UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Correctness Criteria for Function-Based Reclassifiers: A Language Based Approach

—

Steinar Brenna Hansen

INF-3981 Master's thesis in Computer Science - June 2022

“I think it is inevitable that people program poorly. Training will not substantially help matters. We have to learn to live with it.”
–Alan Perlis

“Computer viruses are an urban legend.”
–Peter Norton, 1988

Abstract

An emerging problem in systems security is controlling how a program uses the data it has access to. Information Flow Control (IFC) propagates restrictions on data by following the flow of information, for example if a secret value flows to a public value, that value should be considered secret as well. A common problem in IFC is reclassification of data, for instance to explicitly make data less restricted. An IFC mechanism often has strict flow rules in its normal operation, but reclassification by definition need to bypass these restrictions.

This thesis proposes correctness criteria that aim to provide stronger semantic guarantees for the behavior of reclassification functions. We first conduct a survey on prior work in IFC, which concludes that little emphasis has been put on crystallizing such criteria. We then define a set of criteria for reclassification and implement a parser to enforce these criteria. If a piece of code is successfully analyzed by the parser, then that code can be safely used to reclassify data. Rust is emerging as one of the more prominent languages for systems programming due to its memory safety, and we conjecture this can be analogously continued to target IFC as well.

Acknowledgements

First and foremost, I want to thank my supervisor Elisavet Kozyri and co-supervisor Håvard Johansen for their guidance and support throughout this process. Special thanks to Eliza for the thorough followup and for sharing your extensive knowledge of IFC with me.

I want to thank my classmates for their friendship and company throughout these years. We did it together, and I will treasure the bonds we have formed forever. Special thanks to my partner in crime Mariel for collaborating with me whenever possible. Thanks to my friends from outside the university for welcome distractions over the years.

Thanks to my family for your love and support over the years, and specially my uncle and aunt for feeding me whenever I visit. Thanks to my wonderful girlfriend Astrid for your love and support, you made the most challenging times in writing this thesis much more pleasant.

While the last 5 years have been stressful, they have also been the best years of my life. If I could go back in time, I would do it all over again.

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

List of Abbreviations

AST	Abstract Syntax Tree
CXPT	Context-Points
DLM	Decentralized Label Model
IDE	Integrated Development Environment
IFC	Information Flow Control
JIF	Java Information-Flow
RIF	Reactive Information Flow

1 Introduction

While *access control* manages security at the end points of a system, and *encryption* handles security outside a system, *Information Flow Control (IFC)* deals with security inside a system [1]. When access control grants a program permission to execute, it has little control over what that program does. IFC deals with security *during* execution, and aims to ensure that data in the system is handled correctly according to the security policy of the system. This is often done by labeling data with security labels that mark their sensitivity, and then follow and update labels according to how data influences other data throughout the system. For example, a company might want to label some of their documents as private and some others as public. Depending on the policies of the company, they might want public documents to never contain information from private documents, meaning any public document that is written private data to should be labeled as private. Enforcement of IFC is often done at the programming language level.

A common operation in IFC systems is manually altering the restrictions of a value [2]. This is referred to as *reclassification*, and can be used to describe changes to both integrity or confidentiality in data labels. Reclassification is for many IFC systems the only way to override labels in the system, and it is necessary to implement many practical security policies. For example, if a company is ordered by a judge to publicize some private data, the company could *declassify* the data to be public. Reclassification is specified into four cases: *declassification*, *endorsement*, *erasure*, and *deprecation* [3]. Declassification refers to a decrease in confidentiality, and inversely erasure refers to an increase in confidentiality. Endorsement refers to an increase in integrity, and deprecation refers to a decrease in integrity.

Generally, models that implement reclassification can be placed in three main categories: *state-based*, *authority-based*, and *function-based* reclassification. For simplicity, we refer to models by their reclassification scheme throughout the thesis, as this is the most relevant factor for our comparisons. *State-based reclassification* uses state predicates to determine when data should be reclassified. For example, the bids in an auction could be private until after the auction closes. *Authority-based reclassification* ensures the principals in the system have sufficient authority over data to reclassify it. An example of this could be a judge deciding that some sensitive case information is safe to make public, as they have high authority over this data. *Function-based reclassification* uses functions to modify the data to restrict more or less information, and these functions are used to describe the security policy of the system. An example of this is how the average of some sequences of numbers will reveal very little information about the original sequence. Generally, there are many approaches to reclassification by the different models. We explore this more in the survey in chapter 2.

Rust is an increasingly more prominent programming language for systems due to its focus on correctness, memory safety [4] and performance [5]. According to both Google [6] and Microsoft [7], a significant number of security bugs are related to memory safety. Rusts memory model can almost entirely eliminate memory related bugs in many systems. However, as far as we are aware, there has not been significant work done on creating standalone IFC mechanisms for Rust.

This makes Rust an interesting programming language to explore regarding IFC, and we conjecture that extending the safety guarantees provided by Rust with those of IFC models can enable building robust security systems in the future.

1.1 Project Statement

This project investigates reclassification in a comparative survey. One of the main conclusions of this survey is that the function-based models do not impose strict guarantees for how reclassification functions may manipulate data.

> *This thesis sets out to define criteria that can be used to provide guarantees for how reclassifier functions transform data.*

A standalone enforcement mechanism for these criteria is implemented using Rust and procedural attribute macros.

1.2 Context

This project was completed in the context of the Cyber Security Group (CSG) at UiT. The CSG group investigates fundamental systems problems rooted in practical application domains [8]. The group investigates interdisciplinary problems related to computer science, law, medicine, business and more, by bringing together persons of different expertise to accomplish these problems. The methodology chosen by the CSG group is primarily an experimental systems approach, where we construct prototype systems as part of solving a particular research problem. However, a more theoretical approach might be used if the problem lends itself naturally to this.

A recent publication from the CSG group related to IFC is "Expressing Information Flow Properties" from Kozyri, Chong, and Myers [3]. This monograph provides a recent summary of the field of IFC, based on how it has evolved over the last 40 years, and poses some challenges for the future.

1.3 Methodology

Comer et al. [9] describes an intellectual framework for the discipline of computer science. This was the final report of the Task Force on Computer Science, and was endorsed and approved by the ACM Education Board. The report presented a way to divide computer science into three distinct paradigms.

> *"Computer science and engineering is the systematic study of algorithmic processes (their theory, analysis, design, efficiency, implementation, and application) that describe and transform information.".*

They define the following paradigms as part of their model:

- **Theory** is based on the mathematical roots of computer science, and consists of four steps followed in the development of coherent, valid theory:

    1. Characterize objects of study by *definitions*.

    2. Hypothesize possible relations between objects with *theorems*.

   3. Determine whether such relationships are true via *proofs*.
   4. Interpret results.

- **Abstraction** is based on the notion that scientific progress is based on forming hypotheses and verifying and validating them. Abstraction is concerned with the ability to use relationships among objects to make predictions that can be compared with the world. Abstraction consists of four stages that are followed in the investigation of a phenomenon:

   1. Form a *hypothesis*.
   2. Construct a *model* and make a *prediction*.
   3. Design an *experiment* and collect *data*.
   4. Analyze results.

- **Design** is based on the notion that progress is achieved by posing problems and following the design process to construct systems that solve them. Design is concerned with the ability to implement specific instances of object relationships and use them to perform useful actions. The four steps followed in the construction of a system to solve a given problem is defined as follows:

   1. State *requirements*.
   2. State *specifications*.
   3. *Design* and *Implement* the system.
   4. *Test* the system.

The overlap of these paradigms is emphasized, and this overlap is also apparent in how the directions describe object relations.

In this thesis, we work mostly within the design paradigm. We state the requirements and specifications of the system based on the findings from our survey. We then present a design or a system, and implement a prototype of this design. This prototype is then evaluated by a set of conjectured tests, to show how the implementation matches with the specification.

1.4 Thesis outline

The remainder of the thesis is structured as follows:

Chapter 2: Reclassification Survey is a survey on how different models implement reclassification, and how this appears in the literature. The chapter is divided into function-, authority-, and state-based models, and presents a simple taxonomy of each category.

Chapter 3: Correctness Criteria and Parsing outlines the design of correctness criteria for reclassifiers, and a parser to detect and verify them.

Chapter 4: Implementation describes the implementation of the parser and the choices made to fulfill the specifications.

Chapter 5: Evaluation describes some examples used to evaluate the implementation, and showcases some points of discussion.

Chapter 6: Discussion and Future Work discusses the results from the previous chapters, including discussing the results from evaluation.

Chapter 7: Related work and Conclusion compares our results to some related works, and summarizes the thesis.

2 Reclassification Survey

2.1 Comparative Analysis

We investigate approaches that use state-based reclassification, authority-based reclassification and function-based reclassification. To provide a structured comparison of the different models, we constructed a set of common properties we find IFC systems often inhabit, and attempt to use these properties as points of comparison between the established models. These properties can be found in Figure 2.1. We chose papers that we feel have contributed to the field in different ways and are relevant to reclassification.

We structure this survey into three sections representing the different approaches to reclassification. Each section contains a summary of some prominent models, and a taxonomy of how these relate to the common properties in Figure 2.1. We conclude the chapter with a summary of the survey, and some remarks on where we found some of the models lacking.

1. How the authors express policies in their framework, in regard to syntax and security conditions.
2. How is a reclassifier formulated (syntax of reclassifiers).
3. What type of enforcement is used.
4. What are the criteria for performing a reclassification within the enforcement mechanism.

Figure 2.1: List of properties used for comparative analysis in survey.

2.2 State-based reclassification

State-based reclassification denotes models where the reclassification of a given value depends on some state in the system. This state can be any value in the system, from constantly incrementing values such as the current time, to rarely updated access policies. The reclassifier mechanism would need to track the system state predicate, and compare to its reclassify condition to fulfill the desired policy. An example policy could be that the value v is sensitive until 4 hours afters its initial creation, after which it is public. An inverse policy of this is also possible, a policy might say the value v is public until 2 weeks after its release.

This is a flexible approach to reclassification in that it poses little restrictions on what can and cannot be used as a basis of reclassification. This means that it can be leveraged in many ways to model security policies, and has produced a breadth of different approaches that fall within this category.

2.2.1 Erasure

Erasure [10] is the term for when a reclassifier makes a value more sensitive, in other words the operation of raising the confidentiality of a value. This is implemented as part of an enforcement system, when a state in the system fulfills some condition, some other data can be marked as unreadable. Chong and Myers [10] implement erasure within the DLM system, enabling policies to be defined to delete some value based on a state predicate.

