on to describing the transfer functions for the monotone framework.

6.2 Transfer Functions

We define a transfer function for each instruction in our subset of MIR. As with the semantics, many of the transfer functions depend on the Operands O and the Rvalues R, so we start our discussion with those.

Operands

Operands can be either a place P or a constant z. We define the function $AOeval$ to evaluate an O as follows:

$$AOeval(O, State) = \begin{cases} u & If\ O = const\ z \\ State(P) & If\ O = P \end{cases}$$

If the operand is a constant, we can be sure that it is unaffected by any input to the program. Therefore it is safe to assume that a constant evaluates to u (untainted). On the other hand, if the operand is another place, it can have been affected by input to the program in an earlier state, and we must check the $State$ lattice for the current abstract state of the place.

Rvalues

Rvalues can be a use, a binary operator or a unary operator. Similarly to operands, we define a function $AReval$ to evaluate R:

$$AReval(R, State) = \begin{cases} AOeval(O_1, State) \sqcup AOeval(O_2, State) & If\ R = bop(O_1, O_2) \\ AOeval(O, State) & Otherwise \end{cases}$$

If the Rvalue is a binary operator, and one of the operators can be controlled by input to the program, then the result of the operation can also be controlled by input to the program. Therefore we evaluate both operands in the current state and return their least upper bound. If the Rvalue is a unary operator or simply a use, then it suffices to just return the abstract state of the operator. Note that the only two unary operators we have are used to negate and inverse operands, and thus do not remove a taint from an operand.

Assignment

The transfer function for assignment P `=` R is the following:

$$f_{t:assignment}(State) = State[P \mapsto Reval(R, State)]$$

The assignment transfer function either outputs the abstract state of an operand or outputs the join of the abstract state of two operands. From this it should be clear that the transfer function is monotone.

Call

The transfer function for the function call P `=` i`(`O^*`) -> bb`n`;` instruction requires a bit more work. First we introduce three special functions. One is the `input` function denoted f_{input}. We use this function to introduce taint in a program, and as such, it uses the following transfer function:

$$f_{t:call:input}(State) = State[P \mapsto t]$$

As this transfer function simply maps a place to t (tainted), it is also monotone. The second special function is the $f_{sanitise}$ function. This function is used to simulate a function that takes untrusted input, and processes it in such a way that it can be considered trusted input. The transfer function for the sanitise function is:

$$f_{t:call:sanitise}(State) = State[P \mapsto u]$$

As such, the input isn't transformed, but a sanitised version of the input is returned from the function. Again this transfer function is clearly monotone following the arguments presented earlier.
The last special function is the `output` function. This function has the identity transfer function:

$$f_{t:call:output}(State) = State$$

The identity transfer function is trivially monotone. If any operands O used in parameters to the output function evaluates to t, then our analysis should report that there is a potential security risk in the program. As such it is a *taint sink* as described in Chapter 1.
We also introduce the special `main` function as the starting point of an analysis. The output of the initial node in the `main` function is thus the initial value set by the analysis.
For all other function we use the functional approach described in Section 3.2.1. During this approach, each non-special function call is split into two CFG nodes, a *call* and *return* node. The call node will have the identity transfer function:

$$f_t(State) = State$$

Note that in MIR the return value of the function is always stored in the place `_0` of a function. Therefore the transfer function for the return node becomes:

$$f_{t:call:return}(State) = State[P \mapsto FunctionName\texttt{:\_0}]$$

Where $FunctionName$ is the name of the returning function. This transfer function is also monotone following the arguments of the assignment transfer function.

SwitchInt

For the SwitchInt terminator, `switchInt(`O`) -> [`$[X,]^*$ `otherwise:` β`];`, there is a non-trivial decision to make about our taint analysis. While it is obvious that we do not want tainted information to flow into output functions, tainted data can also be used to decide control flow in control flow instructions. An argument against having the conditions in control flow instructions as sinks is that it is perfectly normal for user input to decide control flow of a program. However, it can also be argued that such input could be sanitised before being used to decide control flow. We do not use conditions in control flow instructions as sinks, and consider it to be out of scope.

Terminators

Common for all other terminators that we include in our subset of MIR is that they only affect the control flow of the program, and not its places. Therefore they all have the simple transfer function:

$$f_{t:terminators}(State) = State$$

With transfer functions for all our instructions defined and their monotonicity argued for, this concludes our monotone framework. We give an example of a program, and instantiate an analysis based on that program and our monotone framework.

6.3 Instantiated Analysis

To set up an analysis we first require a program which we want to analyse. We use the MIR program in Listing 1. For simplicity this program does not include non-special function calls, for an example of an ICFG see Example 3.7. This program takes user input in line 3 and depending on the input, either binds _4 to the

```
1  fn main() {
2      bb1{
3          _1 = input() -> bb2;
4      }
5      bb2{
6          _2 = Sub(_1, const 3);
7          _3 = Gt(const 0, _2);
8          switchInt(_3) -> [0: bb3, otherwise: bb4];
9      }
10     bb3{
11         _4 = const 2;
12         goto -> bb5;
13     }
14     bb4{
15         _4 = input() -> bb5;
16     }
17     bb5{
18         _5 = output(_4) -> bb6;
19     }
20     bb6{
21         return;
22     }
23 }
24
```

Listing 1: The MIR program to be analysed

constant value 2, or to the result of new user input. In line 18, _4 is used as output, a taint sink. If the program at runtime follows the bb3 branch, then this is not a problem. However, if the program follows the bb4 branch, then we have taint flowing into a taint sink.

To analyse this program using our monotone framework, we first turn the program into a CFG. This CFG is visualised in Figure 6.2.

We use our monotone framework to analyse the CFG of the program. We start the analysis at the initial node. By definition, the input to the initial node of the main function is an element in our lattice. We assume that no values are tainted at the beginning of the program, since taints are introduced through function calls which expose us to external input. Therefore the output of the initial node will be that each place maps to u. We shown the steps that the analysis goes through to reach its result in Table 6.1.

In l_8 the instruction is the special `output` function. The program passes parameter $\_4$ to the function. By the definition of the `output` function, if we pass it a t operand, there is a potential security risk in the program. As $\_4$ is t, we do exactly that, and our analysis reports that there is an issue in the program.

