Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automated Verification of a Rust Differential Privacy Library

Master's Thesis

Till Arnold

March 9, 2024

Advisors: Jonáš Fiala, Anouk Paradis, Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

Differentially private algorithms allow for the computation of statistical results without compromising the privacy of individuals. To accomplish this goal, differential privacy provides a mathematically rigorous definition of privacy which ensures that no data subject can be affected by being included in a dataset. However, implementations of differentially private algorithms that even slightly deviate from this definition can be prone to catastrophic data reconstruction attacks and thus differential privacy libraries, such as the Rust-based OpenDP, are prime candidates for formal verification.

In this thesis we identify important properties of differential privacy transformations, including hyperproperties. In order to be able to prove these properties we implemented and extended the necessary features in Prusti, a Viper-based verifier for Rust. Due to an ongoing rewrite of Prusti this not only involved implementing support for hyperproperties, but also support for foundational Rust language features.

While verifying the identified properties, we were able to show that one of the properties does not hold in the OpenDP library.

Acknowledgments

I want to thank my supervisors Jonáš Fiala and Anouk Paradis for their support and advice. I would also like to thank Aurel Bílý for his technical expertise and his many detailed answers to my many questions about Prusti's architecture and design. I thank Marco Eilers for implementing the *modular product programs* Viper plugin and for the advice on working with it.

Furthermore I would like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on such a highly interesting and in my opinion important project like Prusti and for fostering a positive and supportive culture within the Programming Methodology Group.

Finally I owe a lot of thanks to my friends and family who supported me with kind words, good food and near infinite patience throughout my Master's thesis project.

Contents

Chapter 1

Introduction

Differential privacy (DP) is a rigorous mathematical approach for ensuring that publishing statistics does not compromise the privacy of the individuals who are featured within the dataset. DP strives to apply the same degree of rigorousness to privacy as the field of cryptography applies to secrecy. This is accomplished by introducing noise to the result of the statistical analysis, such that it can be mathematically proven that if one individual in the dataset had answered differently, the statistical distribution of the results would be nearly identical. Intuitively, this ensures the privacy of an individual, since if one person's answers can be completely changed without affecting the statistical results, then it is not possible to learn anything about that person's answers from the resulting statistics.

OpenDP [1] is a open-source DP library, which is implemented in the memory-safe Rust programming language. OpenDP allows users to compose differentially private algorithms out of *transformations*, which are deterministic computations on datasets, and *measurements*, which are randomized functions responsible for adding the random noise as described above. A series of transformations and a measurement can be composed into a new measurement.

For a measurement to add the correct magnitude of noise, that is to say, enough to ensure privacy, but not so much as to render the resulting statistic meaningless, it is important to understand the properties of the dataset passed by the transformations. Intuitively, the output domain of a transformation affects the magnitude of the required noise: The appropriate noise for computing a differentially private sum of N numbers in the domain $[0, 1]$ is trivially insufficient for the same operation on the domain $[10^{99}, 10^{100}]$.

Producing outputs which are not in the expected output domain can compromise the guarantees of a DP system [2]. As such, ensuring the correctness of transformations is one of the requirements for ensuring that the en-

tirety of a differentially private algorithm is correct. OpenDP aims to ensure the correctness of their implementation by including handwritten proofs in LaTeX on the basis of pseudocode alongside their Rust implementation. These proofs, however, are neither automatically verified nor directly linked to the Rust implementation. Furthermore, although OpenDP applies static and dynamic checks and uses Rust's advanced type system to prevent some implementation mistakes, it does not proof the correctness in an automated way. To achieve this verification tools, such as Prusti [3], the Viper-based verifier for Rust, can be used.

In this thesis we identify three central properties of OpenDP transformations, two of which being so-called hyperproperties, and describe how we extended Prusti to allow us to prove these properties. The work in this thesis focuses on transformations and does not verify any properties of measurements, which would involve probabilistic reasoning in Viper and as such exceeds the scope of this thesis.

At the time of this project a rewrite of Prusti was just started with the aim of improving its maintainability. As such, many fundamental features were re-implemented as part of this project, thus this project also serves to test out the new architecture and serves as a basis for future Prusti projects which will be able to utilize these same fundamental features. We refer to the rewritten version simply as Prusti, while we call the previous implementation *old Prusti*.

Chapter 2

Background

2.1 Viper

Viper [4] is a verification infrastructure for simplifying the development of automated verifiers by providing an intermediate verification language, also named Viper, which serves as a translation target for a variety of different programming languages. The Viper language is a simple imperative programming language with built-in support for objects, permissions and contracts.

Viper's built-in types include booleans `Bool`, unbounded mathematical integers `Int`, as well as references `Ref` to objects. Fields of objects can be declared using `field`, as exemplified by `field foo: Int`, which declares a field named `foo` of type `Int`. However, Viper has no built-in support for structs or classes, instead *accessibility predicates* are used to describe the *permission* to access a field on a `Ref`. The accessibility predicate `acc(self.foo)` for example indicates the permission to read and write the field `foo` of an object called `self`. Accessing a field on an object while not having the correct permission causes the verification to fail. Viper permissions are *exclusive*: Only one permission to the same location can be held at the same time. As such, given that both `acc(x.foo)` and `acc(y.foo)` hold, this implies that `x != y`. This allows the verifier to reason that modifying a field of one variable does not affect any other variables, since the exclusive permissions prevent aliasing between references. This approach is related to the separating conjunction in separation logic [5] and Viper can express separation logic [6].

Viper predicates can be used to abstract over assertions and accessibility predicates. For example the `predicate P`, as can be seen below, asserts that both the `foo` and `bar` field are accessible on `self` and the value of `bar` has to be larger than 10. The bodies of predicates are not automatically visible to code calling them, thus to use the fact that `self.bar > 10` when `P(self)`

holds, one needs to explicitly `unfold P(self)`. The `fold` operation can be used to obtain `P(self)` given that all the assertions in the predicate body hold.

```
predicate P(self: Ref) {
    acc(self.foo) && acc(self.bar) && self.bar > 10
}
```

Viper offers two ways to represent computations: *methods* and *functions*, which both can have preconditions, denoted by `requires`, and postconditions denoted by `ensures`. Viper formally verifies that, given the preconditions, the postconditions hold after executing the body of the function or method. We refer to the pre- and postcondition together as the *contract* or *specification* of a method or function.

Viper methods consist of a body which is a sequence of operations such as assigning to local variables, branching or calling other methods or functions. Methods are modular: When calling a method the caller can only rely on what is specified in the contract of the called method, but the implementation is not visible to the caller. The values of arguments can be modified in the body of a method, which necessitates the `old(...)` syntax in order to allow us to refer to a variable's state at the beginning of a method execution. For example a postcondition which guarantees that a field `f` of an argument `A` is not modified is written as `ensures A.f == old(A.f)`.

Viper functions have a body consisting of a single expression and, as opposed to methods, the implementation of a function is visible at the call site and can be used there for reasoning. Viper functions cannot modify arguments and represent pure computations; as such they cannot call methods but are able to call other functions and can be used in specifications, as opposed to methods. A Viper function without a body is an *abstract function*, which informs Viper that it should assume that the specifications of the function hold.

```
domain s_i8 {
  function read(s_i8): Int
  axiom s_i8_bounds {
    forall self: s_i8 :: {read(self)} -128 <= read(self) <= 127
  }
}
```

Viper domains allow the user to define new types with associated functions as well as axioms, which specify the behavior of the functions. The example seen above declares a new type called `s_i8`, which models an 8-bit signed integer, and defines a total function `read` and an axiom `s_i8_bounds` which ensures that `read` always returns a value within a particular range. The expression in curly braces in the `forall` quantifier is a *trigger*, and informs

the underlying SMT solver when to instantiate the quantifier. Incorrect triggers can lead to long verification times or even failure to verify an otherwise correct program.

2.2 Rust

Rust [7] is a systems programming language developed with the goal of eliminating common problems inherent to other systems programming languages, such as undefined behavior, memory-unsafety and data races. Rust's type system ensures this by utilizing the concepts of ownership and borrowing: At any point in time there can exist either one mutable reference or any number of immutable references to the same data, as such there can neither be both immutable and mutable references, or multiple mutable references to the same data at any given point in time. Rust provides a solid type system including *structs*, sum types called *enums*, as well as *traits* which define interfaces similarly to type classes in Haskell.

2.3 Prusti

Rust's language design and type system can be used to detect some programming errors at compile time, however, Rust is not able to guarantee the general functional correctness of programs: Listing 1 shows an incorrectly implemented `clamp` function which will return incorrect results, since the return values for the last two branches have been swapped. To solve this problem Prusti, a verifier for Rust, introduces pre- and postconditions in the form of `#[ensures(...)]` and `#[requires(...)]` attributes. Prusti formally verifies that the conditions hold by translating Rust into Viper code, which is then verified by the Viper infrastructure. For this example function verification expectedly fails, since Viper cannot prove the postcondition.

Prusti leverages Rust's type system to simplify the verification process by mapping Rust's ownership rules to exclusive Viper permissions. Viper-based verifiers for other languages make the use of Viper permissions and

```
1 #[requires(min <= max)]
2 #[ensures(result <= max)]
3 fn clamp(value: i32, min: i32, max: i32) -> i32 {
4   if value < min { min }
5   else if value > max { value }
6   else { max }
7 }
```

Listing 1: Incorrectly implemented Rust function to clamp an integer, annotated with Prusti attributes

predicates explicit while in Prusti this is already expressed by the regular Rust types. As such, Prusti is suitable to be used by programmers who may not be experts in formal verification, since pre- and postconditions are specified using regular Rust syntax. Similarly, this also simplifies the implementation of Prusti since Rust programs translate more naturally to Viper code.

2.4 Differential Privacy

The *Fundamental Law of Information Recovery* states that "overly accurate answers to too many questions will destroy privacy in a spectacular way" [8]. While the precise mathematical reasoning is not in scope for this project, a wide range of research has shown that heuristic solutions attempting to ensure privacy can fail spectacularly and are often vulnerable to reconstruction attacks.

Differential privacy provides a mathematically rigorous definition of privacy by precisely specifying how an attacker would be unable to tell apart the statistical results obtained from neighboring datasets. There are two major definitions for neighboring datasets: In *unbounded* DP two datasets are considered neighboring if one can be obtained from the other by adding or removing a single entry, while in *bounded* DP two datasets are considered neighboring if one can be obtained from the other by changing one entry.