Policies are expressed with simple labels, denoted by a down arrow for declassification and an up arrow for erasure, as such $H \searrow^{pred} L \nearrow^{pred} H$. Reclassification is done by defining state predicates on these arrows. This is enforced by a type system, and the main criteria for reclassification is the state predicates.

2.2.2 Flow Locks & Paralocks

Flow locks [11] is a language for building expressive and statically verifiable IFC policies. Paralocks [1] extends Flow Locks to include principals, roles, and relations. A flow lock specifies a state predicate for when an actor may gain information about some protected data. A way to model labels using this is by a set of locks for each predicate. Paralocks was used to encode a simple version of an authority based model, and shown to be verifiable through a simple programming language with its specifications and a static type system that enforces its semantics.

Paralocks expresses policies as sets of locks. Reclassification is done by opening (set predicate to true) and closing (set predicate to false) locks with the Open and Close primitives. They are generally enforced using a type system, but can also be statically verified. The reclassifier criteria are the predicates in each lock, meaning the reclassifier depends on a value in the system to pass a certain check for a reclassification to be triggered.

2.2.3 Progress insensitive noninterference

Askarov and Chong [12] considers IFC from the aspect of attacker knowledge. They propose a system based on channels, where each channel is a security level and the channels are isolated from each other. The sensitivity of a value can be changed by moving the value to a different channel using a setPolicy function, which is part of the language.

Policies can be expressed by creating channels to represent the desired security levels and placing values in these appropriately. Reclassification is done by moving values to other channels, via the setPolicy function. Enforcement is done using a type system. Users of the system can use the setPolicy function to change channels at will, there is no policy that restricts this.

2.2.4 Flow specs

Flow specs [13] are an interesting approach to program verification. They define flow specs for declassification to complement the existing label type-system, and these flow specs can be verified by external programs. The flow specs are separated from the labels to allow for external analysis of the security policies defined in the flow spec. They use a security lattice and labels to annotate values, and define flow specs that define the security policies that the system should adhere to. Flow specs define which expressions are allowed to be declassified.

Policies are described by sets of flow specs, augmented by labels in the code. Reclassification is done in the flow spec, using the Declass function. Enforcement is done by combining a type system for the labeled code and assertion checking based on the flow specs. If the asserting checking is successful, then the declassification is accepted.

2.2.5 Taxonomy

Table 2.1 shows a comparison of the models discussed in this section. Each column corresponds to one of the properties in figure 2.1, and the cells are summarized information from the more detailed sections above. Dashes indicate that there is no policy for the column property.

Model	**Policy expression**	**Reclassifier syntax**	**Enforcement type**	**Reclassifier criteria**
Erasure [10]	$H \searrow^{pred} L$ $\nearrow^{pred} H$	state predicate	Type system	State predicate
Flow/Para-Locks [1]	Set of locks, $\sum \Rightarrow a$	Open/Close locks	Type system	State predicate
Progress-(in)sensitive nonint. [12]	Lattice of channels.	setPolicy function	Type system	—
Flow specs[13]	Lattice of levels + flow specs	Declass, set of flow specs	Type system, assertion checking	Assertion checking of flow specs set

Table 2.1: Analysis table for state-based reclassification models.

2.3 Authority-based reclassification

Authority-based reclassification is a subset of state-based reclassification, in which the predicates consider the authority an executing principal has for a value for reclassification. While this is a subset of the previous section, its prominence in the literature leads us to elevate this to a separate category. Many of the more popular models are based on authority, such as the decentralized label model and its derivatives. These systems usually use principals to describe actors in the system, and use labels to specify the flows that are allowed between principals.

2.3.1 Decentralized label model

The Decentralized Label Model (DLM), introduced by Myers and Liskov [14], is one of the first IFC frameworks based on trust in principals in the system. This forms the basis of the JFlow [15] language, and later the Java Information-Flow (JIF) [16] language. Principal are entities that can interact with data. Labels on data specify which principals own the data and which principals can read the data, for example, the label $\{Bob : Alice\}$ assigns Bob as the owner and Alice as a reader. Data initially has one owner, but via an acts-for paradigm, a principal can place trust in another principal to add them to the owner's label of some data. A declassify function is used to add readers to the set of trusted readers, reducing its confidentiality. This is only allowed if the data owner trusts the executing principal.

Reclassification in the DLM is based on authority. Initially, only the data owner may change the label of a value. The data owner can give other principals acts-for permissions, which will allow them to act on behalf of the owner to change the labels of values.

Policies are expressed as labels between principals. The reclassifier syntax is the declassify function, or for integrity labels the endorse function. Enforcement is done in part by static analysis, but also includes some dynamic checks. The reclassifier criteria is based on the authority the executing principal has over the value, either as the data owner or via acts-for.

2.3.2 Robust declassification

Robust declassification [17, 18] is a general model for declassification that ensures that an attacker cannot influence what is being declassified. This is enforced with integrity labels and ensures that declassification is only done based on data that is trusted enough by the declassifying principal. They base their scheme on a principal-trust based label model like the DLM, and introduce some reclassification criteria that must be met for a reclassification to be valid.

This model extends the DLM to include integrity labels in combination with the confidentiality labels. These integrity labels are used to ensure stricter integrity constraints on reclassification.

2.3.3 Noninterference modulo trusted methods

Hicks et al. [19] introduce noninterference modulo trusted methods. They approach authority based declassification with a function-based approach. Principals specify which functions they trust, so the mechanism can ensure only these declassification functions are used on data these principals has authority over. This is implemented as an extension to JIF, but the model can also be applied to other principal based approaches.

Policies are expressed using the DLM syntax with labels denoting flows between principals. Reclassifier syntax is: *p allows m(p')*. This means that principal *p* allows method *m* to declassify *p*'s data for principals *p'*. Enforcement is based on a type system. Reclassification criteria for trusted methods is stricter than the DLM, declassification functions can be configured on a more granular basis.

2.3.4 Taxonomy

Table 2.2 shows a comparison of the models discussed in this section. Each column corresponds to one of the properties in figure 2.1, and the cells are summarized information from the more detailed sections above.

The authority based models seem to often be either designed to augment the DLM, or be inspired by the way they implement reclassification.

Model	**Policy expression**	**Reclassifier syntax**	**Enforcement type**	**Reclassifier criteria**
Decentralized Label model [14]	Principals, {owners: readers}	Declassify Endorse	Static analysis, Dynamic checks	Data owner, acts-for
Robust declassification [17]	DLM with integrity labels	DLM	DLM	Strict integrity constraints
Non.-int. modulo trusted methods [19]	DLM + declass methods	*p allows m(p')*	Type system Jif extension	Specified by policy

Table 2.2: Analysis table

2.4 Function-based reclassification

Function-based reclassification schemes use functions to transform data in a way that it can be reclassified. These models usually employs some lattice of security levels, and functions are used to move data from one level to another. For example, in some models, the average of a private sequence could be regarded as less sensitive than its input. In this example, the average function would be what we refer to as a reclassification function, or a reclassifier. We find that since labels and data are both transformed by functions, they are more closely linked than in authority-based models.

Function-based models are less prevalent in the literature compared to authority-based and state-based approaches. This might be due to the popularity of DLM based label models, or that there are other facets of IFC that are more interesting for research purposes. For example, when designing a knowledge based model, such as Progress insensitive noninterference in Section 2.2.3, the reclassification scheme is not of high importance. In these cases it makes more sense to use a simple *declassify* construct to move values below in a lattice than to implement a function-based scheme.

2.4.1 Relaxed noninterference

Li and Zdancewic [20] introduce the concept of function-based declassification and prove that this can fulfill the security property relaxed noninterference. They create a framework for defining security policies that include policies for downgrading. These policies are modeled as labels, which are sets of functions that may move a value to a lower sensitivity level. This introduces an extended lattice with intermittent security levels where labels have different functions in their sets and intermittent security policies when they have been transformed by functions.

A downgrading policy is a λ-term (function) that when applied to the value, the value is considered public.

These policies are enforced with a type system in a simply typed λ- calculus. Declassification is done using *Actions*, which are functions that can transform the labels of a value from one label to another. This can be done as labels in the local system are ordered by the restrictiveness of the functions in the downgrade policy.

Policies are expressed as sets of λ-terms. The reclassifier syntax is Action(λ-term). Enforcement is done using a type system. Criteria for reclassification is that λ-terms are compared syntactically.

2.4.2 Reactive information flow control

Reactive IFC uses Reactive Information Flow (RIF) automata to specify restrictions on a value. A RIF automaton is a finite-state automaton whose states map to sets of principals, and in which transitions act as reclassifiers. This means that they can represent mappings from sequences of reclassifiers to sets of principals. These can be constructed to cover changes in both integrity and confidentiality.

Policies are expressed through a RIF automaton for each value. The reclassifier syntax is a *reclassify* keyword that is used to specify that a specific function can act as a reclassifier. Enforcement is done through a type system. The reclassifier criteria is that the automaton restricts transitions in its state based on the functions in its transition set.

2.4.3 Delimited release

Sabelfeld and Myers [21] introduce delimited release as a model that satisfies noninterference, with the addition of escape hatches to allow for declassification. Delimited release is defined as only allowing expressions within escape hatches to declassify information.

Policies are expressed with security levels, forming an ordered lattice. Reclassification is done through the *declassify*(e, l) expression, which moves the expression e to the security level l. Enforcement is done through a type system, and there are no direct restrictions on what can be declassified.

2.4.4 Taxonomy

Table 2.3 shows a comparison of the models discussed in this section. Each column corresponds to one of the properties in figure 2.1, and the cells are summarized information from the more detailed sections above.

Model	**Policy expression**	**Reclassifier syntax**	**Enforcement type**	**Reclassifier criteria**
Relaxed nonint. [20]	Set of λ-terms	Action(λ-term)	Type system	Syntactic comparison
Reactive IFC [22]	RIF automata, conf. and int.	Reclassify keyword and ass. function	Type system	According to automata transition
Delimited Release [21]	Lattice of sec. levels	Declassify escape hatch	Type system	—

Table 2.3: Analysis table

2.5 Survey Conclusion

The state-based models naturally use state-predicates for their reclassifier criteria. We also see similar enforcement schemes for these models. They approach reclassifier syntax differently because they have different focuses in what they investigate.

Authority-based models are often related to the DLM or direct extensions of the model. This is probably because it was among the first authority-based models, and has a relatively mature enforcement mechanism in the JIF system. It seems like others use the JIF system to implement and verify their models because it is simpler to modify the existing system than implement one from scratch.