This matches our intuition from the informal program description. Naturally, for larger programs, especially programs with loops, it becomes very hard to do such an analysis by hand, but the same procedure followed here will give correct results.

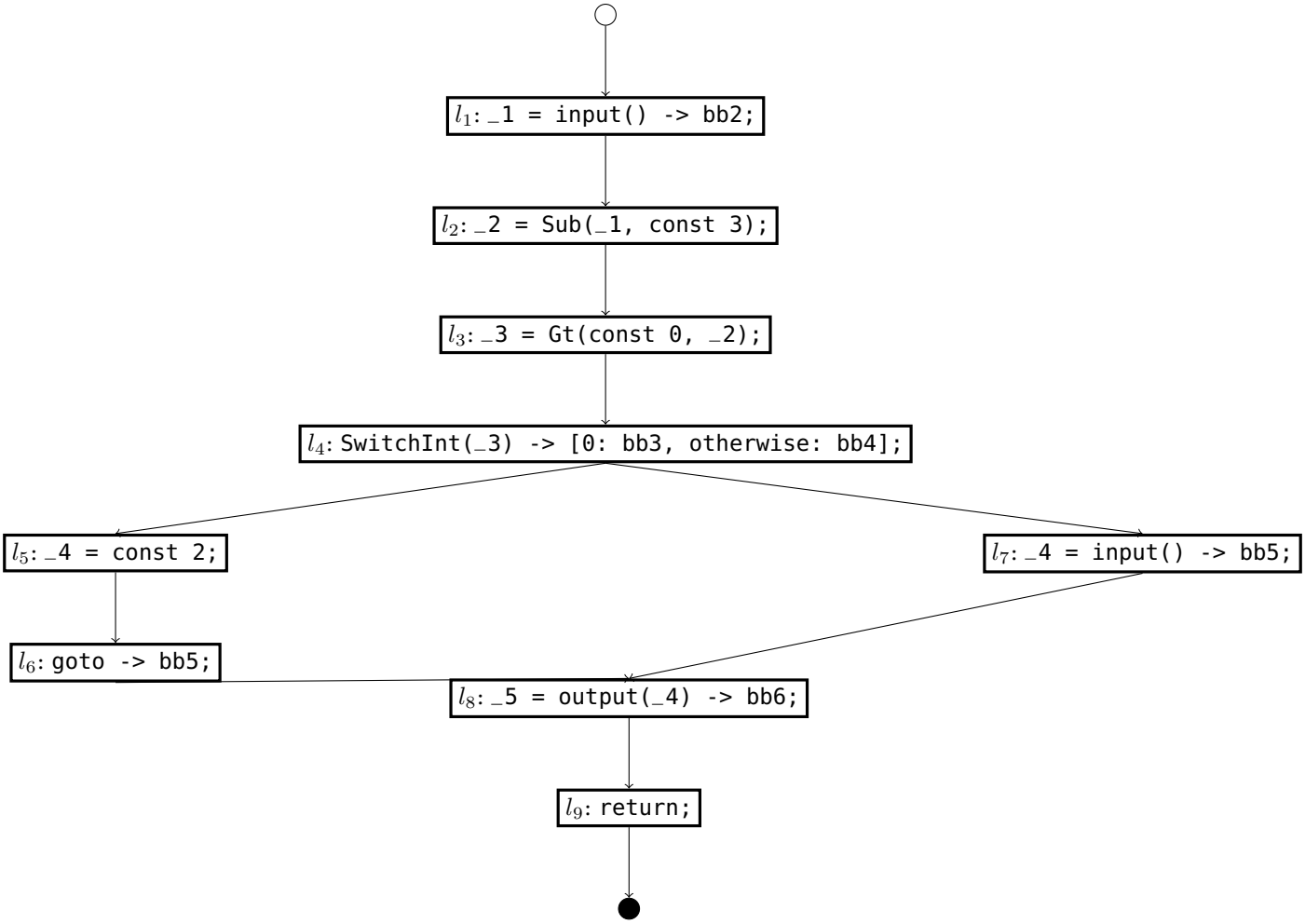


Figure 6.2: The CFG of the program in Figure 1

Table 6.1: Steps of the analysis created from the monotone framework and the program in Figure 1

Label	Instruction	Transfer Function	Input	Output
l_1	`_1 = input() -> bb2;`	$f_{t:call:input}(State) = State[P \mapsto t]$	$\forall P \in$ Places : $P \mapsto u$	$State(l_1)[\_1 \mapsto t]$
l_2	`_2 = Sub(_1, const 3);`	$f_{t:assignment}(State) = State[P \mapsto eval(R, l_i)]$	$output(l_1)$	$State(l_2)[\_2 \mapsto t]$
l_3	`_3 = Gt(const 0, _2);`	$f_{t:assignment}(State) = State[P \mapsto eval(R, l_i)]$	$output(l_2)$	$State(l_3)[\_3 \mapsto t]$
l_4	`SwitchInt(_3) -> [0: bb3, otherwise: bb4];`	$f_{t:terminators}(State) = State$	$output(l_3)$	$State(l_4)$
l_5	`_4 = const 2;`	$f_{t:assignment}(State) = State[P \mapsto eval(R, l_i)]$	$output(l_4)$	$State(l_5)$
l_6	`goto -> bb5;`	$f_{t:terminators}(State) = State$	$output(l_5)$	$State(l_6)$
l_7	`_4 = input() -> bb5;`	$f_{t:call:input}(State) = State[P \mapsto t]$	$output(l_4)$	$State(l_7)[\_4 \mapsto t]$
l_8	`_5 = output(_4) -> bb6;`	$f_{t:call:output}(State) = State$	$output(l_6) \sqcup output(l_7)$	$State(l_8)$
l_9	`return;`	$f_{t:terminators}(State) = State$	$output(l_8)$	$State(l_9)$