One of the metrics that is used to assess the closeness of two datasets is the *symmetric distance*, which for two multisets A and B is defined as $d_{sym}(A, B) = |A \Delta B|$ where Δ is the *symmetric difference*, which is defined as $A \Delta B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (B \cap A)$. It holds that $d_{sym}(A, B) = \sum |h_A(z) - h_B(z)|$ where $h_x(z)$ is the multiplicity of z within the multiset x.

For a vector V we write $Ms(V)$ for the multiset interpretation of V: $h_{Ms(V)}(e)$ is equal to the number of times the vector V contains the element e. We also define the symmetric distance and difference for vectors as the corresponding function of the multiset interpretation of the vector: For two vectors V and U it holds that $d_{sym}(U, V) = |U \Delta V| = |Ms(U) \Delta Ms(V)|$.

2.5 OpenDP

Rust was chosen as the implementation language for OpenDP due to its performance and reliability, memory- and type-safety and ease to reason about. While the core is implemented in Rust, bindings for Python and R are available.

OpenDP transformations have an associated *stability* function and they keep track of the input- and output domain. OpenDP also stores a *metric* with

```
type Fallible<T> = Result<T, ()>;
struct Function(Rc<dyn Fn(&Vec<i32>) -> Fallible<Vec<i32>>>);
struct StabilityMap(Rc<dyn Fn(u32) -> u32>);
struct AtomDomain { bounds: Option<Bounds> }
struct VectorDomain { element_domain: AtomDomain }

struct Transformation {
  input_domain: VectorDomain, output_domain: VectorDomain,
  function: Function, stability_map: StabilityMap,
}
```

Listing 2: Transformation type in OpenDP

the transformation, which is a notion of closeness of datasets in the input and output domains. OpenDP is designed to be generic enough to serve many different definitions of differential privacy. As such data formats such as dataframes and many different primitive datatypes are supported and OpenDP includes support for a variety of different metrics. For simplicity's sake we consider a simplified OpenDP in which the only datatypes are vectors of 32 bit signed integers and the metric is always the symmetric distance.

Listing 2 shows how we define a transformation in this case. The `input_domain` and `output_domain` keep track of the bounds of the values, while the `function` field contains the actual transformation and `stability_map` contains a function which computes the *stability*. A transformation T is c-stable if for two datasets x and y it holds that:

$$d_{sym}(T(x), T(y)) \leq c \cdot d_{sym}(x, y)$$

This definition is described in terms of a factor c and the `stability_map` in OpenDP allows for nonlinear stability. However in this thesis we constrain ourselves to the linear case.

Listing 3 shows how a clamping transformation is implemented. Many operations such as clamping function on a row-by-row basis, that is in the case of a vector on a element-by-element basis. As such `fn make_clamp` declares that its output domain will be constrained to the given bounds and then passes a closure that clamps an `i32` to the `make_row_by_row_fallible` function. This function in turn constructs a new instance of the `Transformation` type and defines the struct field `function` as a closure that iterates over all the elements of the vector while applying the passed clamping function to each element. `make_row_by_row_fallible` also sets the `stability_map` to be the multiplication with a factor of one, which is equivalent to the identity function, since the clamping transformation is 1-stable.

```rust
fn clamp(value: i32, bounds: &Bounds) -> Fallible<i32> {
  if bounds.min > bounds.max { Err(())}
  else if value < bounds.min { Ok(bounds.min) }
  else if value > bounds.max { Ok(bounds.max) }
  else { Ok(value) }
}

fn make_row_by_row_fallible(
    input_domain: VectorDomain,
    output_row_domain: AtomDomain,
    row_function: impl 'static + Fn(&i32) -> Fallible<i32>
  ) -> Transformation {
    let output_domain = VectorDomain {
     element_domain: output_row_domain
    };
    Transformation {
      input_domain,
      output_domain,
      function: Function(Rc::new(move |arg: &Vec<i32>| {
        arg.iter().map(&row_function).collect()
      })),
      stability_map: StabilityMap(Rc::new(
        move |d_in: u32| d_in*1
      ))
    }
}

fn make_clamp(input_domain: VectorDomain, b: Bounds)
  -> Transformation {
    let output_row_domain = AtomDomain {
      bounds: Some(b.clone())
    };
    make_row_by_row_fallible(
      input_domain,
      output_row_domain,
      move |arg: &i32| { clamp(*arg, &b) }
    )
}
```

Listing 3: Definition of a clamping transformation in OpenDP

Many of the transformations covered in the OpenDP LaTeX proofs share the same postconditions. We have chosen to verify the following properties, due to their common occurrence and central importance in OpenDP:

- *Appropriate output domain*: Applying a transformation to an element of the input domain results in an element of the output domain or alternatively raises a runtime exception. For example after applying a clamping transformation to a vector of integers, all elements contained in it have to be within the specified clamping bounds or a runtime error is raised.
- *Stability guarantee*: Let u and v be any two vectors which have a symmetric distance of d, then the results of applying the transformation to u and v have a symmetric distance of $stability(d)$ or less. For a c-stable transformation T this is equivalent to the symmetric distance $d_{sym}(T(u), T(v))$ having an upper bound of $c \cdot d_{sym}(u, v)$ and as such is exactly equivalent to the definition of being c-stable.
- *Data-independent runtime exceptions*: If a transformation raises a runtime error, that error cannot be dependent on the data passed to the transformation, only on the transformation itself. This ensures that no information about the data is leaked through this side channel.

2.6 Modular Product Programs

The stability guarantee and data-independent runtime exceptions properties are *hyperproperties*, as they do not describe properties of just one execution but relate two executions. For example, the data-independent runtime exception property can be expressed as follows: For two executions of the same function, where all parameters are the same except for the data, both must either return the same error or no error.

Hyperproperties are well-studied in formal verification and one approach for verifying them are *product programs*, which work by constructing a function that combines two or more executions of the original function, and afterwards applying regular verification tools. This approach however suffers from not being modular, and as such does not allow us to reason about each function separately.

The paper *modular product programs* [9] presents a refinement of product programs which allows for modular verification of hyperproperties. The implementation is available for use in Viper in the form of a plugin[1] which provides a `rel` keyword that allows us to refer to a variable in one of two executions. This can be seen in Listing 4 for the example of a Viper method

[1] https://github.com/viperproject/silver-sif-extension

implementing the function $f(x) = x^3$. The `ensures` postcondition encodes the definition of monotonicity ($\forall x \forall y\ x < y \implies f(x) < f(y)$) by asserting that, given that the argument `x` in a first execution (`rel(x,0)`) is less then the argument `x` in a second execution (`rel(x,1)`), then the result of the first execution is less than the result of the second execution (`rel(res, 0) < rel(res, 1)`). Due to this approach, which supports modular verification, a method returning `2 * cubic(x)` also fulfills the monotonicity property by only using the specification of `cubic` without taking its implementation into account.

```
1 method cubic(x: Int) returns (res: Int)
2   ensures rel(x,0) < rel(x,1) ==> rel(res, 0) < rel(res, 1)
3   { res := x * x * x; }
```

Listing 4: Using the `rel` keyword to show that a cubic function is a monotonically increasing function

Chapter 3

Implementation

The goal of this project is to verify the properties of OpenDP as outlined in Chapter 2. To achieve this goal a range of Prusti features, including support for fundamental Rust language features, as well as Prusti-specific functionality, are required. Since Prusti was being rewritten while this project was taking place, these features were implemented as part of this project.

The implementation of these features makes up a major part of this project. While the strategies for encoding types and functions draw inspiration from old Prusti, the code can not simply be copied and adapted from old Prusti due to the significant design improvements made to Prusti. As a result, in most cases the code is written without consulting the source code of old Prusti. Because Prusti is being rewritten as a collaboration between multiple contributors, not everything described in this chapter is implemented by us. Wherever applicable, the sections of this chapter denote which functionalities predate our work and how we extended them.

This chapter describes the implementation work done as part of this project by first presenting a high level description of the architecture of Prusti and then describing the implemented features. By starting with simple examples of primitive types and moving on to more involved Rust features such as structs and enums, as well as mutability and their corresponding encoding in Viper, we construct an approximation of OpenDP's clamping transformation. The chapter concludes with how hyperproperties, such as data-independent runtime exceptions, are proven using Prusti.

3.1 Architecture of Prusti

Prusti and old Prusti share an overall design of using the Rust compiler (`rustc`) as a library. By doing so we avoid parsing Rust code manually and ensure that Prusti is always dealing with valid Rust programs. That is, if

`rustc` rejects a program due to syntax or type errors, or invalid lifetimes, we never run Prusti on it. Prusti works by offering a `rustc`-compatible command-line interface and as such integrates into the regular build system for Rust projects. Prusti translates Rust's *Mid-level Intermediate Representation* (MIR) into Viper code. MIR is a simplified representation of Rust code, in the form of a control-flow graph with all types being explicit.

Prusti features such as `#[requires(...)]`, `#[ensures(...)]`, `forall(...)` and `old(...)` are implemented through a crate called `prusti_contracts`, which exposes these features as regular functions and macros to Rust programs. As such, if a Rust project with Prusti annotations is compiled without Prusti enabled, these annotations have no effect. Since Prusti aims to make formal verification accessible to developers, as opposed to being primarily targeted at formal verification experts, regular Rust syntax is used in Prusti specifications. However, some minimal syntactic extensions are made available in specifications, which are implemented using the standard procedural macro facilities provided by Rust. For example Prusti offers an `===` operator which does not exist in standard Rust; this is detected by Prusti's macros and is rewritten as a function call.[1]

To encode the Viper type representation of a struct, Prusti needs to know the type representation of each field. However, we cannot simply fully recursively encode all the types as that might lead to infinite loops. In old Prusti the mechanisms to avoid this caused code duplication and maintainability issues. Prusti solves this with a pattern called the `TaskEncoder`. The `trait TaskEncoder` is implemented on individual encoders, each responsible for a small task, such as an encoder for constants or an encoder for types. Each of the `TaskEncoder` implementations has three main associated types:

- `TaskKey`: A complete description of the task to be performed. For the type encoder this would be the MIR representation of that type.
- `OutputFull`: The result of the encoding. For the above encoder this is the resulting Viper domain including all functions and axioms.
- `OutputRef`: A partial result of the encoding that suffices to refer to it. In the case of the type encoder this is the name of the resulting Viper domain.