Function-based models are less prevalent in the literature. An interesting property we observe is that the models we investigated place strict restrictions on ensuring flow altering only happens through functions, but pose little to no scrutiny upon what these functions actually do in practice. Delimited release places no restrictions on what is declassified through the escape hatches. Reactive IFC uses function identifiers to allow reclassification, but this places trust in the name of the function, not how it manipulates data. Relaxed noninterference uses equality checks between lambda terms for reclassification. This is stricter than RIF, but there are not any restrictions on what these lambda terms can contain. The lambda terms are trusted by the model, and created at compile time, so should be reasonably trusted.

In function-based models, the reclassification functions have significant trust placed in them by the model. The models provide semantic guarantees for where data is allowed to flow, except for when flowing through the reclassification functions. We find it natural that there are restrictions placed on what these functions may do with their input.

3 Correctness Criteria and Parsing

Reclassifiers are commonly used to alter the sensitivity of data in IFC models. When building an IFC enforcement mechanism, one would build a system that enforces no reclassification, and allow some reclassification mechanism to break the rules applied to the rest of the code. This means that while one might implement a very strict enforcement mechanism for flows outside the reclassification mechanism, the enforcement mechanism naturally cannot provide the same guarantees for the reclassification mechanism.

This makes them an integral component in the enforcement mechanism of these models. We conjecture that there should be enforcement for reclassifiers that provide guarantees of their safety to strengthen the IFC mechanism.

3.1 Overview

Our goal is to design a system that can be configured to fit many useful reclassification criteria. Our focus will be in the function-based reclassification models, as we find these to be the most relevant in terms of code properties. Authority based approaches can rely more on IFC mechanisms such as data owners to denote reclassification. Function-based reclassification is more reliant on the code of the reclassifier than who is executing, so code correctness is more important.

Designing a configurable system can be approached in several ways. We chose to build a general purpose parser that can extract the necessary dependencies we need to be able to evaluate the configured criteria. We then analyze the parser output according to the configured criteria, and based on this decide if the function should be able to compile or not. Another possible approach could be to design several parsers that only verify a specific criterion, and evaluate the function again for each parser. We have not compared these approaches, but intuitively parsing in as little passes as possible is ideal. The parser currently strategically performs multiple passes in some sections to get better contextual information of subtrees in the code.

Reclassification can be divided into the four categories: declassification, endorsement, erasure, deprecation. We tried to primarily target declassification and endorsement in our criteria design, as we found these to be more common in function-based models. We also find that erasure and deprecation are more closely tied to the overarching IFC mechanism, and as such are less relevant for our solution.

3.2 Correctness Criteria

We strive to define criteria that are both useful for modeling security requirements, but also practical to implement. These criteria should capture the intricacies of many possible reclassification mechanisms, and remain general enough that they could be implemented in other languages as well. Further, since we design criteria to be composable, they should not depend upon another directly.

3.2.1 Aggregates

One of the most general patterns for declassification is aggregation. Generally, an aggregate function like the sum or average value of a sequence will be of lower sensitivity than the individual values in the sequence. To support declassification, we define the LiberalAggregate criterion. Its name is motivated by the opposing StrictAggregate criterion we define later in this section.

Definition 1. A function fulfills the LiberalAggregate criterion if and only if all outputs from the function are aggregates of its input.

Supporting this in code is not trivial. If we look at the function in Listing 3.1, first a loop is used to sum the input vector and then this is divided by the length of the vector. We can see that this is a correct average function over this vector, but which parameters can we design a parser to scan for to detect this? If we define the pattern that an aggregate is created when an iterating value is used to modify some value, this is not strict enough for any reasonable validation.

```rust
fn vec_avg(v: Vec<u32>) -> u32 {
    let mut sum = 0;
    for i in v {
        sum += i
    }
    sum / v.len() as u32
}
```

Listing 3.1: Loop based aggregation example.

Another alternative is to target language constructs that are often used for aggregation. In Rust a common pattern is `seq.iter().fold()`, which can be used to implement the same functionality as in Listing 3.1, but with a stricter structure. However, since the fold function uses a closure (an inline function) to perform the aggregation, it can also be bypassed relatively easily. Nonetheless, the LiberalAggregate criterion is parsed by defining the aggregation variable in the closure as an aggregation, and as such the result of any fold is marked as an aggregate.

```rust
fn vec_sum(v: Vec<u32>) -> u32 {
    v.iter().fold(0, |a, x| a + x)
}
```

Listing 3.2: Function that sums a given vector.

```rust
fn vec_first(v: Vec<u32>) -> u32 {
    v.iter().fold(0, |a, x| if a == 0 {x} else {a})
}
```

Listing 3.3: Function that gets the first value in a given vector.

Listing 3.2 shows an example of a safe aggregate of the vector y, it returns the sum of its values. Listing 3.3 shows a bad aggregate that will only ever return the first value of the vector. While we can see the difference in the semantic meaning of these functions, their syntax is rather similar. They both use iter and fold to perform some function over the sequence, and both of these functions only use the components from the fold. To better define what behavior we want to allow in safe aggregation, we need a definition that can capture the notion of good and bad aggregates. This is based on the notion that any safe aggregate should consider all elements in a sequence when calculating the aggregate. For example, given a sequence `s = {1,2,3,4,5}`, the max of this sequence is the 5th element. However, if we change the sequence to be `{6,2,3,4,5}`, now the max is the 1st element. This shows us that the output of this function is based on all values in the sequence.

Definition 2. A good aggregate function is, for any function f and sequence s, when the output of f can be influenced by a change in any value in s.

Definition 3. A bad aggregate function is, for any function f and sequence s, when the output of f does not depend on all values in s.

Based on the definitions of good and bad aggregates, we want to define a criterion that only accepts good aggregates. However, detecting when a function does and does not consider all values is tricky for many syntaxes. While some subsets of iter and fold might be safe, some variations might not be, as illustrated by the examples in listings 3.3 and 3.2. At the cost of some expressiveness, we define a stricter aggregate criterion.

Definition 4. A function fulfills the StrictAggregate criterion if and only if all outputs from the function are aggregated using the strict syntax in equations 3.1 or 3.2.

Here x denotes every value in the sequence, acc is the accumulated result of the function, and $\oplus$ represents any valid binary or unary operation. A strict aggregate is present in a function that iterates and folds over input, and the folding function/closure has the form:

$$|acc, x|\ acc \oplus x \tag{3.1}$$

$$|acc, x|\ if\ (x \oplus acc)\ \{acc\}\ else\ \{x\} \tag{3.2}$$

These patterns are interesting because they can be used to construct many declassification functions, such as sum, min, max, product etc. The values may switch places inside the function body, but may not be replaced with other values or contain only one of the two values. The values in Equation 3.2 may also switch sides in the if condition, and the branches of the if-test may also switch sides, but they cannot contain other values and must be exactly the opposite of each other.

This syntax enables writing simple aggregate functions such as min, max, sum, etc., but is restrictive enough to detect bad aggregate functions that do not match the patterns. This syntax is also relatively simple to parse, allowing for more portability to other languages.

3.2.2 Trusted and Distrusted Functions

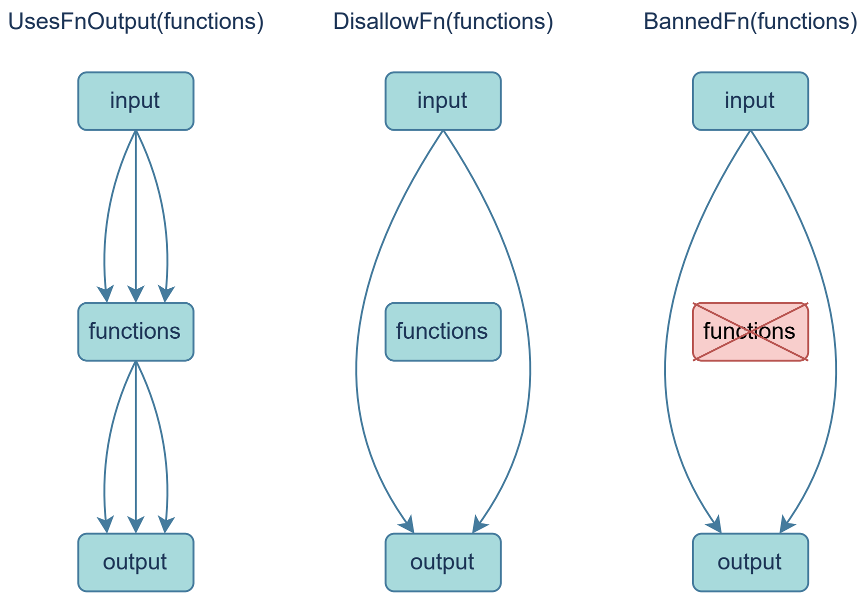


Figure 3.1: Showcase of how the criteria handle their configured functions.

An interesting set of criteria are those that are related to how certain functions are used within the reclassifier function. This includes reclassifiers that want to make sure a function affects the output, and, inversely, that a function does not affect the output.

We define 3 main criteria that work with function identifiers. These are UsesFnOutput, DisallowFn, and BannedFn. These are visualized in Figure 3.1

Definition 5. A function fulfills the UsesFnOutput criterion if and only if, all outputs from the function are outputs from the configured functions.

UsesFnOutput ensures that all outputs from the reclassifier are derived from any of the specified functions. Ideally, the specified functions should act as a gate between input and output, as shown in Figure 3.1. We want to make sure all possible outputs of the reclassifier have passed through any of the given functions. Listing 3.4 shows an example reclassifier that can benefit from this criteron. The function uses the trim and to_lowercase functions to sanitize a string. The UsesFnOutput criterion can be used to guarantee that the output of the function uses these functions if the body of the function is updated in the future.

```
fn string_sanitize(s:&str) -> String {
    s.trim().to_lowercase()
}
```

Listing 3.4: Function that sanitizes a string, an endorsement reclassifier

Definition 6. A function fulfills the DisallowFn criterion if and only if, all outputs from the function are not influenced by the configured functions in any manner.

DisallowFn ensures that all outputs from the function have not been affected by the outputs of any of the configured functions. This means that the disallowed functions cannot in any way affect any of the outputs, but may still appear in the reclassification function. This could be useful for functions that we don't mind being present in the reclassifier, but cannot affect the output in any form. For example, one could allow a function call that logs the first element of a vector, but they don't want the output of the function to be the first value in the vector.