Chapter 7

Ownership

One of the key features of Rust is the ownership model, which we describe in Chapter 4. The ownership model is useful, because it establishes rules which make certain kinds of analysis straightforward. The idea of ownership dates back as far as 1998 [14], where it was shown to improve the precision of static analysis on programs which manipulate the heap. Its main purpose is to restrict aliasing, to make it easier to reason about how data is manipulated. It strictly limits the use of references, by differentiating between mutable and immutable references. Any number of immutable references can be used simultaneously, but if a mutable reference is held, it must be the only reference to the underlying resource. This allows us to make assumptions about the impact of immutable and mutable references in our taint analysis. For instance, since an immutable reference cannot mutate the underlying value, we can assume that taint cannot be introduced through immutable references. Conversely, having a mutable reference means that there are no other references pointing to the underlying resource, which allows us to directly taint the underlying resource.
We also differentiate between values which can be copied trivially, and are marked with the `Copy` trait, and those which are not. In general, if a value cannot be copied trivially, the compiler moves it instead, and invalidates the previous binding.
It is important to note that Rust allows for code that disables the ownership system through the use of the *unsafe* keyword. Many standard library features, such as the vector struct, are defined using the unsafe keyword. There is ongoing work to verify that the unsafe parts of the standard library uphold the properties otherwise guaranteed by the Rust compiler [17]. While do not formally prove that our analysis is sound when obeying the ownership system, we can definitely say that the analysis is not sound when using the unsafe keyword to disable the ownership system.
We now extend the subset of MIR to support the ownership syntax and semantics.

7.1 Syntax

When an operand refers to a place, it does so as either a move or a copy operation.

$$O \in \mathit{Operands} \quad ::= \quad \texttt{move}\ P \mid \texttt{copy}\ P \mid \texttt{const}\ z$$

Rvalues support mutable and immutable references:

$$\begin{array}{rcl} R \in \mathit{Rvals} & ::= & O \\ & \mid & \mathit{bop}(O_1, O_2) \\ & \mid & \mathit{uop}(O) \\ & \mid & \&[\texttt{mut}]\ P \end{array}$$

We must now consider the semantic impact of the move operation and the inclusion of references.

7.2 Semantics

The syntactical changes have a cascading effect throughout our definition of MIR. All definitions which use Rvalues and operands now must account for references, and whether a value is `const`, `copy`, or `move`. We

describe the particulars of these changes for the statements and terminators they affect.

7.2.1 Store Changes

With the new changes, we store references to Locations in Locations, therefore we need to update the $Store$ function to be able to map to other Locations. We also add a new $\top$ value to represent invalidated Locations.

$$\zeta \in Store = \mathsf{Locations} \to (\mathbb{Z}^{\top}_{\bot} + \mathsf{Locations})$$

7.2.2 Operands

Since both `move` P and `copy` P consist of a Place P, they are evaluated according to the $Oeval\ P$ rule. Thus we redefine the $Oeval$ function as follows:

$$Oeval(O, \zeta, e) = \begin{cases} z, \ if\ O = \texttt{const}\ z \\ \zeta \circ e(P), \ if\ O = \texttt{move}\ P \\ \zeta \circ e(P), \ if\ O = \texttt{copy}\ P \end{cases}$$

7.2.3 Rvalues

To accommodate that Rvalues can now contain references we need to expand the $Reval$ function to evaluate references:

$$Reval(R, \zeta, e) = \begin{cases} Oeval(O, \zeta, e) & \text{if } R = O \\ \mathbf{bop}(O_1, O_2) & \text{if } R = bop(O_1, O_2) \\ \mathbf{uop}(O) & \text{if } R = uop(O) \\ \zeta \circ e(P) & \text{if } R = \&[mut]P \end{cases}$$

7.2.4 Assignment

We need a new semantic rule for assignment when using the move operand and for the new invalidated location. The new assignment semantics are described below.

$$[Ass]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad (R = (O \vee uop(O)) \wedge O \neq \texttt{move}\ P) \vee (R = bop(O_1, O_2) \wedge O_1 \neq \texttt{move}\ P' \wedge O_2 \neq \texttt{move}\ P'')}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \to_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R)]\rangle}$$

$$[Ass_{Move1}]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad O = \texttt{move}\ P' \quad R = (O \vee uop(O))}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \to_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P') \mapsto \top]\rangle}$$

$$[Ass_{Move2}]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad O_1 = \texttt{move}\ P' \quad O_2 \neq \texttt{move}\ P'' \quad R = (bop(O_1, O_2))}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \to_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P') \mapsto \top]\rangle}$$

$$[Ass_{Move3}]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \to_S \zeta[e(P) \mapsto Reval(R)] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad O_1 \neq \texttt{move}\ P' \quad O_2 = \texttt{move}\ P'' \quad R = (bop(O_1, O_2))}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \to_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P'') \mapsto \top]\rangle}$$

$$[Ass_{Move4}]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \rightarrow_S \zeta[e(P) \mapsto Reval(R)] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad O_1 = \texttt{move } P' \quad O_2 = \texttt{move } P'' \quad R = (bop(O_1, O_2))}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \rightarrow_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto Reval(R), e(P') \mapsto \top, e(P'') \mapsto \top]\rangle}$$

When the Rvalue on the right hand side of an assignment is a reference $R = \&[mut]P$, the Rvalue is not evaluated using $Reval$. We first introduce a function called $BorrowType$ which takes a place P and returns whether P is not borrowed, borrowed immutably or borrowed mutably.

$$BorrowType = P \rightarrow \{borrowed_{not}, borrowed_{imut}, borrowed_{mut}\}$$

Defining the semantics of the ownership system is out of scope of this project, as it requires an entire analysis on its own. The resulting semantic rule is then:

$$[Ass_{Ref:mut}]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \rightarrow_S \zeta[e(P) \mapsto e(P')] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad R = \&mut\ P' \quad BorrowType(P') = borrowed_{not}}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \rightarrow_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto e(P')]\rangle}$$

$$[Ass_{Ref:imut}]$$

$$\frac{\langle \texttt{P = R}, \zeta\rangle \rightarrow_S \zeta[e(P) \mapsto e(P')] \quad (BindingType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top \quad R = \&P' \quad BorrowType(P') \neq (borrowed_{mut})}{prg \vdash \langle \texttt{\{P = R; S*T\}}, (\beta, P, e, fun) :: cs, \zeta\rangle \rightarrow_B \langle \texttt{\{S*T\}}, (\beta, P, e, fun) :: cs, \zeta[e(P) \mapsto e(P')]\rangle}$$

7.2.5 Normal Call

For the normal function call, we also need to introduce new semantics for using the move operand. The additions in this rule is that the left hand side Place can not be $\top$, and all operands given to the function using the move semantics are $\top$ after the function call.
The new call semantic rule can be found below.