In addition to these types each `TaskEncoder` implements a method `do_encode_full` which has a parameter of type `TaskKey` and produces an `OutputFull`. The implementations of `do_encode_full` are required to call the `emit_output_ref` method to produce an `OutputRef`. There is no explicit separation between generating the `OutputRef` and the `OutputFull`, both are created while the encoders are recursively calling each other. However, we avoid

[1] This snapshot equality operator will be explained in Section 3.9.

infinite loops, as long as `emit_output_ref` is always called by an encoder before its output is needed in the recursion.

The `TaskEncoders` use a thread-local cache, with the Viper expressions being stored in an allocation arena[2]. Note that the `TaskEncoder` as presented here is simplified, in reality it also handles encoding errors and the function signatures contain lifetimes dependent on the allocation arena. The pattern was not designed by us as part of the project, but is utilized in the code written by us.

Once all `TaskEncoders` have finished executing, Prusti produces the textual representation of Viper code and the Viper command-line tools are then manually ran on these text files. This is only a temporary solution and the aim is to eventually use the *Java Native Interface* (JNI) to directly create the Viper expressions as Java objects in memory, as is already the case in old Prusti.

The following sections describe how Rust types, expressions and functions are translated into Viper. In broad terms we define two encodings: The *snapshot* encoding for mathematically pure computations and immutable values, which uses Viper domains and functions, and the *predicate* encoding for mutable values, which uses Viper predicates, permissions and methods. For each Rust type $\mathcal{T}$ we define a *snapshot type* $s\_\mathcal{T}$ and a *predicate type* $p\_\mathcal{T}$. As mentioned in the previous section, Prusti operates on MIR and not directly on the Rust source code. The examples in this section however are shown as Rust source code, since this is easier to understand and the concepts apply in the same way.

3.2 Encoding Primitive Type Snapshots

As described before, the OpenDP library uses transformations to deterministically transform data. A simple but realistic transformation is the one which clamps all integers in a vector between two integers. This will be our running example. As such we first define a function to clamp a single integer, as shown in Listing 5, annotated with a Prusti contract.[3]

In order to translate any Rust function to Viper we need to encode all the types used by the function. Since all the arguments as well as the return value of the `clamp` function are of the type `i32`, it suffices to encode that type. Fundamentally, primitive types such as integers and booleans are modeled utilizing their corresponding Viper type. Note however that we cannot directly use a Viper `Int` to model a Rust `i32`, since the Viper `Int` is

[2]Using the `bumpalo` crate https://github.com/fitzgen/bumpalo

[3]The postcondition `min` $\leq$ `result` would also hold but it was left out for simplicity's sake.

```
#[pure] #[requires(min <= max)] #[ensures(result <= max)]
fn clamp(value: i32, min: i32, max: i32) -> i32 {
  if value < min { min }
  else if value > max { max }
  else { value }
}
```

Listing 5: Rust function to clamp a 32-bit integer annotated with Prusti attributes

```
domain s_i32 {
  function s_i32_cons(Int): s_i32
  function s_i32_read(s_i32): Int

  axiom ax_i32_cons_read {
    forall val: Int :: {s_i32_cons(val)}
        s_i32_read(s_i32_cons(val)) == val
  }
  axiom ax_i32_cons {
    forall self: s_i32 :: {s_i32_read(self)}
        s_i32_cons(s_i32_read(self)) == self
  }
  axiom s_i32_bounds {
    forall self: s_i32 :: {s_i32_read(self)}
        -2147483648 <= s_i32_read(self) <= 2147483647
  }
}
```

Listing 6: Snapshot encoding of an `i32`

```
function f_clamp(value: s_i32, min: s_i32, max: s_i32): s_i32
  requires s_i32_read(min) <= s_i32_read(max)
  ensures s_i32_read(result) <= s_i32_read(max)
{
  s_i32_read(value) < s_i32_read(min) ? min
: s_i32_read(value) > s_i32_read(max) ? max : value
}
```

Listing 7: Viper encoding of the `clamp` function from Listing 5

a mathematical, and thus unbounded integer, while Rust integer types are constrained by their fixed size. We instead define a snapshot type using Viper domains: For each primitive Rust type $\mathcal{T}$ we define a Viper `domain` called $s\_\mathcal{T}$. In this domain we define a *constructor function* named `s_`$\mathcal{T}$`_cons` which, given an argument of the primitive Viper type which corresponds to $\mathcal{T}$, returns a value of the snapshot type $s\_\mathcal{T}$. We also define a function `s_`$\mathcal{T}$`_read` which is the inverse of `s_`$\mathcal{T}$`_cons`. Furthermore we define axioms to ensure that these functions behave as intended. Values of snapshot types are intended to represent Rust values at a specific point in time and as such do not support in-place mutation.

An example of a primitive type encoded as a snapshot can be seen in Listing 6, which shows the constructor function `s_i32_cons`, that, given an `Int`, constructs an `s_i32`, while `s_i32_read` produces an `Int` when given an `s_i32`. The axioms `ax_i32_cons_read` and `ax_i32_cons` ensure that these two functions are in fact inverses of each other. Finally, the axiom `s_i32_bounds` encodes that an `i32` has a bounded size.

With the encoding of `i32` completed, we are now able to encode the `clamp` function as shown in Listing 7. The Viper function's signature mirrors the one of the Rust function: Where the Rust parameter and return types are `i32`, the Viper function in turn utilizes `s_i32` snapshot types. The `if`, `else if`, `else` expression in Rust is encoded as a pair of Viper ternary operators. To perform the comparisons in the if conditions the `s_i32_read` function is used to convert the snapshot type into a Viper `Int`, since there is no concept of performing order comparisons on values of the type `s_i32`, because `s_i32` is a Viper domain. The precondition and postcondition of the Rust `clamp` function were encoded as Viper `requires` and `ensures`. Note how the conditions are encoded in the same fashion as the body of the function.

The snapshot encoding is used for this function since it is annotated with the Prusti `#[pure]` attribute. When our work on the Prusti code base started, the snapshot encoding of primitives and the fundamental functionality to translate Rust code into Viper expressions were already implemented and used to encode contracts. We implemented an encoder for `#[pure]` functions which builds upon this existing implementation. Furthermore, we extended the implementation with functionality needed in function bodies such as support for comparison operators, and by improving the generation of the ternary chains from `if else` expressions. In addition, we also contributed bug fixes to the thread-local cache, to the mechanism for querying `rustc` for the bodies of functions, and to allow functions without pre- or postconditions.

3.3 Encoding Struct Snapshots

One of the main composite data types in Rust is the `struct`. We can adapt our `fn clamp` example from Listing 5 to take a `struct Bounds` instead of two separate `i32` arguments for minimum and maximum value as seen in Listing 8. For this we need to encode structs in Prusti as can be seen in Listing 9. This encoding is similar in structure to the one of primitive types. A domain `s_Bounds` is created for the struct with a constructor function which takes one parameter for each field. We also define a *snapshot field function* for each field, that, given a parameter of type `s_Bounds`, returns a snapshot of the corresponding field. Axioms ensure that the constructor and snapshot field functions behave as expected.

Now that we have encoded the type, we can discuss the encoding of the actual function, as can be seen in Listing 10. The structure is the same as in the primitive case, using two ternary operators, primarily differing by the function arguments: Instead of taking two arguments `min` and `max` of type `s_i32`, the function takes one argument `bounds` of type `s_Bounds` and retrieves the values of the fields `min` and `max` by using the function calls `s_Bounds_field_min(bounds)` and `s_Bounds_field_max(bounds)` respectively. The `s_i32_read` function is still called on the result of the snapshot field functions since the function calls return `s_i32`.

The encoding of structs as snapshots was already implemented when we started our work on the project. We implemented support for expressions using structs such as constructing them and accessing fields. We also needed to change some of the triggers of the axioms to allow verification to complete.

```
1  struct Bounds { min: i32, max: i32 }
2
3  #[pure]
4  fn clamp(value: i32, bounds: Bounds) -> i32 {
5    if value < bounds.min { bounds.min }
6    else if value > bounds.max { bounds.max }
7    else { value }
8  }
```

Listing 8: Rust function to clamp a 32-bit integer using a `struct` to pass the bounds

```
domain s_Bounds {
  axiom ax_s_Bounds_cons_field_min {
    forall f0: s_i32, f1: s_i32 :: {s_Bounds_cons(f0, f1)}
      s_Bounds_field_min(s_Bounds_cons(f0, f1)) == f0
  }
  axiom ax_s_Bounds_cons_field_max {
    forall f0: s_i32, f1: s_i32 :: {s_Bounds_cons(f0, f1)}
      s_Bounds_field_max(s_Bounds_cons(f0, f1)) == f1
  }
  axiom ax_s_Bounds_cons {
    forall self: s_Bounds ::
    {s_Bounds_cons(s_Bounds_field_min(self),
    ↪ s_Bounds_field_max(self))}
    s_Bounds_cons(s_Bounds_field_min(self),
    ↪ s_Bounds_field_max(self)) == self
  }

  function s_Bounds_cons(s_i32, s_i32): s_Bounds
  function s_Bounds_field_min(s_Bounds): s_i32
  function s_Bounds_field_max(s_Bounds): s_i32
}
```

Listing 9: Snapshot encoding of the struct `Bounds` from Listing 8

```
function f_clamp(value: s_i32, bounds: s_Bounds): s_i32 {
  s_i32_read(value) < s_i32_read(s_Bounds_field_min(bounds))
? s_Bounds_field_min(bounds)
: s_i32_read(value) > s_i32_read(s_Bounds_field_max(bounds))
? s_Bounds_field_max(bounds)
: value
}
```

Listing 10: The Viper encoding of the function seen in Listing 8

3.4 Encoding Enum Snapshots

In Rust exceptional control flow is not modeled by using a language construct such as exceptions, but instead by using a `Result<T, E>` enum. Enums are disjoint union types (or sum types) and as such each variant has an associated integer *discriminant* which is used at runtime to distinguish values of different variants. An idiomatic approach to handling the case where the `Bounds` are invalid, specifically where the minimum is larger than the maximum, can be seen in Listing 11, which shows the `clamp` function extended with a check for the validity of the bounds. The listing also defines a postcondition which, using a match expression, ensures that if the result is

```
1  enum Result { Ok(i32), Err }
2
3  #[pure]
4  #[ensures(match result {
5    Result::Err => true,
6    Result::Ok(value) => value <= bounds.max
7  })]
8  fn clamp(value: i32, bounds: Bounds) -> Result {
9    if bounds.min > bounds.max { Result::Err }
10   else if value < bounds.min { Result::Ok(bounds.min) }
11   else if value > bounds.max { Result::Ok(bounds.max) }
12   else { Result::Ok(value) }
13 }
```

Listing 11: Clamping function returning an enum

`Ok(value)`, then the condition `value <= bounds.max` holds, whereas in the case that the result is an `Err` this trivially cannot hold as there is no value. Usually the generic `Result<T, E>` from the Rust standard library would be used, but due to limitations in Prusti this is not yet supported and we instead define a Result type specific to this use case.