Definition 7. A function fulfills the BannedFn criterion if and only if, none of the configured functions are present in the function.

BannedFn ensures that none of the configured functions appear in the reclassifier. This criterion is simpler than the two previous in that it does not rely on tracing flows for enforcement, the presence of the function call is enough to reject a reclassifier. This can be useful for many reasons, for example to ensure that no functions that can write to files are called, or to ensure that unsafe error handling like unwrap is not allowed.

3.2.3 Dynamic Criteria

We have discussed several criteria that can be detected through static analysis. However, these assume some things about the function inputs that might lower their integrity. For example, the aggregate based criteria assume that the aggregate value of the input sequence is less sensitive than the input. This is not always true, for example if a sequence only has one element in it, the element and the average of the sequence will be the same value.

To address this, we present the DynamicAssert criterion. This criterion inserts a run-time check into the reclassifier that verifies that a condition is met before executing the function. The possible conditions will be discussed more in future work, but the current condition is to ensure that the vector is sufficiently long.

```rust
#[verify_reclassifier(DynamicAssert)]
fn vec_avg(v: Vec<u32>) -> u32 {
    if v.len() < 5 {panic!("")}; //inserted by the crierion
    v.iter().fold(0, |a, x| a + x)
}
```

Listing 3.5: Dynamic criterion example.

The example in Listing 3.5 shows a function that calculates the average of a vector. The DynamicAssert criterion on this function inserts a range check for the input vector to ensure its length is long enough for an aggregate function to provide a reasonable downgrade in information.

The problems we discussed in this section relate to data entropy. Aggregates of low entropy data may reveal more information than is permitted by the security policy. We would like to define a dynamic criterion that ensures input entropy is sufficient for safe aggregation, and discuss this more in future work.

3.2.4 Static Blackbox Analysis

Another possibility we considered was testing that a reclassifier is sufficiently transformative by running the function many times at compile time and evaluating its outputs. If the function does not perform sufficient transformations, we can abort the compilation and report the results. Unfortunately, this could incur large amounts of compile time overhead, but would allow us to evaluate the safety of the function from another angle. Another concern is that some functions might be very complex, and computing sufficient permutations of them could be near impossible. This could lead to incorrectly blocking programs that should pass because the compilation never finishes.

This runs into some of the challenges found in formal verification as well. The challenge is often to find practical ways of executing exhaustive tests, not to create the test scenarios.

We see that there are several projects that are working on instrumentation to find unwanted or undefined behavior not covered by the rust compiler. Much of this is focused on the use of unsafe blocks, which are often used to do things the compiler cannot reason about correctly. Unsafe blocks can be used for performance enhancing raw memory manipulation or interfacing with other languages. Unsafe blocks are often used in libraries used by many other users and programs. MIRI [23] executes your program and performs exhaustive tests to ensure that unsafe blocks perform as intended. Loom [24] is a concurrency test harness that runs a test many times while mutating the concurrent execution state.

Another interesting related tool for verification is Kani [25]. “Kani is an open-source verification tool that uses automated reasoning to analyze Rust programs.” [1]. Kani uses proof harnesses to analyze programs. Proof harnesses are similar to test harnesses, especially property-based test harnesses, but can catch other types of errors.

Integrating such tools into either the macro, or bootstrapping such tools to be IFC aware could be a way to approach static analysis.

1. https://model-checking.github.io/kani/getting-started.html

3.3 Parser Design

We present a parser for function Abstract Syntax Tree (AST)s that can parse the necessary information to verify the defined correctness criteria. We base this parser on Rust syntax primarily, but the ideas can be extended to other languages and syntaxes as well. The main Rust specific syntaxes we target are the notions of iterators and folding, which could be described as the most functional parts of the language. This is mostly because these are stricter than other language expressions, the syntax we support is enforced by the type system. For configurability, we want the parser to always perform the same operations and collect the same information, that then can be analyzed to validate the function. The output of the parser is a set of vectors that contain useful information about the function. More details around this can be found in the implementation chapter.

```
fn vec_avg(v: Vec<u32>) -> u32 {
    let avg = v.iter().sum() / v.len() as u32;
    let avg = v.first();
    avg
}
```

Listing 3.6: Reassignment of valid output to invalid output.

A design decision we implemented in our parser is to disallow duplicate variable names in the entire reclassifier function scope. This gives us the guarantee that variables cannot change in meaning throughout the function, such as in Listing 3.6. The example shows a valid aggregate of input assigned to the avg variable, but then an invalid aggregate is reassigned to the same value. Detecting this would require more advanced parsing than the dependency based approach we are using currently. To partially fix this, we disallow any reassignments by not allowing the reclassifier to contain duplicate let assignments. This means that syntax such as `let a = 5; let a = 7;` would be rejected, but leads to stricter guarantees.

3.3.1 Flow-based Dependency Analysis

To detect and verify the criteria described in this thesis, we use dependency analysis. Dependencies in this context relates to which variables may affect the value in others. For example, if a variable a is initiated by a function f called on b, a will depend on both f and b. Additionally, a should inherit any dependencies *b* has, since it may be derived from other variables that have properties we want.

This also applies to blocks and scopes that may affect others. For example, the condition in an if test will affect which of its blocks execute, and potentially affect other variables. In Rust, if-tests are expressions that can output values, so a value initiated by an if-test would be dependent on the variable in both blocks and the condition. However, this also applies to when variables are changed within scopes. For example, if an if-test mutates a value defined in an outer scope, that value should be derived from the condition in the if-test. This is also true for value reassignment, while and for loops, and other common language mechanisms. For each of these, we need to carefully track which variables affect each other.

This can be described as additive dependency analysis, all potential dependencies are tracked for any variable. While this is very useful for criteria such as DisallowFn, this is not accurate enough for the more complex criteria such as UsesFnOutput. We will discuss this more in evaluation and future work.

3.3.2 Parser output analysis

After the parser has collected information about the code inside the function, we parse this output in regard to the configured options. It is natural to evaluate each of the configured criteria separately, as their verification process varies quite a bit. Most of these validators are based on the dependency analysis performed by the parser, while some validators are simpler and can be evaluated from other information collected by the parser.

We add some implicit criteria as well to allow us to eliminate some edge cases. We ensure that all variable names are unique, if the parser detects any duplicate variable names we output a compile error for the latter variable. We also ensure that the same function cannot be both required and or disallowed. Another implicit validation we do is to output compile errors when unimplemented language items are found in the function. This allows us to not implement functionality such as importing other libraries etc., and trust that any such constructs are not present in the function.

The parser collects variables that are aggregations of input in a vector in its output set. For every output from the function, we check if there are any aggregated variables with matching identities, or if the output is derived from any such aggregates. We need to ensure that all outputs are derived from or are aggregations of the input. For strict aggregates the process is similar, and since we know that finding strict aggregates is stricter we know that this will yield safer results.

The function-based criteria are verified by inspecting the relevant vectors in the output set. Since each of them have separate meanings, we collect a vector for each of them based on the configured criteria. For the UsesFnOutput criterion, we verify that all outputs are at some point in the reclassifier derived from any of the configured functions. Inversely, for DisallowedFn we verify that none of the outputs are derived from the configured functions. For BannedFn we simply verify that none of the configured functions are present in the reclassifier.

4 Implementation

We have implemented a configurable enforcement mechanism for our defined correctness criteria using Procedural macros in Rust. In this chapter, we will go into detail on what motivated using procedural macros, explain how our implemented parser works and what information it gathers for analysis.

4.1 Rust Background

When building a system today, one of the more prominent programming languages of choice is Rust. Rust is designed with high standards for memory safety and data safety. The Rust compiler ensures both memory and data safety at compile time, with help from a few language restrictions.

The memory model is based on the notion of ownership [26]. The ownership model means that every value can only have one owner variable. If the value is assigned to another variable, the ownership of the value is moved to the new variable. Since the compiler can infer that any access of the first variable after the move can lead to a data race, it will fail to compile any attempt at access after the move. Based on this, the compiler knows when data needs to be allocated, when a value is moved from one address to another, and that no two variables can own the same value. From this it also can infer when data can be freed, as when the value goes out of scope it cannot be accessed again.

The borrow checker allows for references to values to be safely passed around the program without invoking explicit ownership transfers. References come in a few forms, and impose a few rules to be able to guarantee safety. The simplest reference is immutable, and thus there can be an unbounded amount of these concurrently without any safety concerns. Mutable references are possible, but these are restricted to one and only one reference at a time. Also, if there are any immutable references, there may not be any mutable references. This ensures that data races between references to the same value is not possible, while still allowing for mutable references to alter the value and safe concurrent reads.

Another useful restriction in Rust is opt-in mutability. By default, every variable is immutable, meaning to mutate a variable it needs to be declared using the mut keyword. This means that the compiler can guarantee that concurrent reads from an immutable variable are safe, and can impose stricter concurrency requirements on mutable values.

Procedural macros are a part of the Rust language that allows for expressive meta programming. Macros in themselves are not unique to Rust, C macros have been a staple in many codebases for years. The unique thing about Rust macros is how they are integrated into the compiler such that they can perform more advanced functionality. While C macros will simply perform string substitution, Rust macros can execute an almost arbitrary program to affect the output. Procedural macros in Rust are separated into three categories, function-like macros, derive macros and attribute macros. Function-like macros can be used to create functions with more flexible inputs, an example is the format and print macros. Derive macros can be used to manipulate structs or add trait implementations of structs programmatically. Attribute macros can be used to manipulate items, such as functions, based on the contents of the item and a set of supplied attributes.

4.2 Enforcing correctness criteria in Rust

We would like the criteria and enforcement of them to be language agnostic, but for the purposes of this thesis we chose to implement a proof of concept enforcement mechanism in Rust. Based on the previous research, we chose to implement this with procedural macros. There are a couple options to using procedural macros, such as through a set of linters in Clippy or creating an external tool, but procedural macros are the most ergonomic to work with and provides the best user feedback.

Procedural macros are powerful for compile-time static analysis, but are limited to parsing tokens. This means that the static analysis depends on the user importing and using the macro to tag the functions they want to use as reclassifiers. This is a potential problem that one might need to solve when using such a macro in a larger IFC framework. The focus of this thesis is more towards the correctness of the code within a reclassifier, so we assume that there is some way of ensuring that all reclassification functions used by a mechanism are verified.