$$[Call_{Normal}] \frac{\xi_j \text{ fresh} \quad e' = [\_j \mapsto \xi_j, \dots] \quad \zeta' = \zeta[\xi_j \mapsto Oeval(O_j, \zeta, e'), e'(P_j) \mapsto \begin{cases} \top & If O_j = \texttt{move } P_j \\ \zeta \circ e'(P_j) & otherwise \end{cases}, \dots] \quad j \in \{0, \dots, n\} \quad (LocationType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top}{\begin{array}{r} prg \vdash \langle \texttt{\{P = } i(O_0, \dots, O_n) \texttt{ -> } \beta\texttt{;\}}, (\beta', P', e', fun') :: cs, \zeta\rangle \rightarrow_B \\ \langle prg(i)(\texttt{bb0}), (\beta, P, e, prg(i)) :: (\beta', P', e', fun') :: cs, \zeta'\rangle \end{array}}$$

7.2.6 Panicking Call

The semantic rule for the panicking function call is changed in a similar way as the semantic rule for the normal call.

$$[Call_{Panic}] \frac{\xi_j \text{ fresh} \quad e' = [\_j \mapsto \xi_j, \dots] \quad \zeta' = \zeta[\xi_j \mapsto Oeval(O_j, \zeta, e'), e'(P_j) \mapsto \begin{cases} \top & If O_j = \texttt{move } P_j \\ \zeta \circ e'(P_j) & otherwise \end{cases}, \dots] \quad j \in \{0, \dots, n\} \quad (LocationType(e(P)) = mut \vee \zeta \circ e(P) = \bot) \quad \zeta \circ e(P) \neq \top}{\begin{array}{r} prg \vdash \langle \texttt{\{}i(O_0, \dots, O_n)\texttt{;\}}, (\beta', P', e', fun') :: cs, \zeta\rangle \rightarrow_B \\ \langle prg(i)(\texttt{bb0}), (\_, \_, e, prg(i)) :: (\beta', P', e', fun') :: cs, \zeta'\rangle \end{array}}$$

7.2.7 Remaining instructions

The semantics of all remaining instructions are unaffected by the introduction of ownership.

7.3 Extending Taint Analysis

We can split the implications that introducing ownership to MIR has into two parts: The implications of the move semantics in assignments and function calls, and the implications of now being able to create references.

7.3.1 Move Implications

The move semantics of MIR invalidates the right hand side Places of an assignment, and the actual parameters of a function call. As an invalidated Place will no longer map to any value, we can say that it will be untainted u. However, using an invalidated Place in a MIR program will result in a compile time error. Therefore any program that would try to use an invalidated Place is not a valid MIR program, and we do not change the transfer functions due to the move semantics.

7.3.2 Reference Implications

Statically analysing languages that allow for references or pointers, has typically resulted in less precise analyses than analyses for similar languages without those constructs. This is due to the aliasing problem [20], which we illustrate with the following program:

> ***Example 7.1 – Aliasing Problem***
> In this program the value of x depends on whether or not y points to the same location, in other words, whether or not they are aliases.
>
> ```
> 1  x = 3
> 2  y = 5
> ```

Typically, the way to deal with this would be to create an allocation site abstraction, and use an algorithm to figure out what locations could possibly point to which allocation site. If two or more locations could point to the same allocation site, then the analysis would have to assume that the locations could be aliases. This is also known as a points-to analysis [20]. The fact that the ownership model in Rust does not allow for more than one mutable reference to the same Place, prevents the situation where two mutable references point to the same Place, and where modification of one of them may change the state of the other. Note that it is still possible for several references to point to the same Place, but all references would be immutable.
Therefore the ownership concept simplifies the aliasing problem, and allows for increased precision in static analyses with references.
In the case that a Location L' that stores another Location L'' becomes tainted, taint is propagated from L' to L''. If L'' is also a Location storing another Location, the taint is again propagated and so on. This continues until the Location storing a $\mathbb{Z}_{\perp}^{\top}$ is tainted.

Chapter 8

Implementation

In addition to the theoretical analysis in the previous chapter, we present an implementation of the analysis. The Rust compiler provides a convenient interface for controlling the compilation process. The analysis implementation uses this interface to run the compilation past the analysis phase, at which point we run the taint analysis. This allows us to assume that our analysis runs on valid Rust code, as the compiler, barring bugs, will already have performed the necessary validation. Once the taint analysis finishes, we end the compilation.

8.1 Hooking into the Compiler

The Rust compiler design makes it possible to use it both as a binary and as a collection of libraries. Rust compiler crates can be recognized by their common prefix, `rustc`, such as `rustc_driver`, which provides the interface for running the compiler.
To have access to the compiler internals, several things are required. Firstly, there is no stability guarantee for the compiler internals, and the programs using compiler internals must do so using *nightly* versions of Rust. Secondly, the crates must have the `rustc_private` feature, which allows crates to access individual compiler crates. Thirdly, these crates must be explicitly specified in the crate root, using the `extern crate` keywords. For example, if we wanted to include the internal crate `rustc_driver`, we would write in the crate root:
`extern crate rustc_driver;`
With compiler internals available, we implement the callback trait and execute the compilation process as follows:

```
1  struct TaintCompilerCallbacks;
2
3  impl rustc_driver::Callbacks for TaintCompilerCallbacks {
4      /// All the work we do happens after analysis, so that we can make assumptions about ↩
       ↪ the validity of the MIR.
5      fn after_analysis<'tcx>(
6          &mut self,
7          compiler: &rustc_interface::interface::Compiler,
8          queries: &'tcx rustc_interface::Queries<'tcx>,
9      ) -> Compilation {
10         compiler.session().abort_if_errors();
11         enter_with_fn(queries, mir_analysis);
12         compiler.session().abort_if_errors();
13         Compilation::Stop
14     }
15 }
```

Listing 8.1: The Rust compiler interface accepts an object which implements its callback trait, allowing us to execute code after each compilation phase.

The trait has four available functions which can be implemented, but we are only interested in the one which is called after analysis. If we had the need, we could also call the ones that happens after parsing and expansion, as well as configuration.
Running the compiler is done as follows:

```
rustc_driver::RunCompiler::new(&args, callbacks).run()
```

Listing 8.2: Using the Rust compiler like a library.