Listing 12 shows how this enum is encoded in Viper. We define a function `s_Result_discr` to get the discriminant of an enum instance, as well as a constructor function for each of the variants. An axiom ensures that the discriminant is always valid, while for each variant an axiom ensures that the discriminant of the constructor is as expected. For the variant with a field, the snapshot field function `s_Result_Ok_read_0` (Line 18) is defined similarly to how it is defined for structs, consequently, if the enum variant had more fields or other variants had fields, then more such functions would be defined. Additional axioms are defined to establish the relation between the constructor and the snapshot field function.

The body and the contract of the function are then encoded in the same way as in the struct case, while match expressions are encoded as ternary chains using comparisons of the discriminant.[4] Listing 13 shows this on the example of the postcondition. A function called `p_Bool_unreachable` is used to model the fact that the last branch of the ternary chain is not reachable. `p_Bool_unreachable` is simply a nullary function returning an `s_bool` with pre- and postconditions of false. Such a function is generated for all snapshot types.

Explicit discriminants Note how the `s_Result_discr` function returns an `s_isize` instead of a Viper `Int`. This is the case since accessing the discrimi-

[4]This is the case because conditional jumps in MIR are translated to such ternary chains.

```
domain s_Result {
  function s_Result_discr(s_Result): s_isize
  function s_Result_Ok_cons(s_Int_i32): s_Result
  function s_Result_Err_cons(): s_Result

  axiom s_Result_discr_bounds {
    forall self: s_Result ::
    ↪ {s_isize_read(s_Result_discr(self))}
    0 <= s_isize_read(s_Result_discr(self)) &&
    ↪ s_isize_read(s_Result_discr(self)) <= 1
  }
  axiom ax_s_Result_Ok_cons_discr {
    forall f0: s_i32 :: {s_Result_discr(s_Result_Ok_cons(f0))}
      s_Result_discr(s_Result_Ok_cons(f0)) == s_isize_cons(0)
  }
  axiom ax_s_Result_Err_cons_discr {
    s_Result_discr(s_Result_Err_cons()) == s_isize_cons(1)
  }

  function s_Result_Ok_read_0(s_Result): s_i32
  axiom ax_s_Result_Ok_cons_read_0 {
    forall f0: s_i32 :: {s_Result_Ok_cons(f0)}
      s_Result_Ok_read_0(s_Result_Ok_cons(f0)) == f0
  }
  axiom ax_s_Result_Ok_cons {
    forall self: s_Result ::
    ↪ {s_Result_Ok_cons(s_Result_Ok_field_0(self))}
      s_Result_Ok_cons(s_Result_Ok_field_0(self)) == self
  }

}
```

Listing 12: Snapshot encoding of the `Result` enum as seen in Listing 11

```
function f_clamp(value: s_i32, bounds: s_Bounds): s_Result
  ensures s_Bool_read(
  s_isize_read(s_Result_discr(result)) == 1
  ? s_Bool_cons(true)
  : (s_isize_read((s_Result_discr(result)))) == 0
  ?  s_Bool_cons(s_i32_read(s_Result_Ok_read_0(result)) <=
  ↪  s_i32_read(s_Bounds_field_max(bounds)))
  : p_Bool_unreachable()
  ) { /* ... */ }
```

Listing 13: Viper encoding of the clamping function returning an enum as seen in Listing 11

```
#[repr(i8)]
enum Color { Red = 42, /* ... */}
```

```
domain s_Color {
  axiom ax_s_Color_Red_cons_discr {
    s_Color_discr(s_Color_Red_cons()) == s_i8_cons(42)
  }
  // ... `cons_discr` axioms for other colors
  axiom s_Color_discr_values {
    forall self: s_Color :: {s_i8_read(s_Color_discr(self))}
    s_i8_read(s_Color_discr(self)) == 42 ||
    // other discriminants
  }
  function s_Color_discr(s_Color): s_i8
  function s_Color_Red_cons(): s_Color
  // other constructors ...
}
```

Listing 14: Rust enum with an explicitly set discriminant and the corresponding Viper snapshot encoding

nant is done directly in the MIR code and as such the discriminant needs to be an actual Rust type. This is exemplified further in Listing 14 where the discriminant is explicitly set to the value `42` and the type of the discriminant is manually set to an `i8`.

We implemented enum support in Prusti in its entirety as part of this project: This includes the snapshot type encoding, enum usage in `#[pure]` functions and specifications, as well as refactoring the encoder for structs to allow for code reuse.

3.5 Encoding Mutable Primitive Types

The functions we use as examples so far are labeled as `#[pure]`. This allows us to use them in contracts, however `#[pure]` functions are not able to mutate data and the entire function has to be able to be encoded as a single Viper expression to be able to become a Viper function. This restricts which language features are usable in `#[pure]` functions.

All Rust functions not labeled as `#[pure]` are encoded as Viper methods. We need Viper methods if we want to be able to support Rust language features such as loops, or if we mutate function arguments. For this we need a second representation of types called the *predicate encoding*, as snapshots types do not support in-place mutation.

A primitive type $\mathcal{T}$ is encoded by defining a field `f_`$\mathcal{T}$ of type `s_`$\mathcal{T}$ and a predicate `p_`$\mathcal{T}$ which allows access to the Viper field. Additionally, we define a *snapshot function* `p_`$\mathcal{T}$`_snap` to convert a `p_`$\mathcal{T}$ into an `s_`$\mathcal{T}$, which is implemented directly by accessing the field. We define an abstract method `assign_p_`$\mathcal{T}$ to convert a value of a snapshot type into its corresponding value of the predicate type and store it in a `Ref`: The method takes a parameter `self` of type `Ref` as well as an `s_`$\mathcal{T}$ and has postconditions ensuring that after the method call `p_`$\mathcal{T}$`(self)` holds and that if `self` is converted back to a snapshot using the snapshot function, it is equivalent to the second value passed to the `assign_p_`$\mathcal{T}$ function. This can be seen in Listing 15 on the example of an `i32`.

```
1  field f_i32: s_i32
2  predicate p_i32(self: Ref) { acc(self.f_i32) }
3  function p_i32_snap(self: Ref): s_i32 requires p_i32(self)
4  { unfolding p_i32(self) in self.f_i32 }
5
6  method assign_p_i32(self: Ref, self_new: s_i32)
7    ensures p_i32(self) && p_i32_snap(self) == self_new
```

Listing 15: Predicate encoding of an `i32`

Listing 16 shows how our original primitive clamping function from Listing 5 is encoded with the predicate-based approach using a Viper method. Note that instead of returning a value the first argument of the method is a `Ref` which is used to pass back the result as can be seen on line 10. The MIR basic blocks are converted to Viper labels and gotos are used to jump to them. The user-provided pre- and postconditions are encoded by first converting the arguments to their snapshot representation using their corresponding snapshot functions and then encoding the conditions analogously to `#[pure]` functions. Additionally, the methods also include specifications that encode the type of the `Ref` arguments and return values using predicates. Since the predicate-based encoding allows for mutation, the postconditions need to use `old(...)` correctly when referring to arguments, as omitting the `old(...)` calls causes them to refer to the arguments in their state after the execution of the method.

The encoding described here was already implemented before this thesis started, however, for the clamping example to be successfully translated into Viper, our previously described additions to the snapshot encoding of primitive types, such as the implementation of comparison operators, are required. Since it was not implemented as part of this project, we forego a detailed explanation of the mechanism for encoding method bodies and instead present an abstract overview of the process: The basic blocks of the Rust MIR are each converted to a label in Viper, while the *free place capability summary analysis* (freepcs) is used to determine where to insert fold, unfold and exhale statements.[5] Overall this process is guided by and made simpler by Rust's ownership rules.

Functions labeled `#[pure]` get Viper method encoding in addition to the previously described function encoding. This allows for future features such as verifying unsafe code and can be used to detect bugs in `rustc`: If the Rust compiler erroneously allows code which violates the ownership rules, Viper could reject that code, as it would lead to unsatisfied accessibility predicates.

3.6 Encoding Mutable Structs

Listing 17 shows how a struct is encoded as a predicate in Viper. For each field of the struct we create a *predicate field function*, which is an abstract Viper function with parameter and return type `Ref`. These predicate field functions model accessing a field in the struct by taking a reference to the struct and returning a reference to the field. We use abstract functions instead of Viper fields to model struct fields because this resembles the real memory representation of a Rust struct more closely: Unless explicitly de-

[5] The `exhale` statements where omitted from the listings as they do not aid in understanding.

```
method m_clamp(return_value: Ref, value: Ref, min: Ref, max:
↪  Ref)
  requires p_i32(value) && p_i32(min) && p_i32(max)
  requires s_i32_read(p_i32_snap(min)) <=
  ↪  s_i32_read(p_i32_snap(max))
  ensures p_i32(return_value)
  ensures s_i32_read(p_i32_snap(return_value)) <=
  ↪  s_i32_read(old(p_i32_snap(max))) {
  // ... declare temporary variables

  assign_p_Bool(cond_tmp,
  ↪  s_Bool_cons(s_i32_read(p_i32_snap(value_tmp)) <
  ↪  s_i32_read(p_i32_snap(min_tmp))))
  if (s_Bool_read(p_Bool_snap(cond_tmp)) == false){goto bb_1}
  assign_p_i32(return_value, p_i32_snap(min))
  goto end

label bb_1
  assign_p_Bool(cond_tmp,
  ↪  s_Bool_cons(s_i32_read(p_i32_snap(value_tmp)) >
  ↪  s_i32_read(p_i32_snap(max_tmp))))
  if (s_Bool_read(p_Bool_snap(cond_tmp)) == false) { goto bb_2
  ↪  }
  assign_p_i32(return_value, p_i32_snap(max))
  goto end

label bb_2
  assign_p_i32(return_value, p_i32_snap(value))

label end
}
```

Listing 16: Predicate encoding of the primitive clamp function from Listing 5

```
function p_Bounds_field_min(self: Ref): Ref
function p_Bounds_field_max(self: Ref): Ref

predicate p_Bounds(self: Ref) {
  p_i32(p_Bounds_field_min(self)) &&
  ↪ p_i32(p_Bounds_field_max(self))
}

function p_Bounds_snap(self: Ref): s_Bounds
requires p_Bounds(self)
{ unfolding p_Bounds(self) in
↪ (s_Bounds_cons(p_i32_snap(p_Bounds_field_min(self)),
↪ p_i32_snap(p_Bounds_field_max(self)))) }

method assign_p_Bounds(self: Ref, self_new: s_Bounds)
  ensures p_Bounds(self) && p_Bounds_snap(self) == self_new
```

Listing 17: Predicate encoding of the `Bounds` struct from Listing 5

clared, the fields of a struct in Rust are not references, but are directly laid out next to each other in memory as part of the struct. As such the predicate field functions represent offsets into the struct. We then use the predicate field functions to define the predicate `p_Bounds` for the struct. The body of the predicate consists of the conjunction of the predicates of the field types applied to the predicate field functions. In other words, `p_Bounds(x)` holds if all the predicate field functions of `Bounds` applied to `x` satisfy the predicate of the type of that field. As in the primitive case we define a snapshot function to convert from the predicate encoding to the snapshot encoding, which for structs is implemented by creating a snapshot of each field and applying the snapshot constructor of the struct to them. The `assign` method operates identically to the primitive case, ensuring that the `Ref` passed can be converted to a snapshot correctly.