Listing 4.1 shows the first idea of an enforcement mechanism. The example shows an enforcement API that takes a function, uses a macro to parse the function, and create a reclassifier variant of that function with some slight alterations. This is to our knowledge not possible with Rust procedural macros, but could be an alternative in other more dynamic languages. We can't do this in Rust because the macro will only see the function name passed to the macro, and has no way of finding the function body.

```rust
fn vec_avg(v: &Vec<Labeled<i32>>) -> Labeled<i32> {
    let label = v.iter().max_by(|x| x.label);
    let avg = v.iter().sum_by(|x| x.val)/v.len();
    return Labeled::new(avg, label);
};

let some_vec = vec![private(1), private(2), private(3)];
let declassify_avg = validate_declassifier!(vec_avg);
let public_avg = declassify_avg(some_vec);

assert!(vec_avg(some_vec).label == Private);
assert!(declassify_avg(some_vec).label == Public);
```

Listing 4.1: Example use case for macro that sometimes declassify data

A more viable approach in Rust is to use procedural attribute macros. These macros can be applied to many expressions, most notably functions, and then has access to both read and modify the original code. Using libraries such as syn [27] and quote [28], we can parse the AST inside the function to analyze how information propagates and flows throughout the function.

```rust
#[verify_reclassifier(LiberalAggregate)]
fn some_avg_fn(v: Vec<u32>) -> u32 {
    v.iter().fold(0, |a,x| a+x) / v.len() as u32
};
```

Listing 4.2: Realistic example using procedural macros.

Listing 4.2 shows how the syntax would look like using procedural macros. This shows a declassification function that gets the average of a vector. The attribute passed to the macro specifies which criterion should be used to verify the function. We will discuss if this example satisfies our implementation or not in the evaluation section, but by the definition of the criterion, this example is valid.

4.3 Parsing the Rust AST

Procedural attribute macros have access to a TokenStream of the function they are applied to. Using the Syn crate, we can parse this function into an abstract syntax tree (AST) of Rust syntax. Parsing this AST can give us access to insights into the flows within the function, where variables flow to another, function calls on variables and what the function returns.

The AST is however quite complex. The syntax we are mostly concerned with consists of Statements, Expressions, Items, and Patterns. A function can be parsed as a function-item, which contains information about its input types and variables, potential output type, and the function body. The function body is a set of statements, which again can be Items or Expressions, which can contain more Expressions or Patterns etc. This leads us to conclude that we need a recursive parser to be able to extract the information we need for analysis.

To allow the parser to accumulate values that fulfill certain conditions, we pass a mutable "Conditions" struct down the recursive calls, and append variables that fulfill certain conditions to vectors within the struct. This allows the recursive functions to influence the final state at any depth. We also need a way to propagate values up the recursive calls. This could either be done by returning values from the recursive functions, or on a list in the Conditions struct. We chose the latter, but both approaches are probably viable.

A tradeoff between correctness and expressiveness often needs to be carefully considered. For our purposes, we have a tradeoff between the syntaxes we can safely evaluate as correct and allowing more expressive functions. For the purposes of reclassifier correctness, we will try to be more conservative. We would much rather the macro reject a function as a false positive than allowing true negatives to compile. The macro should be strict with what it allows, and lead users to try implementing the same functionality using alternative methods instead.

4.4 Dependency propagation

An important detail in our parser is that we propagate dependencies when initializing or reassigning variables. All local variables are tracked in a list, and if a variable is used to create another, we need to track this. This is relevant for all criteria we implement, and is used for verifying most of them. For example, DisallowFn uses dependency analysis to ensure that functions have not affected output any variables.

We define a data-structure called Context-Points (CXPT), which consists of a variable-identity and a set of variable-identities that have affected that identity, its dependencies. The parser creates a set of CXPT vectors, and each vector in this set corresponds to a property we are interested in for analysis. Among the interesting properties are which CXPTs are returned, if there are any aggregates, if there are any required, disallowed or banned functions, and if any unsupported syntax is present etc.

The UsesFnOutput criterion is not trivial to fully parse using dependency analysis. Since dependencies propagate very easily in our parser, we could not find a safe way to guarantee that all outputs are only the product of the supplied functions. Ensuring that all paths from input to output passed through the given functions requires more information than the dependency analysis can supply. We hypothesize that a graph-based parser could be better suited for this criterion, more on this in the future work section.

4.5 Configuring Attribute Macros

Attributes in the procedural attribute macro is a simple TokenStream. Unlike the function, since there is no Rust standard for these attributes, the common parsing libraries do not have build in functions for parsing many of the possible syntaxes. This also means that we can decide how complex or simple we want to make the accepted arguments. A simple syntax could be a comma separated list of arguments.

For our needs, we need to be able to parse single arguments and arguments with attributes inside parenthesis. An example of this syntax can be seen in listing 4.3, with a single argument for LiberalAggregate, and a configured argument for BannedFn with function names inside the parenthesis. The configuration is parsed by looking at first if there is an identity and then if there is any parenthesis behind the token. If there is a parenthesis, we first consume the identity, then the parenthesis and then the comma separated list inside. If the identity is not followed by a parenthesis, we can simply consume the identity and continue parsing.

```
#[verify_reclassifier(LiberalAggregate, BannedFn(unwrap, min))]
fn some_fn(...) {
    ...
};
```

Listing 4.3: Example use case for macro that sometimes declassify data

The parser first consumes the list of criteria identities, and then maps them to a configuration struct. This config struct has a boolean value for each possible single criteria, and a vector of identifiers for each of the configured criteria. This struct is more efficient to access throughout the parsing and correctness evaluation than the list of parser options.

4.6 Resolving criteria conflicts

Detecting invalid criteria configurations is another interesting implementation detail. The UsesFnOutput and DisallowFn criteria are in essence opposites, and configuring both with the same function could lead to undefined behavior. We see two possible approaches to this, define a priority between the criteria or block conflicts from compiling. In the implemented parser we have implemented both these approaches: banned functions will always be banned, and configuring UsesFnOutput and DisallowFn with the same function will be rejected.

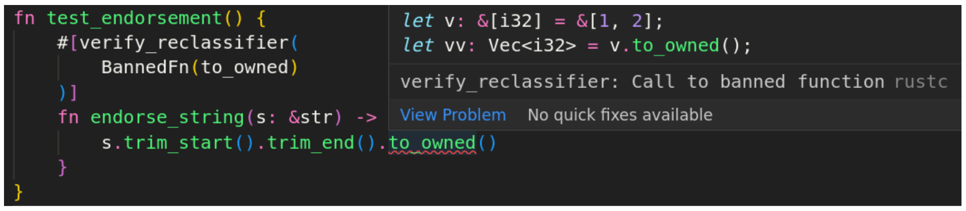
We want the criteria parser to detect when the same function is configured for both UsesFnOutput and DisallowFn. The function in listing 4.4 shows the syntax we should reject when parsing for configurations. The implementation correctly detects and stops this, and gives the user feedback on the incorrect syntax.

```
#[verify_reclassifier(UsesFnOutput(min), DisallowFn(min))]
fn some_fn(...) {
    ...
}
```

Listing 4.4: Evaluating invalid configuration of criteria.

4.7 Error reporting

Error reporting is done by inserting compile errors into the tokenstream output. The Rust compiler uses spans to keep track of where identifiers and other items are located in a file. This allows us to set the target span of the compile errors to where the invalid syntaxes are in the function. We do this for every invalid syntax, and if we are missing valid syntax, we can also point the compile error at the function name. Figure 4.1 shows a snapshot from Visual Studio Code, where the function `endorse_string` is configured with the BannedFn criterion with the `to_owned` function. We see that the macro generates an error as expected, and the code does not compile.



```
fn test_endorsement() {
    #[verify_reclassifier(
        BannedFn(to_owned)
    )]
    fn endorse_string(s: &str) ->
        s.trim_start().trim_end().to_owned()
    }
}
```

```
let v: &[i32] = &[1, 2];
let vv: Vec<i32> = v.to_owned();
verify_reclassifier: Call to banned function rustc
```

Figure 4.1: Error reporting example.

This leads to a more ergonomic development experience, the developer can get real-time feedback from the parser if the flows are adhering to the configured policy. For clarity, the macro prepends its name to all error outputs to allow users to more easily locate the origin of the error.

4.8 Implicit returns in Rust

Rust implements implicit return from blocks. This is showcased in listing 4.2, as there is no semicolon after the last expression, the result of this expression is returned from the function. This is a nice feature from a developer perspective, but leads to some complexity in our parser.

To determine the outputs of the reclassifier we both need to scan for return expressions, and parse the last expression in the function body. Finding return expressions is trivial for the parser, as it already parses each node in the function AST. Parsing the last expression leads to some interesting behavior from the parser. If we again look at the example in listing 4.2, the return statement contains several variables. These variables are v, a, x, and the function is a valid reclassifier. Normally the parser bases its analysis on the output functions, and tries to detect that they are in this case aggregates of the input. The difficult part then is to detect that while the input is part of the output, aggregated values of the input are also part of the output, and as such it should be allowed.

5 Evaluation

This chapter describes the experiments used to evaluate the implemented parser. These experiments consist of various reclassifier functions, designed to test each of the supported correctness criteria. We use a test harness that compiles each reclassifier in the test set, and checks if they compile successfully or not.

While it would be preferred to base our experiments on real world code, as there is little to no IFC work in Rust, finding exact examples is not simple. Instead, we construct a set of test functions for each criterion, with examples of simple cases we expect them to perform well on and more advanced cases where we could struggle. For brevity's sake, we will summarize the tests for each criterion and show examples of valid reclassifiers that pass and invalid reclassifiers that we catch. As we discussed in 4.8, an interesting syntax is Rust is that expressions implicitly return their last expression. Several criteria in the implementation support implicit returns, so we also present some tests where this is stressed.

5.1 Aggregate criteria

We have defined two aggregate-based criteria, one that assumes that folds aggregate safely and one that restricts the syntax for what is safe within a fold. Both of them rely on the iter into fold pattern, and the experiments reflect this.

The first example we present in Listing 5.1 shows two functions that both sum a vector using folds. The difference is that the first returns the result inline, and the other stores the result in a variable and returns the variable. These are both valid reclassifiers, and the parser allows them to pass as expected.

```rust
#[verify_reclassifier(LiberalAggregate)]
fn reclassifier_inline(v: Vec<u32>) -> u32 {
    v.iter().fold(0, |acc, x| acc + x)
}

#[verify_reclassifier(LiberalAggregate)]
fn reclassifier(v: Vec<u32>) -> u32 {
    let r = v.iter().fold(0, |acc, x| acc + x);
    r
}
```

Listing 5.1: Aggregate test, inline and not inline.