Where the arguments are a vector of strings, as they would be submitted when calling the binary itself. This allows us to implement a wrapper for the compiler which can accept custom commands for our taint analysis, submitting some of those to our callbacks and others to the compiler itself.
The callbacks are those structs we implement ourselves, in this case the `TaintCompilerCallbacks` defined above.

8.2 Taint Analysis

The only callback we implement is the `after_analysis` function, where we run the taint analysis. We only support programs which have a single `main` function, as this simplifies the analysis, such that we only have to analyse the path available from the entry point.
To perform the analysis, we need access to the type context `TyCtxt`, which allows us to make queries to get the High-level Intermediate Representation (HIR) and MIR instances for the programs we analyse.
We abstract the complexity of accessing the type context with the `enter_with_fn`, written as follows:

```
/// Call a function which takes the ‘TyCtxt‘.
fn enter_with_fn<’tcx, TyCtxtFn>(queries: &’tcx rustc_interface::Queries<’tcx>, enter_fn: ↩
    ↪ TyCtxtFn)
where
    TyCtxtFn: Fn(TyCtxt),
{
    queries.global_ctxt().unwrap().peek_mut().enter(enter_fn);
}
```

Listing 8.3: Providing the `TyCtxt` to the function passed as a parameter.

`enter_with_fn` takes a function which accepts as its first and only parameter a type context, simplifying the call as demonstrated in Listing 8.1, where the function is called.
We begin the taint analysis by submitting `mir_analysis` as the argument to `enter_with_fn`, the implementation shown in Listing 8.4.

```
/// Perform the taint analysis.
fn mir_analysis(tcx: TyCtxt) {
    let (entry_def_id, _) = if let Some((entry_def, x)) = tcx.entry_fn(LOCAL_CRATE) {
        (entry_def, x)
    } else {
        let msg =
            "this tool currently only supports taint analysis on programs with a main ↩
    ↪ function";
        rustc_session::early_error(ErrorOutputType::default(), msg);
    };

    let main_id = entry_def_id.to_def_id();
    main::eval_main(tcx, main_id);
}
```

Listing 8.4: We begin the taint analysis by finding the entry function’s identifier, and passing that into our main eval function, along with the type context.

8.2.1 Tagged Functions

To make the analysis practically applicable, we add support for function annotations. Functions can be marked as sources, sinks or sanitizers, such as in the test program defined in Listing 8.5. The Rust programming language allows for such notations for cases where external tools may need extra information. While it only recognizes two tools as of writing this report, extra tools can be registered using the `register_tool` feature.

```
#![feature(register_tool)]
#![register_tool(taint)]

fn main() {
    let val = source();
    sink(val); //~ ERROR function ‘sink‘ received tainted input [T0001]
}

#[taint::source]
fn source() -> i32 {
    15
}

#[taint::sink]
fn sink(_: i32) {
    ()
}
```

Listing 8.5: One of the example programs in our test suite. This program tests that the annotations work, and that they lead to a simple propagation from a source to a sink.

Before the taint analysis proper begins, we parse the MIR, looking for such annotations. The Rust compiler provides a trait and methods for finding annotations quickly using the visitor pattern, and our implementation of those can be seen in Listing 8.6 and Listing 8.7.

```
impl<’v> ItemLikeVisitor<’v> for TaintAttributeFinder<’_> {
    fn visit_item(&mut self, item: &’v rustc_hir::Item<’_>) {
        self.visit_hir_id(item.hir_id());
    }

    fn visit_trait_item(&mut self, trait_item: &rustc_hir::TraitItem<’_>) {
        self.visit_hir_id(trait_item.hir_id());
    }

    fn visit_impl_item(&mut self, impl_item: &rustc_hir::ImplItem<’_>) {
        self.visit_hir_id(impl_item.hir_id());
    }

    fn visit_foreign_item(&mut self, foreign_item: &rustc_hir::ForeignItem<’_>) {
        self.visit_hir_id(foreign_item.hir_id());
    }
}
```

Listing 8.6: Implementing the `ItemLikeVisitor` trait for our type, which accepts all function declarations that have been annotated. The trait defines functions for four kinds of items, which are essentially Rust constructs in different environments. We accept annotations on all kinds of functions, and thus implement this behavior for all item types. This uses the `visit_hir_id` method, since the behavior is shared between item types.

This implementation uses an internal method which we share, since the behaviour is the same for all item types. Essentially, we want to see if the item is a function, and whether it has been annotated with one of the three traits we accept. Annotations are read as Symbols, many of which are declared internally in the compiler. Keywords and reserved words are defined as symbols, for example. To read the custom symbols we implement,

we must define our own symbols. This can be done using the `Symbol::intern` function. Knowing this, we implement the necessary behavior to recognize sources, sinks and sanitizers in the HIR of the programs we analyse, as shown in Listing 8.7.

```
impl TaintAttributeFinder<'_> {
    fn visit_hir_id(&mut self, item_id: hir::HirId) {
        let def_id = self.tcx.hir().local_def_id(item_id).to_def_id();
        let sym_source = Symbol::intern("source");
        let sym_sink = Symbol::intern("sink");
        let sym_sanitizer = Symbol::intern("sanitizer");
        let attrs = self.tcx.hir().attrs(item_id);
        for attr in attrs {
            if let AttrKind::Normal(ref item, _) = attr.kind {
                if let Some(symbol) = get_taint_attr(item) {
                    if symbol == &sym_source {
                        self.info.sources.push(def_id)
                    } else if symbol == &sym_sink {
                        self.info.sinks.push(def_id)
                    } else if symbol == &sym_sanitizer {
                        self.info.sanitizers.push(def_id)
                    } else {
                        self.tcx.sess.emit_err(InvalidVariant {
                            attr_name: symbol.to_ident_string(),
                            span: item.span(),
                        })
                    }
                    break;
                }
            }
        }
    }
}
```

Listing 8.7: We define the `visit_hir_id` method, which stores the identifiers of functions annotated as sources, sinks or sanitizers.