Listing 18 shows how the clamp function from Listing 8, which takes a struct as a parameter, is translated to a Viper method. This is analogous to the primitive case with the exception that it uses the predicate field functions. Also note that `unfold` statements are needed to access the assertions specified by the body of the predicate. The `fold` and `unfold` statements are generated based on the freepcs analysis and were not implemented as part of this project.

The predicate encoding of structs already existed in Prusti when this thesis started, however we built upon this for the predicate encoding of enums.

```
method m_clamp(return_value: Ref, value: Ref, bounds: Ref)
  requires p_i32(value) requires p_Bounds(bounds)
  ensures p_i32(return_value) {
  // ... variable declarations
  unfold p_Bounds(bounds)
  assign_p_i32(min, p_i32_snap(p_Bounds_field_min(bounds)))
  assign_p_Bool(cond_tmp, s_Bool_cons((s_i32_read_0(
  ↪  p_i32_snap(value_tmp))) <
  ↪  (s_i32_read_0(p_i32_snap(min)))))
  if (s_Bool_read_0(p_Bool_snap(cond_tmp)) == false){goto bb_1}
  assign_p_i32(return_value,
  ↪  p_i32_snap(p_Bounds_field_min(bounds)))
  //...
  fold p_Bounds(bounds)
}
```

Listing 18: Predicate encoding of the clamp function which takes a struct as an argument from Listing 8

3.7 Encoding Mutable Enums

Listing 19 shows how the `Result` enum from the enum example is encoded as a predicate. The discriminant is stored as a Viper `field` `p_Result_discr`. A predicate for each variant is generated (`p_Result_Ok`, `p_Result_Err`) which behaves similarly as if the variant was a struct, as such predicate field functions are generated for the fields of the enum variants as can be seen with `p_Result_Ok_field_0`. A predicate for the entire enum `p_Result` is generated which bounds the value of the discriminant and includes a ternary chain which calls the predicate for the variants depending on the value of the discriminant.

The snapshot function `p_Result_snap` unfolds the predicate to access the discriminant `field` and gain access to the inner predicates, which then are unfolded depending on the value of the discriminant and, as with structs, snapshots of each fields are created and the snapshot constructor of the enum is called with the field snapshots as arguments. Finally `assign_p_Result` is defined exactly the same as for structs.

The clamping function is translated to a Viper method analogously to the struct case, differing only in returning an `Err` by `assign_p_Result(return_value, s_Result_Err_cons())` and wrapping each of the return values in `Ok` such as `assign_p_Result(return_value, s_Result_Ok_cons(p_i32_snap(...))`.

The predicate encoding of enums was implemented in its entirety as part of this project, while reusing parts of the predicate encoder for structs.

```
field p_Result_discr: s_isize
function p_Result_Ok_field_0(self: Ref): Ref
predicate p_Result_Ok(self: Ref)
↪  {p_i32(p_Result_Ok_field_0(self))}
predicate p_Result_Err(self: Ref) {true}

predicate p_Result(self: Ref) {
  (acc(self.p_Result_discr) && 0 <=
  ↪  s_isize_read(self.p_Result_discr) &&
  ↪  s_isize_read(self.p_Result_discr) <= 1)
  && (
    s_isize_read(self.p_Result_discr) == 1 ? p_Result_Err(self)
    : s_isize_read(self.p_Result_discr) == 0 ? p_Result_Ok(self)
    : false
)}

function p_Result_snap(self: Ref): s_Result
  requires p_Result(self) {
  unfolding p_Result(self) in s_isize_read(self.p_Result_discr)
  ↪  == 0
? unfolding p_Result_Ok(self) in
↪  s_Result_Ok_cons(p_i32_snap(p_Result_Ok_field_0(self)))
: unfolding p_Result_Err(self) in s_Result_Err_cons()
}
```

Listing 19: Predicate encoding of the `Result` enum

3.8 Encoding Function Calls

We have described how both *#[pure]* and non-pure functions are encoded. Calling a function from another function is directly implemented by a corresponding Viper call. As described before for *#[pure]* functions we generate both a Viper method and a Viper function. When calling a function in a pure context, that is in a specification or in the body of a *#[pure]* function, we have to call the Viper function as it is not possible to call Viper methods in these contexts. Even when doing a call in an impure context we prefer to call the Viper function and only call the method if the called function is not *#[pure]* and thus has no Viper function. We prefer to use the Viper function since Viper can take the body of a `function` into account at the call site, whereas with a `method` only the contract can be used. Listing 20 shows how the call to a function in a Viper method is encoded depending on whether the called function is *#[pure]* or not.

```
fn client(i: i32) -> i32 { clamp(i, Bounds{min: 10, max: 20}) }
```

```
  assign_p_i32(tmp_i, p_i32_snap(i))
  assign_p_Bounds(tmp_bounds, s_Bounds_cons(s_i32_cons(10),
  ↪ s_i32_cons(20)))

// If the clamp function is not #[pure]
  m_clamp(return_value, tmp_i, tmp_bounds)

// If the clamp function is #[pure]
  assign_p_i32(return_value, f_clamp(p_i32_snap(temp_i),
  ↪ p_Bounds_snap(tmp_bounds)))
```

Listing 20: Function call encoding depending on whether the called function is `#[pure]`.

3.9 Snapshot Equality

In Rust the `==` and `!=` operators are available for values of types which implement the `PartialEq` trait. Since it is desirable to be able to express structural equality between values in Prusti specifications, even if the operands do not implement `PartialEq`, Prusti introduces a *snapshot equality* operator (`===`) which uses snapshot types to determine equality. The snapshot equality operator is simply implemented by converting the two operands into their snapshot representation, if they are not already, and then using the Viper `==` operator which is defined for types defined with a Viper domain.

Note that the macro preprocessing to implement the syntax for `===` was already implemented when our work on this project started. We did however implement the translation of the operator to Viper.

3.10 Trusted Functions

While we were able to implement a significant number of features, there are still many features that Prusti does not yet support. Even once all language features are implemented in Prusti, there will still be correct functions which we will not be able to verify due to their complexity, the use of `unsafe` or the use of unstable language features. It is also desirable to be able to skip verification for specific parts of a code base, which allows a user to focus on particularly critical functions, such as central algorithms, while ignoring less interesting parts of a program, for example code implementing a GUI.

For this case Prusti introduces the concept of *`#[trusted]`* functions. When a function is annotated as *`#[trusted]`* Prusti will not check that the post-

```
pub struct Vector { len: usize, contents: i32 }

impl Vector {
    #[pure] #[trusted]
    #[requires(idx < self.len)]
    fn get(self, idx: usize) -> i32 {/* ... */}

    #[trusted]
    #[requires(idx < self.len)]
    #[ensures(result.0 == self.get(idx))]
    #[ensures(result.1 === self)]
    fn impure_get(self, idx: usize) -> (i32, Self) {/* ... */}

    #[trusted]
    #[requires(idx < self.len)]
    #[ensures(self.len == result.len)]
    #[ensures(result.get(idx) == value)]
    #[ensures(forall(|i : usize| (i < self.len && i != idx)
        ==> result.get(i) == self.get(i)))]
    fn set(self, idx: usize, value: i32) -> Self {/* ... */}
}
```

Listing 21: Implementation of a vector datatype using trusted functions

conditions of that function can be proven and will instead assume that they hold.

We can use this for example to simulate having a `Vector` type, as can be seen in Listing 21, despite Prusti not yet supporting arrays or the `Vec<T>` type from the Rust standard library. We define a struct `Vector` as well as a `get` method that, given an index of type `usize`, returns an element of type `i32`. For this method we define only a precondition asserting that the `idx` has to be within bounds. Usually, a function such as `get` would take a shared reference `&self` instead of taking ownership of `self`, but shared references are not yet supported in Prusti. With this signature `get` can only be called once as it consumes the `Vector` argument. To work around this we also define a method `impure_get` which has postconditions encoding that the method will return the `Vector` unchanged as well as the result of calling `get`. Both of these `get` methods are annotated with `#[trusted]` and as such their implementations are ignored.

We also define a `set` method with postconditions such that the `Vector` returned is the same at all indexes except at the index where the new value was inserted. With this we have created an axiomatized equivalent of a `Vec<i32>`. Note how the Vector struct has an unused `i32` field called

`contents`, which is necessary as otherwise for any two vectors `u` and `v` with `u.len == v.len` it would hold that `u === v` by the definition of snapshot equality, since all the snapshot fields are identical. As such, all vectors of the same length would have the same contents. By giving the `Vector` an additional field which is never mentioned in any of the specifications we avoid this issue. Intuitively this field models the actual contents of the vector.

`#[trusted]` is implemented by not emitting a body for the function, resulting in an abstract Viper function. It is important to note that *`#[trusted]`* functions can cause unsoundness, for example if a function is annotated with *`#[trusted]`* and *`#[ensures(false)]`* this function can be used to prove anything. The syntax for *`#[trusted]`* attributes was already implemented when this work started, detecting *`#[trusted]`* functions in Prusti and emitting an abstract Viper function was implemented as part of this project.