Another example on this same criterion is shown in Listing 5.2. Here we see the variables a and x, where a is an aggregate of the input v and x is the value of a. This tries to test if we correctly can determine that, since a is a valid aggregate, x is also a valid aggregate. The parser correctly assesses that this is a valid reclassifier.

```rust
#[verify_reclassifier(LiberalAggregate)]
fn flow_from_aggregate(v: Vec<u32>) -> u32 {
    let a: u32 = v.iter().sum();
    let x = a;
    x
}
```

Listing 5.2: Dependency flow works as intented for aggregates.

The previous examples are of valid reclassifiers, but let us now look at some examples of the parser catching invalid reclassifiers. The function in Listing 5.3 returns the first value of the vector. The parser correctly assesses that this is not a valid aggregate, both as shown in the function or directly inline.

```
#[verify_reclassifier(LiberalAggregate)]
fn reclassifier(v: Vec<u32>) -> u32 {
    let r = v.first().unwrap().to_owned();
    r
}
```

Listing 5.3: Example of invalid reclassifier.

Listing 5.4 shows the same function but with a valid reclassifier present in another variable in the function. This is interesting since there could be cases where the parser finds a valid aggregate in the function and assumes the function is valid. However, the implemented parser correctly finds that the function does not return a valid aggregate.

```
#[verify_reclassifier(LiberalAggregate)]
fn agg_present_not_returned(v: Vec<u32>) -> u32 {
    let a : u32 = v.iter().sum();
    let r = v.first().unwrap().to_owned();
    r
}
```

Listing 5.4: Reclassifier is not valid unless the aggregate is returned, not only present.

The next example in Listing 5.5 shows a case where the LiberalAggregate criterion is not strict enough to capture the unwanted behavior. The function returns the result of a fold, but the result of the function will always be the first value in the vector. The implemented parser does not mark this snippet as invalid, despite it not fulfilling the definition of the criterion. Examples like this are what motivated the StrictAggregate criterion, by allowing less valid syntaxes we can more closely monitor the behavior of the function.

```rust
#[verify_reclassifier(LiberalAggregate)]
fn circumvented(v: Vec<u32>) -> u32 {
    let a = v.iter()
        .fold(0, |acc, &x| if acc == 0 {x} else {acc});
    a
}
```

Listing 5.5: Motivating example for strict aggregates.

In Listing 5.6 we see examples of the two valid syntax patterns allowed under the StrictAggregates criterion. The places of *acc* and *x* can be swapped and other operators can be used, but no other variants are allowed. These are correctly identified by the parser, and the tests in the Listing compile as expected. Implicit return is not implemented for StrictAggregates, meaning functions will need to assign strict aggregates to variables and return these to be valid.

```rust
#[verify_reclassifier(StrictAggregates)]
fn strictness_1(v: Vec<u32>) -> u32 {
    let r = v.into_iter().fold(0, |acc, x| {acc + x});
    r
}

#[verify_reclassifier(StrictAggregates)]
fn strictness_2(v: Vec<u32>) -> u32 {
    let r = v.into_iter().fold(0, |acc, x| {
        if acc > x { acc } else { x }
    });
    r
}
```

Listing 5.6: Valid strict aggregate syntaxes.

The next example in Listing 5.7 shows some of the invalid syntaxes according to the StrictAggregates criterion, and all of these are detected as expected. The output of the function will either be the first or the last value in the vector, which violates the definition of a good aggregate and the defined syntax.

```rust
#[verify_reclassifier(StrictAggregates)]
fn circumvent_get_first_3(v: Vec<u32>) -> u32 {
    let first = v.first().unwrap().to_owned();
    let agg_first = v.iter().fold(first, |acc, &x|
        if acc > x {
            first
        } else {
            x
        }
    );
    agg_first
}
```

Listing 5.7: Stressing validness in strict aggregate.

5.2 Function criteria

We have defined three function-based criteria. These vary in how they are enforced, and as such their tests vary as well. All these criteria work with implicit returns, as is shown in Listing 5.8. In this example, the first function compiles as it correctly finds the required max function. The second and third function do not compile, as they find disallowed and banned functions in the output.

```rust
#[verify_reclassifier(UsesFnOutput(max))]
fn uses_fn(v: Vec<u32>) -> u32 {
    *v.iter().max().unwrap()
}
#[verify_reclassifier(DissallowFn(max))]
fn dissallow_fn(v: Vec<u32>) -> u32 {
    *v.iter().max().unwrap()
}
#[verify_reclassifier(BannedFn(max))]
fn ban_fn(v: Vec<u32>) -> u32 {
    *v.iter().max().unwrap()
}
```

Listing 5.8: Implicit return tests for function-based criteria.

The most advanced of these criteria is UsesFnOutput. By definition, it should only allow flows from input, through a specific function, and to output. For simple functions such as the one shown in Listing 5.8, the parser detects the flows correctly. However, because the nature of the dependency propagation is additive, a malicious actor can create functions that bypass our parser. Listing 5.9 shows a function that is supposed to trim a string and return an all lowercase output. The function performs the expected behavior and combines this with the input, but then extracts only the original input and returns this. We can see that this should not be allowed, as the output does not comply with the criterion, but because the parser thinks that the output depends on a cleaned string, it misclassifies the function as correct. This is a challenging problem we discuss more in future work.

```rust
#[verify_reclassifier(UsesFnOutput(trim, to_lowercase))]
fn bypass(s: &str) -> String {
    let trimmed = s.trim().to_lowercase();
    let original = s.clone();
    let passby = [original, trimmed].get(0).unwrap();

    passby.to_string()
}
```

Listing 5.9: Bypass that creates array of two elements and extracts one of them, but parser keeps both dependencies from the vector.

DissallowFn is not fallible to the same bypass. By definition, this criterion is simpler to verify, as it does not need to ensure the output is exactly derived from any of the given functions, but that none of the configured functions have affected any of the outputs. This matches the technique used by the parser much more closely, and we believe the implementation of this criterion to be much closer to its intended design. In Listing 5.10 we see the same function that was used for the bypass example, but with the DisallowFn criterion instead. The parser correctly identifies that the *passby* value is derived from both *trim* and *to_lowercase*, and as such blocks the function from compiling.

```rust
#[verify_reclassifier(DisallowFn(trim, to_lowercase))]
fn bypass(s: &str) -> String {
    let trimmed = s.trim().to_lowercase();
    let original = s.clone();
    let passby = [original, trimmed].get(0).unwrap();

    passby.to_string()
}
```

Listing 5.10: The same function does not compile with the DisallowFn criterion instead.

The next example, shown in Listing 5.11, shows the difference between the DisallowFn and BannedFn criteria. We see that even though the configured function is present in the reclassifier, it does not flow to the output. This reclassifier satisfies the DisallowFn criterion, but the BannedFn criterion rejects the reclassifier since the configured function is present in the function. For both criteria, the implementation works as expected.

```rust
#[verify_reclassifier(DissallowFn(min), BannedFn(min))]
fn uses_fn_1(v: Vec<u32>) -> u32 {
    let _min_val = *v.iter().min().unwrap();
    0
}
```

Listing 5.11: Function with present function that does not flow to output.

The BannedFn criterion is simpler to validate than DisallowFn, since it does not need to check if the function flows to output or not. According to the BannedFn criterion, if the configured function is present in the reclassifier, the reclassifier should be rejected. The most tricky part of its implementation was that when the function was inside expressions that were not implemented in the parser, so we present the example in Listing 5.12 which targets this case. The fix for this was both to reject unimplemented syntax, and also implement more syntaxes that can contain function calls. In the implemented parser, the reclassifier in Listing 5.12 is rejected as expected.

```
#[verify_reclassifier(BannedFn(min))]
fn uses_fn_4(v: Vec<u32>) -> u32 {
    let min_val = {
        let z = v.iter().fold(0, |acc, x| {
                std::cmp::min(*x, acc)
            });
        z
    };
    let zz = min_val;

    zz
}
```

Listing 5.12: Function with banned function in deep nested block.

5.3 Dynamic criteria

We briefly showcase the implemented dynamic criterion. The example in Listing 5.13 shows an average function that uses the dynamic criterion to verify that the vector is of sufficient length, and two calls to this function. At runtime, the first function call will succeed as normal as its length is sufficient, while the second will evoke a panic and stop the program since its vector is too short.

```
#[verify_reclassifier(DoDynamicAsserts)]
fn dynamic(v: Vec<u32>) -> u32 {
    let r = v.into_iter().fold(0, |acc, x| {acc + x});
    r
}

let sum1 = dynamic(vec![1..10]);
let sum2 = dynamic(vec![1..3]);
```

Listing 5.13: Dynamic criteria example. sum1 will execute and sum2 will force exit the program before it executes.

6 Discussion and Future Work

6.1 Weaknesses with current Dependency Analysis

An issue we see is that some criteria can be bypassed by malicious code. In Listing 5.9, we showcased an example where the dependency analysis we described in parser design was not sufficient to detect the invalid reclassifier. This boils down to the additive nature of the dependency analysis. The example in Listing 5.9 creates an array with $[original, trimmed]$, which has the dependencies of both the values. Then a new value is created by extracting the first value of the array, which again keeps the dependencies from both $original$ and $trimmed$, but now has the value from $original$.

Ideally, the dependency tracing would be more accurate, to the degree that it would correctly follow the flows from the previous example. One potential way we see to help alleviate this problem is to alter the design of the dependency analysis to be more recursive. Instead of the dependencies being inherited, we could store only references to the dependencies, which could be recursively parsed afterwards. This would allow us to better track where dependencies originate from. However, this would not address the problem where we need to remove dependencies from variables.

Detecting when dependencies should be removed is not trivial. Tracking which parts of a value have which dependencies can be difficult for all operations. Based on the example we used previously, we could potentially track which index in the array has which dependencies. Imagine then that we apply a function that swaps the position of elements. The procedural macro has no way to know how the function has modified the array, and as such the index tracking is now outdated. If we then take the item at index 0 and assume it has the same dependencies, it would instead the dependencies of the *trimmed* variable but the value of the *original*.

One idea we had was for the parser to validate variables in their original context, such that when variables are derived from others, this could be used for more accurate dependency flow. Say we define two states for a variable, valid output and invalid output. This state depends on the criteria, for example a variable such as *trimmed* would be a valid output, and *original* would be an invalid output. When creating a new variable based on preexisting variables, its state would be a combination of the states of these variables, if all variables are valid their combination is valid, otherwise it is invalid. This could show that the output of a function is only composed of valid outputs, and would reject the example from earlier as *passby* would be considered invalid.