We only search for these annotations in the crate being analysed, however, the interfaces available through the compiler also make it possible to do so for each dependency. The purpose of the annotation is to explicitly mark these functions in context of a taint analysis, so that we can introduce taints when the sources are called, and emit warnings when a path for tainted values may exist between a source and a sink.
It is also worth noting that we do not support any other granularity than functions, which means that we cannot tag specific arguments a function may accept as sinks, instead of the entire function. It is possible to extend our program to do this however, by accepting more types of items in the finder, and implementing the propagation in the `TransferFunction` methods.

8.2.2 Accessing the MIR

The `TyCtxt` can be queried using function identifiers to access their MIR bodies. The function is called `optimized_mir` and returns a Body which can be submitted to the internal analysis methods available in the compiler. We perform this once for the main function of the program, as demonstrated in Listing 8.8.

```
pub fn eval_main(tcx: TyCtxt<'_>, main_id: DefId) {
    // Find all functions in the current crate that have been tagged
    let mut finder = TaintAttributeFinder::new(tcx);
    tcx.hir().krate().visit_all_item_likes(&mut finder);

    let entry = tcx.optimized_mir(main_id);
```

```
7
8       let _ = TaintAnalysis::new(tcx, &finder.info)
9           .into_engine(tcx, entry)
10          .pass_name("taint_analysis")
11          .iterate_to_fixpoint();
12  }
```

Listing 8.8: The highest-level function dedicated to the taint analysis, `eval_main` finds the functions marked as sources, sinks, or sanitizers, and begins the taint analysis at the `main` function.

The compiler internals support dataflow analysis, and provide traits for defining more. There is also a generic implementation of the dataflow analysis engine which accepts any types that implements these traits. These traits are `AnalysisDomain` and `Analysis`.
Both traits may be implemented for the same datatype, which can then be submitted to the engine. We name our type `TaintAnalysis`, and its definition is shown in Listing 8.9.

```
1   pub struct TaintAnalysis<'tcx, 'inter> {
2       /// We use the type context to emit errors and get the MIR for other functions.
3       tcx: TyCtxt<'tcx>,
4       /// All the functions that have been marked
5       info: &'inter AttrInfo,
6       contexts: Rc<RefCell<Contexts>>,
7       init: InitSet,
8       points: RefCell<PointsMap>,
9   }
```

Listing 8.9: The central type on which the taint analysis is implemented. This type contains all the necessary information for analyzing a single function, and will spawn more analyses of the same type for inner function calls. Some information, like the fields `tcx`, `info`, and `contexts` are shared among all instances of the type, `init` and `points` are relevant to the relation between function calls.

8.2.3 The Domain

We define the domain of the analysis in Rust by implementing the `AnalysisDomain` trait, as shown in Listing 8.10. This trait requires that the implementation define the type of `Domain`, which we define as `BitSet<Local>`. The type `Local` is an index type, corresponding to what we in our formal definition of MIR call a `Place`. It is essentially a series of bits, where `0` corresponds to u, untainted, and `1` corresponds to t, tainted.

```
1   impl<'inter> AnalysisDomain<'inter> for TaintAnalysis<'_, '_> {
2       type Domain = BitSet<Local>;
3       const NAME: &'static str = "TaintAnalysis";
4
5       type Direction = Forward;
6
7       fn bottom_value(&self, body: &Body<'inter>) -> Self::Domain {
8           // bottom = definitely untainted
9           BitSet::new_empty(body.local_decls().len())
10      }
11
12      fn initialize_start_block(&self, body: &Body<'inter>, state: &mut Self::Domain) {
13          // For the main function, locals all start out untainted.
14          // For other functions, however, we must check if they receive tainted parameters.
15          if !self.init.is_empty() {
16              for (_, arg) in self
17                  .init
18                  .iter()
19                  .zip(body.args_iter())
20                  .filter(|(&t, _)| t.unwrap_or(false))
```

```
            {
                state.set_taint(arg, true);
            }
        }
    }
}
```

Listing 8.10: The `AnalysisDomain` trait defines the type of the abstract domain used during the analysis, as well as initial values in `bottom_value` and how these values are handled once the engine begins in `initialize_start_block`.

For the block initialization, we specify that if the `self.init` field is not empty, that we propagate the taint from the caller to the current function. This is the case, for example, when we initially construct the analysis using the entry point. Propagating the taint like this is necessary to understand how different input states may have different results for calling the same function multiple times, and is a part of how we implement the functional approach, which we describe on a theoretical level in Section 3.2.1.

8.2.4 Transfer Functions

`TaintAnalysis` is the central type in the analysis, but for convenience we also define a separate type where we implement transfer functions, which we define in Listing 8.11.

```
struct TransferFunction<'tcx, 'inter, 'intra> {
    tcx: TyCtxt<'tcx>,
    info: &'inter AttrInfo,
    contexts: Rc<RefCell<Contexts>>,
    state: &'intra mut PointsAwareTaintDomain<'intra, Local>,
}
```

Listing 8.11: In order to avoid cluttering `TaintAnalysis` with too much behavior, we separate the methods related to transfer functions into `TransferFunction`. It implements the MIR `Visitor` trait, which lets us focus only on the parts of the MIR that are necessary to analyse during a taint analysis.

Rust compiler internals provides a `Visitor` trait, which can be used to traverse the MIR for a single function. Using those, we can implement the `Analysis` trait for `TaintAnalysis` as shown in Listing 8.12. Note that the `Analysis` trait will not compile unless the `AnalysisDomain` trait has also been implemented, since it uses the `&mut Self::Domain` type to define the state that it accepts as a parameter in its functions.

```
impl<'tcx, 'inter, 'intra> Analysis<'intra> for TaintAnalysis<'tcx, 'inter> {
    fn apply_statement_effect(
        &self,
        state: &mut Self::Domain,
        statement: &Statement<'intra>,
        location: Location,
    ) {
        TransferFunction {
            tcx: self.tcx,
            info: self.info,
            contexts: self.contexts.clone(),
            state: &mut PointsAwareTaintDomain {
                state,
                map: &mut self.points.borrow_mut(),
            },
        }
        .visit_statement(statement, location);
    }

    fn apply_terminator_effect(
        &self,
```

```
        state: &mut Self::Domain,
        terminator: &Terminator<'intra>,
        location: Location,
    ) {
        TransferFunction {
            tcx: self.tcx,
            info: self.info,
            contexts: self.contexts.clone(),
            state: &mut PointsAwareTaintDomain {
                state,
                map: &mut self.points.borrow_mut(),
            },
        }
        .visit_terminator(terminator, location);
    }

    fn apply_call_return_effect(
        &self,
        _state: &mut Self::Domain,
        _block: BasicBlock,
        _func: &Operand<'intra>,
        _args: &[Operand<'intra>],
        _return_place: Place<'intra>,
    ) {
        // do nothing
    }
}
```

Listing 8.12: We implement the `AnalysisDomain` trait for statements and terminators by constructing a `TransferFunction`, which we use to visit the relevant parts of the MIR.