3.11 Verifying Hyperproperties

As described before, two of the properties we want to show are hyperproperties. As such we cannot directly show them using regular Viper. However, we can utilize the Viper plugin from the modular product programs paper as described in the background chapter. The plugin provides a `rel` keyword which can be used as `rel(expr,0)` or `rel(expr,1)` to refer to an expression `expr` in one of two executions. We implemented syntax in Prusti as `rel0(expr)` and `rel1(expr)` which correspond to the plugin syntax. Note that while these are regular functions in Rust syntax we cannot actually treat them as regular functions: If `rel1` was a regular function, then an expression such as `rel1(f(expr))` would be equivalent to `let x = f(expr); rel1(x)` which in turn in Viper would then become `let x = f(expr) in rel(x,1)`. This is however semantically different from `rel(f(expr),1)` as this does not refer to the `expr` from execution 1 and instead is equivalent to just `f(expr)`. As such we need preprocessing to mitigate this: An expression such as `rel1(expr)` is translated to the expression { `rel1_start(); let r = expr; rel1_end(); r` }. The *marker functions* `rel0_start` and `rel0_end` have no effect at runtime, but Prusti detects the occurrences of these marker functions in its macros and changes its behavior to wrap all function parameters and return values in the corresponding `rel` call, which for `rel1(f(arg1, arg2))` where `arg1` and `arg2` are function parameters, results in Viper code equivalent to `f(rel(arg1,1), rel(arg2,1))`. This works in particular since the arguments at this point are snapshots. Corresponding `rel0` marker functions are used in the case when `rel0` is used.

Listing 22 shows an example use case for `rel`, which verifies successfully. If any of the `Ok` branches were changed to `Err`, verification would fail. The

```
#[ensures(rel0(&bounds) === rel1(&bounds) ==>
match (rel0(&result), rel1(&result)) {
  (Result::Err,Result::Err) => true,
  (Result::Ok(_),Result::Ok(_)) => true,
  _ => false,
})]
fn clamp(value: i32, bounds: Bounds) -> Result {
  if bounds.min > bounds.max { Result::Err }
  else if value < bounds.min { Result::Ok(bounds.min) }
  else if value > bounds.max { Result::Ok(bounds.max) }
  else { Result::Ok(value) }
}
```

Listing 22: Prusti postcondition showing that the `clamp` function has data-independent errors using the `rel0` and `rel1` syntax

postcondition encodes that if the two bounds are the same, that is the function calls only differ in their `value` parameters, then the result has to be either an `Err` in both cases or in neither: The errors of the function are independent from the `value`.

The same preprocessing also is also required for `old` as there too `old(f(foo))` is not equivalent to `let x = f(foo) in old(x)`.

3.12 Optimizer

To improve readability of the generated Viper code and potentially improve performance, we implemented some basic optimizations including simple boolean simplifications. In addition to this, for variables v_1 and v_2 and an expression e, we transform expressions of the shape `let` v_1 `=` e `in` v_1 into e, as well as `let` v_1 `=` v_2 `in e` into $e[v_1 \mapsto v_2]$. To work around some further issues with triggering, we prototyped an optimizer pass which inlines all variables. It also simplifies snapshot field function calls of constructors, for example by transforming `Foo_field_x(Foo_cons(a,b,c,d))` into `c`, given that the field `x` is the third parameter of the `Foo` constructor function. Note that due to Viper bugs or missing triggers in our generated Viper code the optimizer is currently required for some of the examples presented in this thesis to verify.

Chapter 4

Verifying OpenDP

This chapter discusses how the implemented Prusti features described in the previous chapter are used to verify the desired properties of OpenDP. As outlined in the background chapter, OpenDP uses numerous Rust features which do not yet have support in Prusti, among them closures, loops, iterators and generics.

This chapter starts by describing a simplified clamping transformation which only uses features currently supported by Prusti. The code is then annotated with Prusti specifications to verify the appropriate output domain as well as the data-independent runtime exceptions properties. The chapter concludes with an approach to verifying the stability guarantees once additional Rust language features are supported by Prusti. This is illustrated with an implementation directly written as Viper code.

4.1 Simplified Clamping

Since Prusti does not yet support all the features needed for OpenDP, notably closures, loops, iterators and generics, we further simplify the code and reduce it down to its core functionality.

This implementation of clamping over all elements of a Vector can be seen in Listing 23. Due to the lack of generics we utilize two separate enums for errors instead of one generic one. Instead of using a generic transformation holding a closure, the code is limited to just one fixed transformation with a struct `ClampTransform` representing a clamping transformation containing the `Bounds`. We define a `clamp` method on `ClampTransform` which clamps the provided integer to the `Bounds`, as well as a `clamp_impure` method which does the same while not being `#[pure]`, returning the `ClampTransform` for further use and handling the case that the `Bounds` are invalid by returning a `FallibleI32`. As such, the `clamp` functions is essentially the same as

```
struct Bounds { max: i32, min: i32 }
struct ClampTransform { bounds: Bounds }
enum FallibleVec { Ok(Vector), Err }
enum FallibleI32 { Ok(i32), Err }

fn apply_row_by_row(transform: ClampTransform, data: Vector)
-> (FallibleVec, ClampTransform) {
    if data.len <= 0
      { return (FallibleVec::Ok(data), transform); }

    let l = data.len;
    apply_row_by_row_rec(transform, data, l - 1)
}

fn apply_row_by_row_rec(
    mut transform: ClampTransform, mut data: Vector, idx: usize,
) -> (FallibleVec, ClampTransform) {
  if idx >= 1 {
    (data, transform) = match apply_row_by_row_rec(transform,
    ↪  data, idx - 1) {
      (FallibleVec::Ok(vec), transform) => (vec, transform),
      (FallibleVec::Err, transform) =>
          return (FallibleVec::Err, transform),
    };
  }

    let (cur, data) = data.impure_get(idx);
    let (clamped, transform) = transform.clamp_impure(cur);
    if let FallibleI32::Ok(clamped) = clamped {
        let data = data.set(idx, clamped);
        return (FallibleVec::Ok(data), transform);
    }
    (FallibleVec::Err, transform)
}
```

Listing 23: Simplified clamping transformation only using features supported by Prusti

the function seen in Listing 8, while `clamp_impure` is essentially the same function as seen in Listing 22.[1] The existence of `clamp_impure` is necessary, due to a lack of support for references, as described for the `impure_get` function on the `Vector` type and as such the `ClampTransform` passed to `clamp_impure` is never modified.

Instead of looping over an iterator to apply the transformation to all elements of a vector, we use a recursive definition as can be seen in `apply_row_by_row_rec`, which processes the vector starting from the end. At first it recurses by calling itself with the index reduced by one and handles the case that the recursive call fails by returning an error (Line 22). After the recursion, the function processes the value at index `idx` by getting its current value from the vector, applying `clamp_impure` to it and finally writing the new value back to the vector. In the case that this call to `clamp_impure` fails, an error is returned (Line 32). This function too takes a `ClampTransform` and returns it unmodified, due to the lack of support for references.

The function `apply_row_by_row` then wraps `apply_row_by_row_rec` to provide an easy to use signature.

4.2 Appropriate Output Domain

Recall the definition of the appropriate output domain property: Applying a transformation to an element of the input domain results in an element of the output domain or alternatively raises a runtime exception. For the clamping transform presented here the input domain is the set of all `i32` and the output domain are the integers which are within the `bounds`. As such the appropriate output domain property holds in this case if after applying a clamping transformation to a vector, all elements contained in it are within the specified clamping bounds or a runtime error is raised. Note that unlike the original OpenDP implementation we do not keep track of the input and output domains in the transformation type. However, we can statically ensure that the appropriate output domain property holds, using Prusti assertions to formally verify this. Listing 24 shows the specification required. To work with the `FallibleI32` type we define a function `unwrap_i32` to access the value in the `Ok` case if we can prove that it is not an `Err`; for this we define a function that we use to indicate that the `Err` branch in `unwrap_i32` is unreachable.[2] The same is done for `FallibleVec`.

`apply_row_by_row_rec` is then annotated with a precondition on line 12, ensuring that the `idx` is a valid index for the vector. The first postconditions on line 13 ensures that the transformation argument is not modified. We also

[1] The exact definitions can be seen in the Appendix in Listing 33.

[2] In theory this should be possible by using the Rust `unreachable!()` macro. This is however not yet supported in Prusti.

```
1  #[trusted] #[pure] #[requires(false)]
2  fn unreachable_i32() -> i32 { unreachable!() }
3
4  impl FallibleI32 {
5      #[pure] #[requires(matches!(self, FallibleI32::Ok(_)))]
6      fn unwrap_i32(self) -> i32 {
7          match self { FallibleI32::Ok(val) => val,
8                       FallibleI32::Err => unreachable_i32() }
9      }
10 }
11
12 #[requires(data.len >= 1)] #[requires(idx < data.len)]
13 #[ensures(result.1 === transform)]
14 #[ensures(transform.bounds.min <= transform.bounds.max <==>
15     matches!(result.0, FallibleVec::Ok(_)))]
16 #[ensures(matches!(result.0, FallibleVec::Ok(_)) ==>
17     result.0.unwrap_vec().len === data.len)]
18 #[ensures(matches!(result.0, FallibleVec::Ok(_)) ==>
19     forall(|i: usize| (i > idx && i < data.len)  ==>
20     result.0.unwrap_vec().get(i) == data.get(i)))]
21 #[ensures(matches!(result.0, FallibleVec::Ok(_)) ==>
22     forall(|i: usize| (i <= idx && i < data.len) ==>
23     result.0.unwrap_vec().get(i) ==
       ↪  transform.clamp(data.get(i))))]
24 fn apply_row_by_row_rec(
25     mut transform: ClampTransform, mut data: Vector, idx: usize
26 ) -> (FallibleVec, ClampTransform)
```

Listing 24: Contracts ensuring the appropriate output domain property property holds for the clampin transformation

guarantee that if and only if the min bound is less than or equal to the max bound, then the function will return an `Ok` value with the postcondition on line 14. The last three postconditions are conditioned on the function returning an `Ok` result, as otherwise, the unwrapping of the vector is not well-defined, because the precondition of the `unwrap_vec` function would not be fulfilled. The first of these three postconditions (Line 16) describes that the returned vector has the same length as the argument vector. The second postcondition (Line 18) asserts that all elements of the vector with an index greater than `idx` remain unchanged. Finally, on line 21 the third postcondition asserts that all elements of the vector with an index less then or equal to `idx` have been clamped.