Another possible approach is to alter the detection scheme entirely. We conjectured in design that using a graph-based approach to ensure function outputs are used as a reclassifier is a valid approach. We think creating a traversable graph could help in some cases, but we still think this would be difficult for all syntaxes. The same problems seem to be prevalent here as in the dependency propagation, exact tracing of values is required and not trivial. This could possibly require run time coupling of labels and values, or some other mechanism we have not considered.

6.2 Information Flow Control in Rust

In earlier work, we demonstrated that there is promise in implementing IFC mechanisms in Rust [29]. This included exploring ways to express label models with types and macros, a run time mechanism for ensuring immutability for dynamic labels, and explored tracing function flows with static analysis.

On labels, our previous work [29] discussed the differences between static and dynamic labels. It concluded that dynamic labels are practical in Rust, to support label models that require updates on the labels. This means treating labels as values, and means that labels can be modified when values are modified. However, this introduces the need for some mechanism that ensures labels are not maliciously modified and are respected in the system.

The proposed solution to this is a run-time *controller* that holds ownership of every labeled value, and ensures that a principal has sufficient permissions to modify value-label pairs. The only way to modify the value-label pairs would be through applying functions to the pair, at the discretion of the controller based on the label and the acting principal.

Further, the previous work investigates using procedural macros to verify statically labeled functions. The attributes in the macro are used to label the inputs to the function, and then a simple flow analysis is implemented to determine the label of the output. This was not directly interoperable with the dynamic controller, but some alternatives to facilitate this were proposed. One option was to insert dynamic checks that ensure that the dynamic label matches the static one. Another proposed dynamic enforcement scheme was to statically create a dependency graph that describes all the flows of the function, which would be stored somewhere in the function to be accessed at run time. This was not explored further, but would potentially allow for dynamic flow analysis at run time. This previous work does not discuss reclassification in much depth. Finally, we discussed briefly how these components can be combined, and proposes to use dynamic labels in the controller with some form of static or dynamic analysis. This would compose a simple implementable IFC system designed for Rust.

6.2.1 Integrating reclassification criteria

The capstone we refer to as previous work presents some interesting components that could be used to build a larger system. We find it interesting that we could use the ownership model to enforce immutability at run time. The integration between static analysis and dynamic labels needs more work to be feasible for standalone systems.

This thesis has been mostly focused on reclassification, which is something the capstone did not go into much depth on. Integrating reclassification into the proposed system from the capstone presents some interesting challenges. Firstly, the system described in the capstone is more focused on DLM style IFC models, with principals and labels being important. The reclassifier correctness criteria are more geared towards function-based reclassification style models. Since the DLM style models often integrate reclassification into their semantics, the reclassifier functions from the scope of our correctness criteria are not as necessary.

6.2.2 Future work on IFC in Rust

We find that there is potential to build elegant IFC mechanisms that target both more function-based approaches and more authority-based approaches. We have not explored state-based approaches, but could see scenarios where this would be useful, and the potential for designing this is also present.

While it is maybe simpler to target pre-existing IFC models, it could be interesting to design a model with the guarantees Rust can provide in mind. The borrow checker and ownership model in Rust are examples of advanced static enforcement mechanisms that work well with the Rust language, and building off of them to create a Rust specific IFC model could yield interesting results.

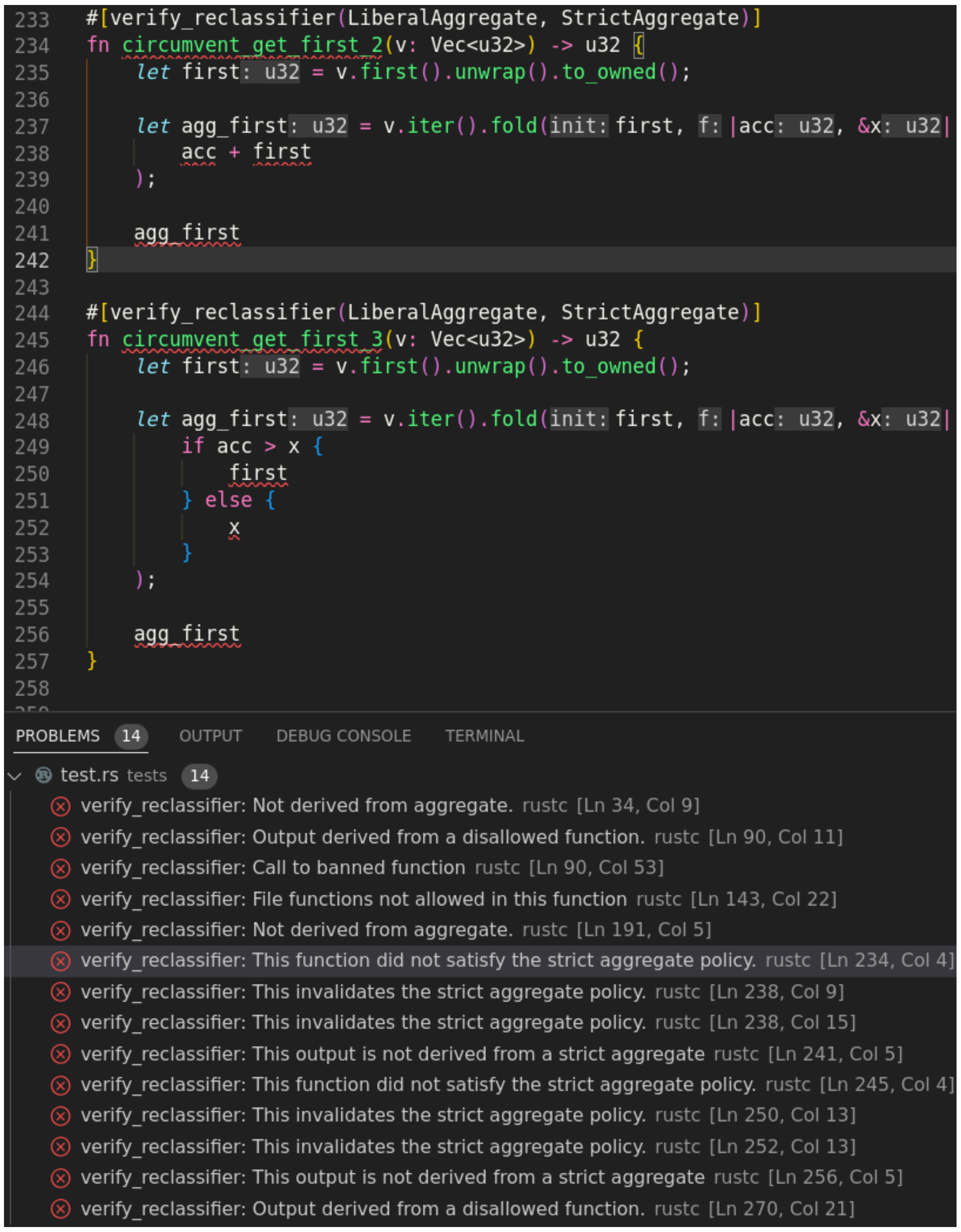
6.3 Discussing implementation details

The parser was implemented incrementally based on small snippets of code at a time. This means that parts of the implementation could be done simpler using the more expanded dependency analysis that was added later in the process. For example, several of the result parsing functions search for context points with the same identifier in the other lists to find more dependencies.

As part of the attribute parsing, at some point we need to map from an identity string that represents a criterion, to the enum type that represents that criterion. To make this more type-safe, we chose to implement a derive macro that creates static strings for each enum variant. This allows us to pattern match using these strings, and create the appropriate configurations. An alternative approach would be to create macros that instantiate from the parsed config, but this ended being more complicated than creating static strings. Both approaches bring more type safety to the parsing, but creating instances directly from the macro results in a nicer API.

6.4 Alternative uses of the macro

An alternative use-case we can imagine is using the macro as a standalone tool for writing and validating reclassifier functions. This would not provide strong guarantees for a full system, but more be useful for configurable analysis of a function as it is being developed. We know software is hard, and having specialized analysis tools is helpful for creating safe programs. The macro is helpful for this because it will emit compile errors when and where the reclassifier is invalid, which can be displayed directly in the Integrated Development Environment (IDE). We saw an example of this in Figure 4.1, and present another figure of this in Figure 6.1. This is a snapshot from one of the development test files, where there are approximately 14 errors generated by the macro. All these show up both inline with red squiggly lines, and in the problems list.



```
233  #[verify_reclassifier(LiberalAggregate, StrictAggregate)]
234  fn circumvent_get_first_2(v: Vec<u32>) -> u32 {
235      let first: u32 = v.first().unwrap().to_owned();
236
237      let agg_first: u32 = v.iter().fold(init: first, f: |acc: u32, &x: u32|
238          acc + first
239      );
240
241      agg_first
242  }
243
244  #[verify_reclassifier(LiberalAggregate, StrictAggregate)]
245  fn circumvent_get_first_3(v: Vec<u32>) -> u32 {
246      let first: u32 = v.first().unwrap().to_owned();
247
248      let agg_first: u32 = v.iter().fold(init: first, f: |acc: u32, &x: u32|
249          if acc > x {
250              first
251          } else {
252              x
253          }
254      );
255
256      agg_first
257  }
258
```

```
PROBLEMS 14    OUTPUT    DEBUG CONSOLE    TERMINAL

test.rs tests 14
  verify_reclassifier: Not derived from aggregate. rustc [Ln 34, Col 9]
  verify_reclassifier: Output derived from a disallowed function. rustc [Ln 90, Col 11]
  verify_reclassifier: Call to banned function rustc [Ln 90, Col 53]
  verify_reclassifier: File functions not allowed in this function rustc [Ln 143, Col 22]
  verify_reclassifier: Not derived from aggregate. rustc [Ln 191, Col 5]
  verify_reclassifier: This function did not satisfy the strict aggregate policy. rustc [Ln 234, Col 4]
  verify_reclassifier: This invalidates the strict aggregate policy. rustc [Ln 238, Col 9]
  verify_reclassifier: This invalidates the strict aggregate policy. rustc [Ln 238, Col 15]
  verify_reclassifier: This output is not derived from a strict aggregate rustc [Ln 241, Col 5]
  verify_reclassifier: This function did not satisfy the strict aggregate policy. rustc [Ln 245, Col 4]
  verify_reclassifier: This invalidates the strict aggregate policy. rustc [Ln 250, Col 13]
  verify_reclassifier: This invalidates the strict aggregate policy. rustc [Ln 252, Col 13]
  verify_reclassifier: This output is not derived from a strict aggregate rustc [Ln 256, Col 5]
  verify_reclassifier: Output derived from a disallowed function. rustc [Ln 270, Col 21]
```

Figure 6.1: Snapshot from Visual Studio Code with many functions tagged with macro.