8.2.5 Function Summaries

We use function summaries to make the analysis context-sensitive. We describe function summaries as *the functional approach* on in Section 3.2.1.
In the case where we visit function calls, we first check if the function has been tagged, and if it hasn't, we check the function summaries.

```
fn t_visit_call(
    &mut self,
    func: &Constant,
    args: &[Operand],
    destination: &Option<(Place, BasicBlock)>,
    span: &Span,
) {
    let name = func.to_string();
    let id = match func.literal.ty().kind() {
        TyKind::FnDef(id, _args) => Some(id),
        _ => None,
    }
    .unwrap();

    match self.info.get_kind(id) {
        Some(AttrInfoKind::Source) => self.t_visit_source_destination(destination),
        Some(AttrInfoKind::Sanitizer) => self.t_visit_sanitizer_destination(destination),
```

```
        Some(AttrInfoKind::Sink) => self.t_visit_sink(name, args, span),
        None => self.t_fn_call_analysis(args, id, destination),
    }
}
```

Listing 8.13: For every function call in the source program, we compute the definition identifier, to see if it is one of the functions that has been marked as a source, sink, or sanitizer. If it has not, we move on to compute the function summary.

The function summaries are stored on the `TaintAnalysis` field `contexts`. We implement it as a hash map, where we map a given function identifier and the input state to the output state, allowing us to distinguish between function calls by the abstract state.

```
fn t_fn_call_analysis(
    &mut self,
    args: &[Operand],
    id: &rustc_hir::def_id::DefId,
    destination: &Option<(Place, BasicBlock)>,
) {
    let init = args
        .iter()
        .map(|arg| match arg {
            Operand::Copy(p) | Operand::Move(p) => Some(self.state.get_taint(p.local)),
            Operand::Constant(_) => None,
        })
        .collect::<Vec<_>>();

    let end_state = self.t_function_summary(id, init);

    if let Some(end_state) = end_state {
        let return_place = Local::from_usize(0);

        if end_state.get_taint(return_place) {
            self.t_visit_source_destination(destination);
        }

        let target_body = self.tcx.optimized_mir(*id);
        let arg_map = args
            .iter()
            .map(|arg| arg.place().or(None))
            .zip(target_body.args_iter())
            .collect::<Vec<_>>();

        // Check if any variables which were passed in are tainted at this point.
        for (caller_arg, callee_arg) in arg_map {
            if let Some(place) = caller_arg {
                self.state
                    .set_taint(place.local, end_state.get_taint(callee_arg));
            }
        }
    }
}
```

Listing 8.14: We begin by computing the input state necessary to see if we already have the function summary stored in our mapping. If `end_state` is `None`, we are in a recursive call, and ignore this branch entirely, simply continuing the analysis of the function's other blocks. If it is `Some`, we use the value to compute how the taint is propagated from the callee to the caller. First, we check the return place, which is always `_0`. Then, we check that any of the parameters we passed into the function have been tainted.

If there is no entry for the current function call, we insert an intermediate value that indicates the analysis for the entry is in progress, and begin the analysis. The intermediate entry allows us to avoid problems with recursion, as calling the nested function with the same input state twice will begin the analysis, but ignore the second function call, instead continuing the analysis of the nested function.

```
fn t_function_summary(&mut self, id: &DefId, init: Vec<Option<bool>>) -> Option<BitSet<Local>> {
    let key = (*id, init.clone());

    if let Some(summary) = self.t_get_cached_summary(&key) {
        summary
    } else {
        // In the case that we have recursive or mutually recursive function calls,
        // we make sure that we only compute a summary once per key by inserting None while we compute it.
        // For subsequent calls, calling ‘t_function_summary‘ will simply return None and the visitor will analyze other branches.
        self.t_insert_summary(&key, None);

        let target_body = self.tcx.optimized_mir(*id);
        let mut results =
            TaintAnalysis::new_with_init(self.tcx, self.info, self.contexts.clone(), init)
                .into_engine(self.tcx, target_body)
                .pass_name("taint_analysis")
                .iterate_to_fixpoint()
                .into_results_cursor(target_body);

        let state = if let Some(last) = target_body.basic_blocks().last() {
            results.seek_to_block_end(last);
            Some(results.get().clone())
        } else {
            None
        };

        // Once the function summary has been computed, we insert it into the cache.
        self.t_insert_summary(&key, state.clone());

        state
    }
}
```

Listing 8.15: This function handles three cases. The first case is that a function summary already exists for the given identifier and state. In that case, we return the pre-existing summary. The second case is that the entry is `None`, which means that we have already begun computing the result, but have not finished. This only happens when the same function is called again within its own scope, i.e., a recursive call. In this case, we ignore subsequent recursive calls with the same input state, instead checking if any other paths in the function may result in taints. The third case is the standard case, where we compute the summary, insert it into the cache and return it.

```
fn t_insert_summary(
    &mut self,
    key: &(DefId, Vec<Option<bool>>),
    val: Option<BitSet<Local>>
) {
    self.contexts.borrow_mut().insert(key.clone(), val);
}
```

Listing 8.16: This function simply inserts information into the function summaries. We keep it in its own scope to ensure that the mutable reference is dropped as soon as we are done inserting.

```
fn t_get_cached_summary(
    &mut self,
    key: &(DefId, Vec<Option<bool>>),
) -> Option<Option<BitSet<Local>>> {
    let contexts = self.contexts.borrow();
    contexts.get(key).map(|res| res.clone())
}
```

Listing 8.17: Returns a clone of the summary if it exists, otherwise `None`. We keep it in its own scope to ensure that the reference is dropped as soon as we have the inner value.