Listing 25 shows how the `apply_row_by_row` function uses the guarantees

```
#[ensures(result.1 === transform)]
#[ensures((transform.bounds.min <= transform.bounds.max ||
↪  data.len == 0) <==>
    matches!(result.0, FallibleVec::Ok(_)))]
#[ensures(matches!(result.0, FallibleVec::Ok(_)) ==>
    result.0.unwrap_vec().len === data.len)]
#[ensures(matches!(result.0, FallibleVec::Ok(_)) ==>
    forall(|ip: usize|(ip < data.len) ==>
    result.0.unwrap_vec().get(ip) ==
    ↪  transform.clamp(data.get(ip))))]
fn apply_row_by_row(transform: ClampTransform, data: Vector)
-> (FallibleVec, ClampTransform)
```

Listing 25: Contract for the appropriate output domain property of `apply_row_by_row`

provided by `apply_row_by_row_rec` to ensure that all values in the vector have been clamped, provided that an `Ok` result is returned. This follows directly from the postconditon of `apply_row_by_row_rec`, which is called by `apply_row_by_row` with an `idx` equal to one less than the length of the vector. Note that due to an implementation detail an additional `data.len == 0` option is needed on the `ensures` specification which asserts that the result will be `Ok` if and only if the bounds are valid, as, in the case that the length of the vector is 0, the bounds will never be checked. With this we have shown that the appropriate output domain property holds for `ClampingTransform`.

4.3 Data-independent Runtime Exceptions

When introducing `rel` we showed in Listing 22 how a clamping function on integers can be proven to have data-independent runtime exceptions. This specification can be applied to the `clamp_impure` method on the `ClampTransform` to allow us to extend the data-independent runtime exceptions property from clamping integers to clamping vectors. Listing 26 shows the postconditions needed in addition to the contract from the appropriate output domain property from Listings 24 and 25 to accomplish this. For `apply_row_by_row_rec` to the left of the implication we assert that in both executions the `idx` and `transform` have to be the same or, in other words, that only the `data` vector is allowed to differ between the two executions. Using this contract this hyperproperty of the function can be verified using Prusti.

However, applying this postcondition to `apply_row_by_row` results in a verification failure. This is because `apply_row_by_row` does in fact *not* satisfy the data-independent runtime exceptions property, since as stated before the bounds are never checked for vectors of length 0. As such the implementa-

```
#[ensures((rel0(idx) == rel1(idx) && rel0(&transform) ===
↪  rel1(&transform)) ==>
match (rel0(&result.0), rel1(&result.0)) {
  (FallibleVec::Err,FallibleVec::Err) => true,
  (FallibleVec::Ok(_),FallibleVec::Ok(_)) => true,
  _ => false,
})]
fn apply_row_by_row_rec(mut transform: ClampTransform, mut data:
↪  Vector, idx: usize)

#[ensures(rel0(&transform) === rel1(&transform) ==>
match (rel0(&result.0), rel1(&result.0)) {
  (FallibleVec::Err,FallibleVec::Err) => true,
  (FallibleVec::Ok(_),FallibleVec::Ok(_)) => true,
  _ => false,
})]
fn apply_row_by_row(transform: ClampTransform, data: Vector)
```

Listing 26: Additional postconditions that together with the specifications from Listings 24 and 25 ensure the data-independence of errors

tion can never raise an error if the vector is empty. While OpenDP mitigates this issue by only allowing valid `Bounds` objects to be constructed this constitutes a mistake in their LaTeX proofs: While they do require that the `row_function`, which in our case is the `clamp_impure` function, only produces data-independent errors, this does not cover the case, that given that the `row_function` raised an error, we learn that the vector cannot have been empty.

To allow `apply_row_by_row` to verify, we can introduce the additional requirement that the two vectors in the two executions have to be of the same length, by adding an additional requirement of `rel0(&data.len) === rel1(&data.len)` to the left of the implication. In addition to demonstrating that Prusti can in fact successfully verify this hyperproperty, this also shows that for bounded DP, the data-independent runtime exceptions property holds, as in bounded DP only vectors of the same length can be neighboring. However, to show the property for unbounded DP, the implementation of the function would have to be modified.

This exemplifies that, even for simple functions, formally verifying hyperproperties is a worthwhile endeavor: Despite the relative simplicity and similarity between these functions, a small implementation detail in the wrapping function leads to the property being violated in a subtle way.

4.4 Stability Guarantee

Due to missing Prusti features, including arithmetic operations in contracts, we are unable to verify the stability guarantee property in Prusti. We are however able to show the property directly in Viper as a proof of concept. We directly write the Viper equivalent of our `Vector` type in the form of a predicate `Vec` and `get` and `len` functions, as well as a `set` method. We also define a `valid_index(vec, idx)` function as a shorthand for `0 <= idx < len(v)`.[3]

Using this we define a Viper version of the clamping function as seen in Listing 27. This function takes a `Vec` and uses a loop to clamp every element. The `function clamp(v: Int): Int`, which is called in the loop, does not have a body which demonstrates that everything in this section also holds for arbitrary transformations as long as they are performed on a element-by-element basis. The loop invariant of the `clamp_vector` function encodes the same concept as the postconditions on `apply_row_by_row_rec` did.

Recall the definition of the stability guarantee for a 1-stable transformation, such as clamp: For any pair of vectors u and v which have a symmetric distance of d, the results of applying the transformation to u and v have a symmetric distance of d or less.

To describe the concept of the symmetric distance between two vectors we will utilize bijective functions, which we model using a Viper domain as can be seen in Listing 28, where we define `translate` and `translate_invert`

```
1  method clamp_vector(V: Ref)
2    requires Vec(V) ensures Vec(V) && len(V) == old(len(V))
3    ensures forall i: Int :: valid_index(V, i) ==>
4      get(V, i) == old(clamp(get(V, i))) {
5    var i: Int := 0;
6    while(i < len(V))
7      invariant Vec(V) && len(V) == old(len(V)) && 0 <= i
8      invariant forall x: Int :: (valid_index(V, x) && x < i) ==>
       ↪  get(V, x) == clamp(old(get(V, x)))
9      invariant forall x: Int :: (valid_index(V, x) && x >= i) ==>
       ↪  get(V, x) == old(get(V, x)) {
10       set(V, i, clamp(get(V, i)))
11       i := i + 1;
12   }
13 }
```

Listing 27: Clamping of all values in a vector directly implemented in Viper

[3]The full listing of this can be seen in the appendix in Listing 32

```
domain Mapping {
    function keys(self: Mapping) : Set[Int]
    function values(self: Mapping): Set[Int]
}
function translate(m: Mapping, key: Int) : Int
    requires |keys(m)| == |values(m)| && key in keys(m)
    ensures result in values(m)
function translate_invert(m: Mapping, value: Int) : Int
    requires |keys(m)| == |values(m)| && value in values(m)
    ensures result in keys(m)

function is_bijection(m: Mapping) : Bool {
    |keys(m)| == |values(m)| &&
    forall v : Int :: v in values(m) ==>
    ↪ translate(m,translate_invert(m, v)) == v
}
```

Listing 28: Viper domain modeling a bijective function

functions, which are used to apply the mapping or its inverse respectively. The domain defines functions `keys` and `values` which return the set that corresponds to the domain and range of the mapping function. We then define a function `is_bijection` which uses the `translate` and `translate_invert` functions to determine whether a function is in fact bijective.

Symmetric distance of 0 If for two vectors A and B it holds that $d_{sym}(A, B) = 0$ then $len(A) = len(B)$ and there exists a bijective function m from the indexes of A to the indexes B such that $B[m(i)] = A[i]$ for all indexes i of A. Using this fact, the `clamp_vector` method can be extended with pre- and postconditions to model this case as seen in Listing 29. We introduce a precondition requiring that the lengths of the two vectors in the different runs are the same. We also require that the keys and values of the mapping are the same and that the sets of keys and values is exactly the same as the set of valid indexes for the vector. We then require that the symmetric distance is 0 initially by asserting that our mapping m can map each index of the `rel0` vector to an equivalent one of the `rel1` vector. We then simply have the same assertion as a postcondition.

Symmetric distance of 1 The symmetric distance of two vectors A and B can only be exactly 1 if they differ in length by exactly 1. In which case for $d_{sym}(A, B) = 1$ it must hold that $Ms(A) \subset Ms(B)$ for the case that $len(A) + 1 = len(B)$. We show the slightly more general case of $len(A) \leq len(B) \wedge Ms(A) \subseteq Ms(B)$. Listing 30 shows the case in which the `rel0` vector is longer and the `rel1` vector is contained within it. We have preconditions requiring that the `rel0` vector is longer than or equal in length to the `rel1`

```
method clamp_0(V: Ref, m: Mapping)
requires Vec(V)
// ...
requires rel(len(V),0) == rel(len(V),1) && rel(m,0) == rel(m,1)
requires keys(m) == values(m)
requires is_bijection(m)
requires forall i: Int :: i in keys(m) <==> valid_index(V, i)
//Symmetric distance is 0
requires forall i: Int :: rel(valid_index(V, i),0) ==>
↪  rel(get(V,translate(m,i)),0) == rel(get(V, i),1)
// Mapping still holds
ensures is_bijection(m)
ensures forall i: Int :: rel(valid_index(V, i),0) ==>
↪  rel(get(V,translate(m,i)),0) == rel(get(V, i),1)
```

Listing 29: Additional specification for Listing 27 to show the stability guarantee for a symmetric distance of 0

vector. We then require that the values of m are a subset of the indexes of `rel0` and that the keys of m are exactly the same set as the indexes of `rel1`. As such we can then require that each index of `rel1` can be mapped to an index in `rel0` such that they share the same value. And since m is a bijection this means that the symmetric distance must be the difference in length between the two. By requiring that the mapping still holds as a postcondition we prove that the symmetric distance does not increase.

Symmetric distance of 2 The previous section covers some of the cases included in $d_{sym}(A, B) = 2$. The left over cases are the ones where A and B have the same length and each one has an element not in the other. We can model this by allowing our mapping to not have a corresponding index in the other vector in exactly one case. Listing 31 shows this with the extra index stored in an argument `extra_B_index`.