6.5 Dynamic Entropy Analysis

As we briefly discussed in Section 3.2.3, aggregates rely on high entropy data to provide safe downgrades of information. There are several problems posed by this statement; first, how do we measure the entropy of a generic sequence, second, how do we enforce this practically; and third, how much overhead is practically possible for enforcing this.

While supporting *any* sequence is not possible, allow for generic analysis can be assisted by the Trait system in Rust. For example, we could require all inputs to implement traits for ordering, comparison and equality to determine the entropy of the sequence.

Enforcing this practically could be challenging. The macro would need to look at the type of all inputs and somehow determine if they implement the required traits. This could be possible by manipulating the function signature to include a trait specification of all variables, but we are not sure if this is possible. The implementation of this check should ideally be as low cost as possible. How much overhead the macro imposes could vary by the size of the inputs, or possibly be configured by the user. We leave the testing and implementation of such a system for future work.

7 Conclusions

To conclude this thesis, we first compare the findings of this thesis with some related work, and then summarize the thesis with concluding remarks.

7.1 Related work

We divide related work into two categories, surveys related to reclassification, and criteria related to reclassification.

7.1.1 Surveys

Sabelfeld and Sands [2] present a survey on declassification, named "Declassification: Dimensions and principles". They provide a road map of the main directions of the current research at the time (2009), but also clarify some overarching concepts. As part of this, they discuss dimensions of information release, which considers what information is released, who releases the information, where in the system information is released, and when information is released.

Our survey differs from theirs by focusing on reclassification in general, which is a broader definition than declassification. We also differ in that our survey poses a set of properties to compare models against, which allows us to create summarized taxonomies of our categories.

Kozyri, Chong, and Myers [3] present a monograph on IFC properties, named "Expressing Information Flow Properties", and a chapter in this provides a survey on reclassification. This is a more up-to-date overview of reclassification, and references the dimensions named in Sabelfeld and Sands [2]. Kozyri, Chong, and Myers [3] categorize the conditions of reclassification into the following subchapters: trusted processes, escape hatches, functions, execution state, interactions. They also discuss robustness and knowledge based models.

Our survey differs from this by organizing different categories, and comparing the models to a common set of properties. One of these properties focus on criteria for when reclassification happens in the model, and we find that this is not discussed as prominently in Kozyri, Chong, and Myers [3].

7.1.2 Reclassification criteria

Chong and Myers [30] present security policies for downgrading and a security type system that incorporates them. They present *declassification policies* which can specify how data should be used prior to declassification, the conditions where declassification is permitted, and how data should be treated after declassification. This is part of the work that lead into Erasure, described in Section 2.2.1.

Our work on criteria differs from this by focusing on function-based reclassification, and looking at a broader scope of criteria. They limit in which contexts declassification is allowed and how data can be handled after declassification, while we try to guarantee that a transformation of data is transformative enough to be used for reclassification.

ANOSY [31] define what they describe as an approximate knowledge synthesizer for qualitative declassification policies. They use refinement types to struct approximations of attacker knowledge. They target boolean queries for multi-integer secrets, which seems to allow them to construct a more enforceable model.

Our work is similar to this, in that we both try to limit what information can be declassified. Our criteria and parser is designed with the Rust language in mind, which has affected several implementation decisions. Their approach also includes attacker knowledge based semantics, which we do not consider.

7.2 Concluding remarks

This thesis has provided an overview of reclassification as an integral part of many IFC models in a reclassification survey. Based on the findings in this survey, we proposed correctness criteria for reclassifiers and a verification framework of these criteria. This was implemented in Rust, using Procedural macros for compile time analysis, and the results of the implementation are shown to be working mostly as intended in the evaluation section.

The comparative analysis in the reclassification survey shows that many existing models rely on reclassification, but not many propose strict requirements on their reclassification functions.

Our work aims to provide a set of composable criteria that can be used to provide guarantees for how reclassifier functions transform data. We chose to target function-based reclassification since we found there to be more flexibility, and subsequently more room for mistakes, when compared to authority-based reclassification. We also chose to target Rust for implementing and experimenting with criteria because it provides flexible compiler hooks and is a prominent language for security because of its innate memory safety.

The designed criteria represent a few of the common patterns we see used for reclassification. Aggregation is commonly used for safe declassification, and functions such as trim or replace are commonly used for endorsement. We find that erasure and deprecation can be supported by our scheme, but since they most likely will be very integrated into the overarching enforcement mechanism, we believe it will not be as relevant for these.

Bibliography

[1] Niklas Broberg and David Sands. “Paralocks: role-based information flow control and beyond.” In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 431–444. DOI: 10.1145/1706299.1706349. URL: http://doi.acm.org/10.1145/1706299.1706349.

[2] Andrei Sabelfeld and David Sands. “Declassification: Dimensions and principles.” In: *Journal of Computer Security* 17.5 (2009), pp. 517–548.

[3] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. “Expressing Information Flow Properties.” In: *Foundations and Trends® in Privacy and Security* 3.1 (2022), pp. 1–102. ISSN: 2474-1558. DOI: 10.1561/3300000008. URL: http://dx.doi.org/10.1561/3300000008.

[4] *How Microsoft Is Adopting Rust*. Aug. 2020. URL: https://medium.com/@tinocaer/how-microsoft-is-adopting-rust-e0f8816566ba (visited on 12/13/2021).

[5] *WHY DISCORD IS SWITCHING FROM GO TO RUST*. Feb. 2020. URL: https://discord.com/blog/why-discord-is-switching-from-go-to-rust (visited on 12/13/2021).

[6] *Chrome: 70% of all security bugs are memory safety issues*. original source: https://www.chromium.org/Home/chromium-security/memory-safety/. May 2020. URL: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/ (visited on 05/26/2022).

[7] *Microsoft: 70 percent of all security bugs are memory safety issues*. Feb. 2019. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (visited on 12/12/2021).

[8] UiT The arctic university. *Cyber Security Group (CSG)*. 2021. URL: https://en.uit.no/forskning/forskningsgrupper/gruppe?p_document_id=343397 (visited on 05/27/2022).

[9] D. E. Comer et al. “Computing as a Discipline.” In: *Commun. ACM* 32.1 (Jan. 1989), 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[10] Stephen Chong and Andrew C. Myers. “End-to-End Enforcement of Erasure and Declassification.” In: *Proceedings of the 21st IEEE Computer*

Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008. IEEE Computer Society, 2008, pp. 98–111. DOI: 10.1109/CSF.2008.12. URL: http://dx.doi.org/10.1109/CSF.2008.12.

[11] Niklas Broberg and David Sands. “Flow locks: Towards a core calculus for dynamic flow policies.” In: *European Symposium on Programming*. Springer. 2006, pp. 180–196.

[12] Aslan Askarov and Stephen Chong. “Learning is change in knowledge: Knowledge-based security for dynamic policies.” In: *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE. 2012, pp. 308–322.

[13] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. “Expressive Declassification Policies and Modular Static Enforcement.” In: *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 2008, pp. 339–353. DOI: 10.1109/SP.2008.20. URL: http://dx.doi.org/10.1109/SP.2008.20.

[14] Andrew C. Myers and Barbara Liskov. “Protecting Privacy Using the Decentralized Label Model.” In: *ACM Trans. Softw. Eng. Methodol.* 9.4 (Oct. 2000), 410–442. ISSN: 1049-331X. DOI: 10.1145/363516.363526. URL: https://doi.org/10.1145/363516.363526.

[15] Andrew C. Myers. “JFlow: Practical Mostly-Static Information Flow Control.” In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, 228–241. ISBN: 1581130953. DOI: 10.1145/292540.292561. URL: https://doi.org/10.1145/292540.292561.

[16] Andrew Clifford Myers. “Mostly-static decentralized information flow control.” PhD thesis. Massachusetts Institute of Technology, 1999.

[17] S. Zdancewic and A.C. Myers. “Robust declassification.” In: *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* 2001, pp. 15–23. DOI: 10.1109/CSFW.2001.930133.

[18] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. “Enforcing robust declassification and qualified robustness.” In: *Journal of Computer Security* 14.2 (2006), pp. 157–196.

[19] Boniface Hicks et al. “Trusted Declassification: High-Level Policy for a Security-Typed Language.” In: *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*. PLAS ’06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, 65–74. ISBN: 1595933743. DOI: 10.1145/1134744.1134757. URL: https://doi.org/10.1145/1134744.1134757.

[20] Peng Li and Steve Zdancewic. “Downgrading Policies and Relaxed Noninterference.” In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, 2005, 158–170. ISBN: 158113830X. DOI: 10.1145/1040305.1040319. URL: https://doi.org/10.1145/1040305.1040319.

[21] Andrei Sabelfeld and Andrew C. Myers. “A Model for Delimited Information Release.” In: *Software Security - Theories and Systems*. Ed. by Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 174–191.

[22] Elisavet Kozyri and Fred B Schneider. “RIF: Reactive information flow labels.” In: *Journal of Computer Security* 28.2 (2020), pp. 191–228.

[23] Scott Olson et al. *Miri Rust Library*. URL: `https://github.com/rust-lang/miri`.

[24] Carl Lerche, Tokio organization, and Rust community. *Loom Rust Library*. URL: `https://github.com/tokio-rs/loom`.

[25] Model-Checking initiative. *Kani Rust Verifier*. URL: `https://github.com/model-checking/kani`.

[26] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.

[27] David Tolnay. *Syn: Parser for Rust source code*. URL: `https://github.com/dtolnay/syn`.

[28] David Tolnay. *Quote: Rust Quasi-Quoting*. URL: `https://github.com/dtolnay/quote`.

[29] Steinar Brenna Hansen. “Information Flow Control using Procedural Macros in Rust.” unpublished. Dec. 2021.

[30] Stephen Chong and Andrew C Myers. “Security policies for downgrading.” In: *Proceedings of the 11th ACM conference on Computer and communications security*. 2004, pp. 198–209.

[31] Sankha Narayan Guria et al. “ANOSY: Approximated Knowledge Synthesis with Refinement Types for Declassification.” In: *arXiv preprint arXiv:2203.12069* (2022).