Once we have the final state of the function we call, we check if any of the references that we passed into the function have been tainted, and taint them in the calling function as well. Here the ownership system provides an advantage, as we have precise information about which parameters are `move`, `copy` or constants, and can track their owners with precision. We compute a mapping between variables within a context, which is added to whenever a reference to a variable is assigned, and follow this mapping when we track the taint in propagation. This way, we ensure that we over-approximate the taint with regard to aliasing, although we could be even more precise, and taint only the value that is referred to, and not the reference itself, only propagating the taint when the reference is dereferenced. We did not implement the analysis with this amount of precision, however.

Chapter 9

Discussion

When designing a static analysis, one has to consider the trade-off between preserving soundness and creating a precise analysis. As mentioned in Chapter 3, many real world static analyses aren't sound. Instead they choose to prefer precision above soundness in order to not produce too many false positives, which may annoy programmers [13, 23]. While we have not formally proven that the static analysis presented in this report is sound, we took care to not take any decision that would lead to an unsound analysis. We made this decision to investigate how precise we could make a sound analysis using the features of Rust.

At the time of writing, MIR is only available as an internal data structure during the Rust compilation process. This works for a tool integrated into the compiler such as the one proposed in this report. However, for an external tool, a stable representation of MIR is highly preferable. A discussion about introducing such a thing has already taken place on the internal chat platform for Rust [11]. By giving a stable representation of MIR, it also becomes easier to formally define the language. This would hopefully lead to an official semantics for MIR, which could increase the trust in tools working on MIR. As such we believe, that for a language which prides itself on safety, that work should be put into standardising and formalizing MIR.

We spent much time perusing the rustc guide to compiler development [5], as well as asking questions on Zulip to understand how the compiler internals work. While the answers provided on Zulip were excellent, we did not always get answers, and the rustc guide is incomplete, with some sections being empty, marked as a work in progress. When implementing analysers and verifiers, it would be beneficial to have a more thorough documentation of the compiler internals, and examples of how they can be used to build simple tools for Rust. What we could not glean from the rustc guide or Zulip, we learned from studying the source code of other tools mentioned Chapter 2 on related work.

One point that we emphasise in Chapter 2 is the need for making analysis tools easily accessible for programmers. The static analysis proposed in this report is made by having the Rust compiler call our analysis during the compilation of a program. Therefore the analysis can with relative ease be included in the Rust compiler for easy access for programmers.

For future work, we believe that work is needed in two main areas, improving the analysis presented in the report, and working on Rust to better support static analysis.

The analysis could be improved by integrating the tool fully into the Rust compiler. It could also be worth doing a more comprehensive study of how precise the analysis is over a large set of programs. To make the analysis even easier to use, pre-written models with known sources and sinks from common libraries could be added. Lastly, support could be added for having several taint types and user defined taint types.

In the context of working on Rust, we have already discussed how a standardised version of MIR could help external tools and research, and how this could lead to formally defining MIR by giving a formal syntax and semantics for the language. Another area where work could be done, would be in expanding the generic dataflow analysis implementation in Rust to easily allow for interprocedural analysis.

Chapter 10

Conclusion

The problem statement is as follows: *How does the precision of static taint analysis benefit from the programming model in the Rust Programming Language?*
This is the main question which we strive to answer. In general, we describe the implications of the ownership system with regard to the aliasing problem in Section 7.3.2. We discuss the practical precision that can be achieved in Section 8.2.5.
In order to answer this question, we make three contributions:

1. We formalize a subset of MIR, which as far as we are aware has not been done before, and we hope that this can lay the foundation for future work on MIR.
2. We define a formal taint analysis, where we investigate how the precision can be increased.
3. We implement the analysis which we define as a Rust tool.

Formalizing MIR While projects exists which strive to create a model similar to MIR, we formalize MIR itself using operational semantics in Chapter 5. In the beginning of the chapter, we explain that we are only formalizing a subset of MIR, but the subset that we formalize is enough to perform a taint analysis. By formally defining MIR, we set up the necessary environment to define exactly where the precision of a taint analysis can be increased, by strictly defining what is available at any given point during the execution of a MIR program.

Defining a Taint Analysis We demonstrate what precision we can achieve with the extra information provided by the ownership system formally, by defining the taint analysis based on the MIR semantics in Chapter 6. We show that the ownership system simplifies reasoning about aliasing, and makes it less difficult to reason about which places a reference could be pointing to.

Rust Tool We create a proof of concept which demonstrates how the analysis will capture tainted input to a sink, and properly handle sanitized functions, described in Chapter 8. This tool is built using compiler internals, and hooks into the compiler to perform its analysis. The analysis can be tightly integrated into the Rust compiler, providing easy access to the analysis for programmers to use it in their workflow.

Bibliography

[1] Cargo introduction. https://doc.rust-lang.org/cargo/.

[2] Github - facebookexperimental/mirai: Rust mid-level ir abstract interpreter. https://github.com/facebookexperimental/MIRAI.

[3] Github - rust-lang/miri: An interpreter for rust's mid-level intermediate representation. https://github.com/rust-lang/miri.

[4] Github - rust-lang/rust-clippy: A bunch of lints to catch common mistakes and improve your rust code. https://github.com/rust-lang/rust-clippy.

[5] Guide to rustc development. https://rustc-dev-guide.rust-lang.org.

[6] Memorable quotes from edsger dijkstra (1930-2002). https://www.cs.scranton.edu/~mccloske/dijkstra_quotes.html.

[7] Owasp top ten. https://owasp.org/www-project-top-ten/.

[8] Prusti programming methodology group | eth zurich. https://www.pm.inf.ethz.ch/research/prusti.html.

[9] Pyre. https://pyre-check.org/.

[10] Rust traits introduction. https://blog.rust-lang.org/2015/05/11/traits.html.

[11] Zulip mir discussion. https://rust-lang.zulipchat.com/#narrow/stream/183875-wg-formal-methods/topic/Access.20to.20MIR.20from.20third-party.20tools.

[12] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

[13] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):6675, February 2010.

[14] David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, 1998.

[15] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[16] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[18] Stephen Cole Kleene, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*, volume 483. van Nostrand New York, 1952.

[19] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 18, USA, 2005. USENIX Association.

[20] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[21] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. 01 1999.

[22] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[23] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):5866, March 2018.

[24] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , 1978.

[25] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 44–51, 2018.

[26] Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of rust, 2020.