In conclusion while we do not cover all possible cases for the stability guarantee, we do cover the most important ones, namely:

- $d_{sym}(A, B) = 0$
- $len(A) \leq len(B) \quad \wedge \quad Ms(A) \subseteq Ms(B)$ which includes all cases of $d_{sym}(A, B) = 1$ except for switching A and B. The case of $d_{sym}(A, B) = 1$ corresponds to unbounded differential privacy, where two datasets are considered neighboring if one can be obtained form the other by adding or removing one entry.
- $len(A) = len(B) \quad \wedge \quad d_{sym}(A, B) = 2$ this corresponds to bounded differential privacy, where we consider two datasets neighbors if one can be obtained from the other by changing one entry.

```
method apply_clamp_longer(V: Ref, m: Mapping)
requires Vec(V)
// ...
requires rel(len(V),0) >= rel(len(V),1) // rel0 is longer
requires rel(m,0) == rel(m,1)
requires is_bijection(m)
requires forall i: Int :: i in values(m) ==> rel(valid_index(V,
↪  i),0)
requires forall i: Int :: i in keys(m) <==> rel(valid_index(V,
↪  i),1)
requires forall i: Int :: rel(valid_index(V, i),1) ==>
↪  rel(get(V,translate(m,i)),0) == rel(get(V, i),1)
// Mapping still holds
ensures is_bijection(m)
ensures forall i: Int :: rel(valid_index(V, i),1) ==>
↪  rel(get(V,translate(m,i)),0) == rel(get(V, i),1)
```

Listing 30: Additional contract for Listing 27 to show the stability guarantee for a symmetric distance of 1

```
method clamp_same_len(V: Ref, m: Mapping, extra_B_index: Int)
requires Vec(V)
// ...
requires rel(len(V),0) == rel(len(V),1)
requires rel(m,0) == rel(m,1) && rel(extra_B_index,0) ==
↪  rel(extra_B_index,1)

requires keys(m) == values(m)
requires is_bijection(m)
requires forall i: Int :: i in keys(m) <==> valid_index(V, i)
requires forall i: Int :: i in values(m) <==> valid_index(V, i)
// Symmetric distance is at most 2
requires forall i: Int :: rel(valid_index(V, i),0) ==>
↪  (rel(get(V,translate(m,i)),0) == rel(get(V, i),1)) || i ==
↪  extra_B_index
// Mapping still holds
ensures is_bijection(m)
ensures forall i: Int :: rel(valid_index(V, i),0) ==>
↪  (rel(get(V,translate(m,i)),0) == rel(get(V, i),1)) || i ==
↪  extra_B_index
```

Listing 31: Additional specifications for Listing 27 to show the stability guarantee for the case where the two vectors have the same length and each has one element not contained in the other

In addition to the `rel`-based `clamp_vector` method presented here we implemented an alternative, more manual approach to show the stability guarantee property, by instead defining the `clamp_vector` method as taking two vectors and a `Mapping` without using the `rel` Viper plugin. In this case the method applies clamping to both vectors and the `Mapping` relates the two vector arguments. This is mostly equivalent to the internal encoding produced by the Viper plugin when using the `rel` keyword, as it constructs a product program. However, this manual approach suffers from the downside that not all of the cases shown can share the same canonical body from Listing 27, as for example in the case where the second vector is longer by one element this last element of the second vector would have to be clamped separately after the loop. The manual approach may be more intuitive to understand and allows for the method to be called directly, as the two vector arguments are available to the caller and thus the mapping can be computed. This is opposed to the `clamp_vector` method using `rel`, where it is impossible to directly construct such a `Mapping`, since it relates the elements of `rel0` and `rel1`, which represent the abstract concepts of vectors in different executions. The `rel`-based `clamp_vector` method represents the idea that, given that such a mapping exists, this property is upheld. We choose to present the `rel` version here because it resembles the original clamping method more closely, which takes a single vector argument and shares one method body across all presented cases. Since both the `rel`-based and manual approach verify correctly, we are convinced that this approach is correct.

4.5 Performance Evaluation

While performance has not been a focus of the implementation work in this project, verification passing in a reasonable amount of time is still important. We took approximate time measurements on an AMD Ryzen 7 5800X with 32GiB 3200 MHz DDR4 RAM using both of the available Viper backends, the verification-condition-generation-based carbon as well as the symbolic-execution-based silicon.

Verifying the appropriate output domain and data-independent runtime exceptions properties with Prusti using the silicon Viper backend takes around 2 minutes, while the carbon backend did not finish. On the other hand, the stability guarantee property as shown directly in Viper takes around 6 seconds to verify using the carbon backend and around 4 to 8 minutes when using silicon.[4]

[4]Silicon version 1.1-SNAPSHOT (cb319dee@(detached))

Chapter 5

Conclusion

As part of this project we implemented additions to the Prusti formal verifier, including support for foundational Rust language features as well as hyperproperties utilizing a preexisting Viper plugin. We use the features we implemented to verify properties, including hyperproperties, of a simplified version of the OpenDP library's clamping transformation. In particular we show that the appropriate output domain property holds, while the data-independent runtime exceptions property is not satisfied: The resulting vector is guaranteed to only contain values that are within the specified bounds, but if the transformation results in an error, this error is not guaranteed to be independent of the passed data vector. In particular, it is not possible for the transformation to result in an error as a result of being given invalid bounds if the given data is the empty vector. In OpenDP this bug is mitigated by only allowing valid bounds to be constructed, however, this still technically constitutes a mistake in the manual OpenDP LaTeX proofs in the general case, since an attacker who observes an error message generated by a transformation can learn that the data cannot have been the empty vector.

Using the underlying Viper mechanisms we also show the stability guarantee property of the transformation: After applying the transformation to two neighboring datasets, the two datasets remain neighboring. We show this for the case of the symmetric distance for both bounded and unbounded differential privacy. Showing that this property can be proven in Viper demonstrates the feasibility of proving the stability guarantee in Prusti, given that some additional features are implemented.

This work is part of an ongoing rewrite of Prusti and demonstrates that the new Prusti architecture can be leveraged to swiftly implement new features, encompassing Rust language features such as enums, Prusti-specific features such as `#[pure]` functions and integration with an external Viper plugin. It also demonstrates that applying formal verification tools, in particular Prusti, to differential privacy libraries is feasible and can uncover

subtle errors which are difficult to spot efficiently without the use of automated verification tools.

5.1 Future Work

Since the stability guarantee was shown directly in Viper and not in Prusti, in a next step one can first implement the prerequisite Prusti features and then verify this property in Prusti. Implementing additional Rust language features in Prusti such as loops, iterators, generics and closures which are heavily utilized by OpenDP, but not yet available in Prusti, will allow for the verification of code which is closer to the original OpenDP code base. The theoretical foundations of these features have already been established in the previous implementation of Prusti.

Lastly, this thesis focused on verifying properties of transformations while not considering measurements. To prove the correctness of differentially private algorithms it is necessary to verify measurements as well. However, verifying properties of measurements involves modeling their inherently probabilistic nature, which poses a considerable challenge for both Prusti and Viper.

Appendix A

Appendix

A.1 Additional Listings

```
predicate Vec(v: Ref)

function valid_index(v: Ref, idx: Int): Bool
  requires Vec(v) { 0 <= idx < len(v)  }

function get(v: Ref, idx: Int): Int
  requires Vec(v) && valid_index(v, idx)

function len(v: Ref): Int requires Vec(v)

method set(v: Ref, idx: Int, val: Int)
    requires Vec(v) && valid_index(v, idx)
    ensures Vec(v) && len(v) == old(len(v))
    ensures get(v, idx) == val
    ensures forall i: Int :: (i >= 0 && i< len(v) && i != idx)
    ↪  ==> get(v, i) == old(get(v, i))
```

Listing 32: A vector datatype modeled directly in Viper

```
impl ClampTransform {
    #[ensures(result.bounds === bounds)]
    fn make_clamp(bounds: Bounds) -> Self {
        Self { bounds }
    }

    #[pure]
    #[requires(self.bounds.min <= self.bounds.max)]
    fn clamp(self, data: i32) -> i32 {
        if data < self.bounds.min {
            (self.bounds.min)
        } else if data > self.bounds.max {
            (self.bounds.max)
        } else {
            (data)
        }
    }

    #[ensures(self.bounds.min <= self.bounds.max <==>
    ↪ matches!(result.0, FallibleI32::Ok(_)))]
    #[ensures(self.bounds.min <= self.bounds.max ==>
    ↪ result.0.unwrap_i32() === self.clamp(data))]
    #[ensures(result.1 === self)]
    #[ensures(rel0(&self.bounds) === rel1(&self.bounds) ==>
    ↪ match (rel0(&result.0), rel1(&result.0)) {
        (FallibleI32::Err,FallibleI32::Err) => true,
        (FallibleI32::Ok(_),FallibleI32::Ok(_)) => true,
        _ => false,
    })]
    fn clamp_impure(self, data: i32) -> (FallibleI32, Self) {
        if self.bounds.min > self.bounds.max {
            (FallibleI32::Err, self)
        } else if data < self.bounds.min {
            (FallibleI32::Ok(self.bounds.min), self)
        } else if data > self.bounds.max {
            (FallibleI32::Ok(self.bounds.max), self)
        } else {
            (FallibleI32::Ok(data), self)
        }
    }
}
```

Listing 33: Full implementation of `ClampTransform`

Bibliography

[1] The OpenDP Team, "OpenDP Library." [Online]. Available: https://github.com/opendp/opendp

[2] S. Casacuberta, M. Shoemate, S. Vadhan, and C. Wagaman, "Widespread underestimation of sensitivity in differentially private libraries and how to fix it," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 471–484. [Online]. Available: https://doi.org/10.1145/3548606.3560708

[3] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30.

[4] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[5] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Computer Science Logic*, L. Fribourg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19.

[6] M. J. Parkinson and A. J. Summers, "The Relationship Between Separation Logic and Implicit Dynamic Frames," *Logical Methods in Computer Science*, vol. Volume 8, Issue 3, Jul. 2012. [Online]. Available: https://lmcs.episciences.org/802

[7] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity*

Language Technology, ser. HILT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: https://doi.org/10.1145/2663171.2663188

[8] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014. [Online]. Available: http://dx.doi.org/10.1561/0400000042

[9] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, nov 2019. [Online]. Available: https://doi.org/10.1145/3324783

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- (●) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Automated Verification of a Rust Differential Privacy Library

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Arnold	Till

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 9. March 2024	J. Arnold

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard