École Polytechnique Fédérale de Lausanne

Time Reasoning and Secure Information Flow in a Rust Verifier

by Alexandre Pinazza

Master Thesis

Approved by the Examining Committee:

Prof. Viktor Kunčak, EPFL
Thesis Advisor

Prof. Dr. Peter Müller, ETH
Thesis Advisor

Aurel Bílý, ETH
Thesis Supervisor

EPFL IC IINFCOM LARA
INR 318 (Bâtiment INR)
Station 14
CH-1015 Lausanne

March 24, 2023

Acknowledgments

I would like to thank Aurel Bílý and Prof. Viktor Kunčak for their supervision, Vytautas Astrauskas and Frederico Poli for their Prusti insights, Marco Eilers for our Modular Product Programs discussions, Prof. Dr. Peter Müller for the opportunity to do my Master's thesis in his laboratory and all the Programming Methodology laboratory for their welcome. I would also like to thank Prof. Mathias Payer for this LaTeX thesis package.

Lausanne, March 24, 2023 Alexandre Pinazza

Abstract

Rust is a modern programming language with a focus on performance and memory safety. It is designed for low- and high-level applications and guarantees memory safety with its unique borrow and type checker. The performance and safety standards of Rust make it a compelling programming language to implement security-critical applications such as operating system kernels.

Even though the Rust compiler prevents memory bugs, it cannot prevent functional bugs. To remedy this Astrauskas et al. developed Prusti, a static verifier for Rust programs build on top of Viper.

In this project, we added to Prusti the ability to reason about the upper and lower bounds of functions' runtime using time credits and time receipts. Additionally, we also added the ability to reason about hyperproperties using Modular Product Programs and use it to check secure information flow and thus preventing information leaks.

Contents

Chapter 1

Introduction

Rust is a multi-paradigm programming language with a focus on performance and safety. It is designed to be suitable for low- and high-level programming and can be used to program drivers, operating system kernels, embedded systems, front- and back-end applications, and much more. The unique feature of Rust is its ownership type system for automatic memory management and safety. It prevents memory corruption and data races at compile time.

Rust's high safety standards and performance make it a compelling programming language to implement security-critical code. Its type system already prevents various bugs at compile time but others can still pass through its checks. To verify the absence of functional bugs, Astrauskas et al. introduced *Prusti* [1], a formal verification tool for Rust programs built on top of the *Viper* verifier [17]. However, it does not provide mechanisms to reason about functions' runtime nor to check for secure information flow. Consequently, Prusti does not see any difference between the functions shown in Listing 1 and Listing 2. Reasoning about the asymptotic runtime is not enough to distinguish these functions as they both run in linear time with respect to the length of the arrays. To differentiate these functions, we must reason about their lower bound. Moreover, Prusti does not have any security level notion. Thus it cannot be used to prevent information leaks as in Listing 3 where depending on the value of a secret Boolean – `secret` – different integers will be printed on the standard output. Additionally, both features can be combined to prevent some timing attacks by proving that the upper and lower bounds of functions are the same and that their expression does not depend on secret values. For example, using the array comparison function from Listing 1 for password checking is unsafe as the runtime of the function leaks information on where the two arrays differ (if they differ).

```rust
fn array_cmp<T: Eq>(a1: &[T], a2: &[T]) -> bool {
    assert!(a1.len() == a2.len());
    let mut i = 0;
    while i < a1.len() {
        if a1[i] != a2[i] {
            return false;
        }
        i += 1;
    }
    return true;
}
```

Listing 1: Equal size arrays comparison function with an early return. The function's runtime varies depending on where the first different elements are within the arrays.

```rust
fn array_cmp<T: Eq>(a1: &[T], a2: &[T]) -> bool {
    assert!(a1.len() == a2.len());
    let mut i = 0;
    let mut res = true;
    while i < a1.len() {
        res &= a1[i] == a2[i];
        i += 1;
    }
    return res;
}
```

Listing 2: Equal size arrays comparison function without early return. This function compares all the elements of the arrays before returning the result.

```rust
fn foo(secret: bool) {
    if secret {
        print_int(0);
    } else {
        print_int(1);
    }
}
```

Listing 3: Secret Boolean leaked through the standard output: if `0` (resp. `1`) is printed on the standard output then we know that `secret` equals `true` (resp. `false`) thus if an adversary is able to access the standard output they could infer the value of `secret`.

In this project, we added two new functionalities to Prusti.

- The ability to reason about the runtime of functions. To do so we used *time credits* [2] and its dual *time receipts* [16]. Section 2.3 gives an overview of them, Section 3.1 explains how they can be encoded using Viper and Section 4.1 presents how they have been implemented into Prusti.
- The verification of non-interference. We used *Modular Product Programs* [8, 10] to do so. Section 2.4 gives an overview of them, Sections 3.2 and 4.2 explain how they were added into Prusti.

Chapter 2

Background

2.1 Viper

Viper [17] is an intermediate verification infrastructure created by Müller et al. It is similar to Boogie [15] or Why3 [12] that are able to verify pre- and post-conditions of functions and loops in imperative programs. A distinguishing feature of Viper is its native support of permission-based reasoning. This reduces the effort required to build front ends for various programming languages. Viper has multiple front ends such as Nagini [9] for Python, Gobra [20] for Go, and Prusti [1] for Rust. Viper supports control flows such as if-then-else, while-loops, and arbitrary gotos.

2.1.1 Specification

To perform modular verification, Viper does not inline methods body and instead requires pre- and post-conditions. Viper's loops also require loop invariants, which are checked at the beginning of loops, on loop iterations, and on loop exits. Listing 4 showcases how these specifications are expressed in a Viper program.

2.1.2 Heap

To model heap references, Viper has a `Ref` type which can be seen as a pointer to a structure containing an entry for each field defined using a `field <name>: <Type>` defi-

```
method sum(n: Int) returns (res: Int)
    requires n >= 0
    ensures res >= 0
{
    var i: Int := 0
    while i < n
        // as the loop invariant is checked on loop exits,
        // we need i <= n
        invariant 0 <= i && i <= n
    {
        res := res + i
        i := i + 1
    }
}
```

Listing 4: Viper specifications.

nition in the program. Then for every access to a field from a reference, Viper will check that we have the necessary permission to access this field. Currently, permissions are real numbers between 0 (no permission) and 1 (full permission). When we read a field, we need non-zero permission; when we write to a field, we need full permission. To specify that a method requires permission to fields of a reference, Viper provides the `acc(<ref>.<field>, <permission amount>)` accessibility predicates, which take two arguments, the first one is the reference and the field to which we want access and the second is the permission amount. The permission amount can be omitted, defaulting to full permission. Listing 5 showcases a program that defines two fields `f` and `g` and then a method that takes a reference `a` as argument and copies the `f` field of this reference into its `g` field. To be able to do so it needs to have some permission to access `a.f`, in this example half permission, and full permission to access `a.g` which are specified in the method's pre-conditions. Loops do not inherit the permissions from their enclosing scope, so the invariant must specify its required permission, as shown in Listing 6.

Viper does not allow a full separation logic specification, it doesn't support disjunctions of access predicates such as `acc(a.f) || acc(a.g)`, nor access predicates in negations, nor in guards of implications.

To modify heap permissions, Viper provides the `exhale P` and `inhale P` statements. The former adds the permissions part of `P` and assumes the rest. Similarly, the latter checks that enough permission is held, remove it and assert the rest. If the permission predicate `acc` is used in an `assert` statement, Viper will only check that the program holds at least this amount of permission and will not change the amount.

```
// Defines two fields: f and g
field f: Int
field g: Int

// This method copies the f field from
// the given reference into its g field
method copy_f_into_g(a: Ref)
    requires acc(a.f, 1 / 2) // requires read permission on a.f
    requires acc(a.g, 1 / 1) // requires write permission on a.g
{
    var tmp: Int := a.f // to read a.f we must have some permission to a.f
    a.g := tmp          // to write into a.g must have full permission to a.g
    // the following would fail as we do not have full permission to a.f
    a.f := 42
}
```

Listing 5: Field and permission check.

```
field c: Int

method count(r: Ref, n: Int)
    requires acc(r.c) && 0 <= n
    ensures acc(r.c)
{
    var i: Int := 0
    while (i < n)
        invariant 0 <= i && i <= n
        // to write to r.c we must have full permission on r.c
        invariant acc(r.c)
    {
        r.c := r.c + 1
        i := i + 1
    }
}
```

Listing 6: Loop invariant with access predicate.

2.1.3 Definitions

To improve readability, modularity, and expressive power, Viper allows the definition of functions, methods, and predicates. Functions are pure and cannot have side effects, thus they can be used inside code and specifications. Methods are impure and can have side effects (i.e. modifying the heap). Methods must be called from a dedicated statement whereas functions can be called inside expressions, as a result if-then-else and loop's conditions must be side-effect free. Predicates allow specification codes such as implications, accessibility predicates, and recursion, they can only be used inside specifications. All of these definitions can be left empty to allow more abstractions and modeling power. Viper uses an isorecursive representation for predicates, so it does not automatically fold or unfold them. Instead, the `fold` and `unfold` statements and the `unfolding ... in ...` expression perform them. Listing 7 showcases all these constructs and their use. Viper also defines access permissions for predicates, it has the effect to multiply all accessibility predicates used in the predicate's definition by the predicate permission. For example, the `len` function of Listing 7 only needs read permission to `l` so by requiring `acc(list(l), 1 / 2)` and by unfolding it, we only gain half the permission to `l.h`, `l.t`, and `list(l.t)`. Access permission on predicates can be greater than 1 and using it with abstract predicates can be used to reason about resources.

When a method is called, its required permissions are consumed from the caller's context and the permissions from the method post-conditions are added to the caller's context. Listing 8 implements a method – `move_a_to_b` – that takes full ownership of the `a` field from the given reference and then moves it into the `b` field from the freshly created return reference. This method is called from the `main` method where the permission to `r1.a` is consumed by the call and the permission to `r2.b` is gained.

2.2 Prusti

Built on top of Viper, Astrauskas et al. created *Prusti* [1] a formal verification tool for Rust programs. It can prove the absence of panics in a large subset of safe Rust code in addition to the verification of pre- and post-conditions and loop invariants.

Prusti uses Rust's powerful macro system to annotate the code with specifications as shown in Listing 9. This allows Prusti to reuse Rust's front end and erase the specifications during full compilation. Moreover, Rust's type checker and borrow checker are executed before calling Prusti so only type-safe programs are verified by Prusti.

```
// In this example a null reference is an empty list
field h: Int
field t: Ref

predicate list(l: Ref) {
    (l != null) ==>
    (acc(l.h) && acc(l.t) && list(l.t))
}

function len(l: Ref): Int
    // we only need read permission
    requires acc(list(l), 1/2)
{
    l == null ? 0 :
        // to satisfy the recursive call precondition,
        // we need to unfold list(l)
        unfolding acc(list(l), 1/2) in len(l.t) + 1
}

method setHead(l: Ref, v: Int)
    requires list(l) && len(l) > 0
{
    unfold list(l)
    l.h := v
}

method cons(hd: Int, tl: Ref) returns (res: Ref)
    requires list(tl)
    ensures list(res)
{
    res := new(*)
    // here the * is to gain access permission to all
    // the fields of the reference
    res.h := hd
    res.t := tl
    fold list(res)
}
```

Listing 7: Linked list in Viper.

```
field a: Int
field b: Int

method move_a_to_b(r: Ref) returns (res: Ref)
    requires acc(r.a)
    // old(r.a) refers the value of r.a at the beginning of the method
    ensures acc(res.b) && old(r.a) == res.b
{
    res := new(b) // only gain permission for b
    res.b := r.a
}

method main() {
    var r1: Ref
    var r2: Ref

    r1 := new(a)
    // we only have permission to r1.a
    r1.a := 42

    // lose permission to r1.a
    r2 := move_a_to_b(r1)
    // gain permission to r2.b

    assert acc(r2.b) && r2.b == 42

    assert acc(r1.a) // this would fail
}
```

Listing 8: Permission consumed and produced by methods.

```
#[requires(x > 0 && y > 0)]
#[ensures(result >= x && result >= y)]
#[ensures(result == x || result == y)]
fn loop_max(x: u32, y: u32) -> u32 {
    let mut r = x;
    while r < y {
        body_invariant!(x <= r && r < y);
        r += 1
    }
    r
}
```

Listing 9: A max function implemented using a loop. The first line is Prusti syntax to express pre-condition and the two following lines express post-conditions.

Prusti receives the Mid-level Intermediate Representation representation of the program produced by the Rust compiler's front end as input. This representation is a Control Flow Graph (CFG), thus, during the encoding into Viper's IR, Prusti uses gotos instead of structured if-then-else and loops. Prusti uses Rust's expressive type system to automatically infer when to fold and unfold Viper's predicates during the encoding of the program and to give appropriate permission to references.

2.2.1 Loops Encoding

One disadvantage of using the CFG produces by the compiler is that the structured nature of if-then-else and loops from the programs is lost for arbitrary block jumps. As a consequence, Prusti does not encode loops into Viper's loops but instead in a series of if-then-else with the looping branch finishing by an `inhale false` which terminates the looping traces. Listing 10 shows Prusti's loop encoding.

2.3 Time reasoning

One approach to reason about runtime in separation logic is to use time credits [2] and their dual time receipts [16]. Time credits are a resource that gets *consumed* by any operation. It is used to reason about the upper bound of the runtime as once the count of time credits is exhausted no additional operation is allowed so a function must start its execution with enough time credits to terminate. Time receipts are the duals of time

```
while {G} {

    C1;


    body_invariant(I)!;

    C2;









}

```

```
g := G
if g {
    C1;
    exhale I;
    // havoc all variables
    // modified in the loop
    inhale I;
    C2;
    g := G
    if g {
        C1;
        exhale I;
        inhale false;
    }
}
inhale !g
```

Listing 10: Prusti's loop encoding. `G` is the condition computation, it can have side effects and be composed of multiple basic blocks so it cannot directly be used inside Viper's pure if conditions. `C1` are the basic blocks from the loop before the loop invariant `I` and `C2` are the blocks after the loop invariant. Note that the ifs shown are actually implemented using gotos.

credits and are *produced* by every operation. They are used to reason about the lower bound of the runtime by checking that when a function returns it has accumulated enough time receipts. To specify that a function `fn foo(x: Int)` runs in `3 * x + 1`, it means that it requires `3 * x + 1` time credits at the start of its execution to finish and that at the end of its execution, it will have produced `3 * x + 1` time receipts.

Sometimes, the exact runtime of functions cannot be entirely known at compile time, for example, Listing 11 shows a function that runs in linear time in the worst case – when the element is not in the array. In the best case – when the element is at the beginning of the array – the function returns in constant time. For such a function, the required number of time credits is proportional to the length of the array and the guaranteed number of time receipts cannot be greater than 1. However, we can be more precise by using the return value of the function. Indeed, if the function returns `None`, we know that it has consumed all its given time credits and produced the same amount of time receipts. If the function returns `Some(i)` then we know that it has looked at `i` elements of the array and so has produced the corresponding amount of time receipts. Moreover, it can give back the time credits it has not used by not looking at the last `n-i` elements of the array.

```rust
fn find(x: u32, l: &[u32]) -> Option<usize> {
    let mut i = 0;
    while i < l.len() {
        if x == l[i] {
            return Some(i)
        }
    }
    return None
}
```

Listing 11: Early return find function. The upper bound of this function's runtime is linear to the length of `l` but if `x` is at the beginning of `l` it will return immediately.

2.4 Modular Product Programs

Secure information flow or determinism can be expressed as *hyperproperties* that reason about multiple executions of a program. One approach to verify such program properties using automated verifiers is *Product Programs* [4]. It works by transforming a program into one that performs multiple executions of the initial program at the same time allowing reasoning about hyperproperties in a normal program. Product Programs work by creating copies for every variable and statement in a program so that each execution has its version. However, this approach does not allow modular reasoning as all method calls would also get duplicated preventing the use of hyperproperties in their specification. To allow modularity, Eilers et al. introduced *Modular Product Programs* [10] which uses activation variables to execute statements conditionally. This means that if-then-else, loops, and method calls do not need to be duplicated anymore and thus modularity can be achieved. Listing 12 shows a method `foo` and its MPP transformation.

Let us represent the transformation using $[[\mathtt{s}]]_{p_1,p_2}$ where `s` is the statements we want to transform and p_1 and p_2 are the two activation variables for the executions. Figure 2.1 shows the MPP transformation for the main statements Prusti uses.

For references, the MPP transformation duplicates each field so that each execution has its sets. This has the advantage that when a new reference is created using the `new` statement, both executions can share the created reference and thus be equal. This encoding requires duplicating each predicate and function that uses references. For non-abstract functions, Viper would unfold the function calls and be able to relate their return values. However, this does not work with abstract functions as now Viper sees two different function calls, that it cannot unfold, and thus is not able to relate the returned

```
method foo(x: Int)

    returns (res: Int)
{
    if (x < 0) {



        res := 1

    } else {
        res := bar(x)


    }
}
```

```
method foo(p1: Bool, p2: Bool,
            x1: Int, x2: Int)
    returns (res1: Int, res2: Int)
{
    t1 := p1 && x1 < 0
    f1 := p1 && !(x1 < 0)
    t2 := p2 && x2 < 0
    f2 := p2 && !(x2 < 0)
    if (t1) { res1 := 1 }
    if (t2) { res2 := 1 }
    if (f1 || f2) {
        tmp1, tmp2 := bar(f1, f2, x1, x2)
        if (f1) { res1 := tmp1 }
        if (f2) { res2 := tmp2 }
    }
}
```

Listing 12: Modular Production Programs transformation example. On the left is the original code and on the right is its modular product program. The method now takes 4 arguments: two activation variables, and two copies of the original argument. Then, instead of the `if` statement, four activation variables are created, two for the then branch (`t1` and `t2`) and two for the else branch (`f1` and `f2`). As the then branch only contains an assignment, it is duplicated and conditionally executed using the then branch activation variables. The else branch contains a method call and as the transformed methods take activation variables, the transformed call is performed if at least one of the two executions executes the else branch. The call takes the activation variables to only influence the corresponding execution and the returned values are conditionally used to not influence inactive execution.

$$
\begin{aligned}
[[s_1; s_2]]_{p_1,p_2} &= [[s_1]]_{p_1,p_2}; [[s_2]]_{p_1,p_2} \\
[[\texttt{if } (c) \texttt{ \{ } s_t \texttt{ \} else \{ } s_e \texttt{ \}}]]_{p_1,p_2} &= t_1 \texttt{ := } p_1 \texttt{ \&\& } c_1 \\
&\quad\; t_2 \texttt{ := } p_2 \texttt{ \&\& } c_2 \\
&\quad\; f_1 \texttt{ := } p_1 \texttt{ \&\& !} c_1 \\
&\quad\; f_2 \texttt{ := } p_2 \texttt{ \&\& !} c_2 \\
&\quad\; [[s_t]]_{t_1,t_2} \\
&\quad\; [[s_e]]_{f_1,f_2} \\
&\quad\; \text{where } fresh(t_1, t_2, f_1, f_2) \\
[[x \texttt{ := } e]]_{p_1,p_2} &= \texttt{if } (p_1) \texttt{ \{ } x_1 \texttt{ := } e_1 \texttt{ \}} \\
&\quad\; \texttt{if } (p_2) \texttt{ \{ } x_2 \texttt{ := } e_2 \texttt{ \}} \\
[[\texttt{inhale } e]]_{p_1,p_2} &= \texttt{if } (p_1) \texttt{ \{ inhale } e_1\texttt{\}} \\
&\quad\; \texttt{if } (p_2) \texttt{ \{ inhale } e_2\texttt{\}} \\
&\quad\; \text{if } e \text{ is not relational} \\
[[\texttt{inhale } e]]_{p_1,p_2} &= \texttt{inhale } [e]_{p_1,p_2} \\
&\quad\; \text{otherwise} \\
[[\texttt{exhale } e]]_{p_1,p_2} &= \texttt{if } (p_1) \texttt{ \{ exhale } e_1\texttt{\}} \\
&\quad\; \texttt{if } (p_2) \texttt{ \{ exhale } e_2\texttt{\}} \\
&\quad\; \text{if } e \text{ is not relational} \\
[[\texttt{exhale } e]]_{p_1,p_2} &= \texttt{exhale } [e]_{p_1,p_2} \\
&\quad\; \text{otherwise}
\end{aligned}
$$

Figure 2.1: MPP transformation for the main statements used by Prusti where $[e]_{p_1,p_2}$ is the transformation for relational expression.

```
fn foo(secret: i32) {
    if secret > 0 {
        println("{secret}");
    }
}
```

Listing 13: Printing to the standard output is an observable event and so this function leaks information about `secret` even if is it negative as nothing is printed.

values for each function. For example, if the original code contains a call to an abstract function taking a reference as a parameter and we know that at a call site, the arguments of this function are the same in both executions. Then as functions are pure in Viper, the value returned in both executions should be the same and for non-abstract functions, Viper can verify this by unfolding the body of the functions which are the same with the field name changed. However, for abstract functions, Viper cannot unfold them and as both use a different function, Viper will not be able to prove that the returned values are equal.

2.4.1 Non-Interference

Non-interference is the property that no secret information can influence anything observable by an adversary, those observable things can be some value sent through an unsecured channel, or some event such as a message being written on the screen even though the message is not secret. So for example, Listing 13 shows a function that leaks some information about `secret`. Indeed, if `secret` is positive, then something is printed and nothing is printed otherwise. Additionally, the value of `secret` is even printed if it is positive.

For encoding reasons and to prevent underspecification mistakes, all variables are by default at a *high* security level unless explicitly specified as *low* using the `low` specification function. Thus to check that a code does not leak high information into low expressions, we want to check that for any two executions of the code that starts with the same values for their low arguments, their low results must also be the same. If it is not, the difference must come from high arguments thus leaking some high information. This can be encoded as a hyperproperty and checked using MPP. Sometimes, however, this model is too restrictive, for example in an encryption function, the message and the keys are high but the encrypted message can safely be sent through the network and thus needs to be low. To remedy this, users can use the `declassify e` statement which assumes that the given expression is low. Figure 2.2 shows how to encode specification.

$$
\begin{aligned}
[\texttt{low}(e)]_{p_1,p_2} &= p_1 \ \texttt{\&\&} \ p_2 \ \texttt{==>} \ e_1 \ \texttt{==} \ e_2 \\
[\texttt{low\_event}()]_{p_1,p_2} &= p_1 \ \texttt{==} \ p_2 \\
[e]_{p_1,p_2} &= p_1 \ \texttt{==>} \ e_1 \ \texttt{\&\&} \ p_2 \ \texttt{==>} \ e_2 \\
&\quad \text{if } e \text{ is not relational} \\
[e \ \texttt{\&\&} \ e']_{p_1,p_2} &= [e]_{p_1,p_2} \ \texttt{\&\&} \ [e']_{p_1,p_2}
\end{aligned}
$$

Figure 2.2: Encoding of specification expressions.

```
fn foo(b: bool) {
    if b {
        print_int(0);
    } else {
        print_int(0);
    }
}
```

Listing 14: False negative example where `print_int(n: Int)` is a function labeled as low event.

Using MPP, we can also verify that no observable event can leak information. Observable events are specified using the `low_event()` specification function and are events observable by an adversary. Thus we want those events to either happen or not independently of high values. One way to encode this hyperproperty for low events using MPP is to check that the two activation variables are equal so that either both executions reach this point or none. One known limitation of this encoding is that all low events are distinct. Thus if a function is labeled as being a low event then even if this function is called exactly once and with the same argument in all executions, which means that no information is leaked, the encoding will still report that some information is leaked as not all pairs of execution reach the same call-sites. For example, in Listing 14 all execution prints the number `0` regardless of the value in `b` thus it is not leaked but the encoding presented here will wrongly report an information leak.

Chapter 3

Design

3.1 Time Reasoning

To reason about the runtime of functions, we use time credits and time receipts as described in Section 2.3. For simplicity, the runtime model used is that one time-credit is consumed and one time-receipt is produced at the beginning of a function – to model to cost of calling the function – and likewise in each iteration of a loop. This model can be extended to count all kinds of operations – for example, memory accesses and binary operations – requiring the user to give the number of times an operation is used inside a function or loop.

3.1.1 Encoding

As mentioned in Section 2.1.3, resources can be encoded in Viper using permission amounts on abstract predicates. So by defining two abstract predicates `time_credits()` and `time_receipts()` we can specify, for example, that a function requires `3*n+1` time credits to execute by adding the pre-condition `acc(time_credits(), (3*n+1)/1)` and that a function ensures it produces at least `4` time receipts by adding the post-condition `acc(time_receipts(), 4/1)`. To encode the runtime model presented previously, we can add calls to an abstract `tick()` method that consumes a time credit and produces a time receipt at the beginning of methods' and loops' bodies. The definitions of the abstract predicates and the `tick()` abstract method can be found in Listing 15 and the array comparison without early return example from the introduction translated into Viper with the required annotations is showcased in Listing 16.

```
predicate time_credits()
predicate time_receipts()

method tick()
    requires acc(time_credits(), 1 / 1)
    ensures acc(time_receipts(), 1 / 1)
```

Listing 15: Time credits, time receipts, and tick method in Viper.

```
method array_cmp(a1: Seq[Int], a2: Seq[Int]) returns (res: Bool)
    requires |a1| == |a2| // Viper notation for length of sequences
    requires acc(time_credits(), (|a1| + 1) / 1)
    ensures acc(time_receipts(), (|a1| + 1) / 1)
{
    tick()
    var i: Int
    i := 0
    res := true
    while (i < |a1|)
        invariant 0 <= i && i <= |a1|
        invariant acc(time_credits(), (|a1| - i) / 1)
        invariant acc(time_receipts(), (i + 1) / 1)
    {
        tick()
        res := res && a1[i] == a2[i]
        i := i + 1
    }
}
```

Listing 16: Array comparison function without early return and with time reasoning.

```
method fact(n: Int) returns (res: Int)
    requires n > 0 // without this pre-condition Viper would
    // complain that the following access amount might be negative
    requires acc(time_credits(), n / 1)
    ensures acc(time_receipts(), n / 1)
{
    tick()
    // time_credits: n - 1, time_receipts: 1
    if (n > 1) {
        res := fact(n - 1)
        // time_credits: 0, time_receipts: n
        res := res * n
    } else {
        // !(n > 1) && n > 0 ==> n == 1
        // time_credis: 0, time_receipts: 1
        res := 1
    }
}
```

Listing 17: Recursive factorial method.

Another approach to keeping track of time credits and receipts is to have explicit variables and manually increment or decrement them instead of adding a call to `tick()`. However, this approach requires more manual bookkeeping as each time we consume a time credit, we must check that the variable is greater than the amount we want to remove and then remove this amount. Furthermore, method calls require extracting the number of time credits consumed and of time receipt produced by the method from its specifications to update the counters. Whereas using access permission on abstract predictions this can all be handled automatically by the `exhale` and `inhale` statements the method's pre- and post-conditions get translated into.

3.1.2 Recursion

As a call to `tick()` is added at the beginning of each method, the time credits must decrease at least by one at each recursive call. As it cannot get negative, the time credits requirement is an upper bound on the number of recursive calls and guarantee the method termination. Listing 17 shows a recursive factorial method that verifies and Listing 18 showcases an infinite recursive method that verifies without the time reasoning encoding but with the encoding Viper prevents the recursive call.

```
method foo(n: Int) returns (res: Int)
    requires acc(time_credits(), 1 / 1) {
    tick()
    // time credits: 0, time_receipts: 1
    res := foo(n) // fails with insufficient time credits
}
```

Listing 18: Infinite recursion is prevented.

3.1.3 Loops

To describe the number of time credits consumed and time receipts produced by a loop, we can use the loop invariant; it should specify the number of time credits required to terminate the loop and the number of time receipts produced so far in the loop. Listing 19 shows how these loop invariants can be stated. Note that counting the time credits needed to terminate the method and the time receipts produced before the loop would also verify as those are removed from the method's counts on loop entry and are added back on loop exit.

As mentioned in Section 2.1.2, loops do not inherit permissions from their surrounding scopes and so if we add a call to `tick()` at the beginning of a loop without specifying its runtime, Viper would prevent the call as shown in Listing 20.

3.2 Secure Information Flow

To allow verification of hyperproperties such as non-interference, Eilers et al. presented Modular Product Programs and implemented the transformation as an extension to Viper [10]. Nagini can use this extension to verify secure information flow in Python programs [8].

However, the transformation does not handle magic wands because Nagini does not generate Viper code that uses them. However, Prusti uses them for functions that perform reborrowing as the programs must wait until the reborrow expires to gain back the permission of the reborrowed reference. For example, Listing 21 showcases an example of reborrow in Rust. Thus to allow Prusti to work with the MPP transformation, we need to extend it to handle magic wands. Currently, the transformation only handles magic wands that do not contain relational specifications, and as a result, creates a copy of each magic wand for each execution.

```
method do_smth(n: Int) returns (res: Int)
    requires acc(time_credits(), 1 / 1)
    ensures acc(time_receipts(), 1 / 1)

method sum(n: Int) returns (res: Int)
    requires n >= 0
    requires acc(time_credits(), (n + 2) / 1)
    ensures acc(time_receipts(), (n + 2) / 1)
{
    tick()
    // time_credits: n + 1, time_receipts: 1
    var i: Int := 0
    res := 0
    // On loop entry Viper will exhale the loop invariant. Then
    // it will create a new context for the loop body, havocking all
    // variables modified in the loop -- here i and res -- and
    // inhale the loop invariant in this new context.
    // On loop entry, i == 0 so the global context is:
    // time_credits: 1, time_receipts: 1
    while (i < n)
        invariant 0 <= i && i <= n
        // The required time credits to finish the loop and the
        // number of time receipts produced so far in the loop.
        invariant acc(time_credits(), (n - i) / 1)
        invariant acc(time_receipts(), i / 1)
    {
        // Loop's context: 0 <= i && i <= n && i < n
        // time_credits: n - i, time_receipts: i
        tick()
        // time_credits: n - i - 1, time_receipts: i + 1
        res := res + i
        i := i + 1
        // after the update to i, the loop's context is:
        // time_credits: n - i, time_receipts: i
        // which satisfies the loop invariant.
    }
    // On loop exit we have !(i < n) && i <= n ==> i == n and Viper
    // inhales the loop invariant in the global context and so we have
    // time_credits: 1, time_receipts: n + 1
    res := do_smth(res)
}
```

Listing 19: Loop invariant describing its runtime.

```
method sum(n: Int) returns (res: Int)
    requires n >= 0
    requires acc(time_credits(), (n + 1) / 1)
{
    tick()
    var i: Int := 0
    res := 0
    while (i < n)
        invariant 0 <= i && i <= n
    {
        tick() // fails with insufficient time credits
        res := res + i
        i := i + 1
    }
}
```

Listing 20: Calling `tick()` in a loop without invariant fails.

```
struct Point { x: i32, y: i32 }

impl Point {
    fn get_mut_x(&mut self) -> &mut i32 {
        // this function reborrows its argument
       &mut self.x
    }
}

fn foo() {
    let mut p = Point {x: 1, y: 2};
    // a mutable reference is created for p and
    // passed to the function call
    let mut x = p.get_mut_x();
    // we must wait to drop x to regain access to p
    *x = 42;
    // x is not used anymore so it is dropped and
    // we gain back full access to p
    assert!(p.x == 42);
}
```

Listing 21: Reborrowing in Rust.

Chapter 4

Implementation

4.1 Time Reasoning

In Section 3.1, we have described how time reasoning can be encoded in Viper using permission amounts on abstract predicates. In this section, we will present how this encoding has been implemented in Prusti and how to specify functions' runtime in Section 4.1.1. Also, as Prusti does not use Viper's loop encoding we had to adapt the time credits and receipts encoding to work with Prusti's encoding for loops. We will explain it more in Section 4.1.2.

4.1.1 Specification

Prusti does not expose Viper's permission amounts to the user. Instead, to specify the runtime of functions, users have to use the new `time_credits(n: usize)` and `time_receipts(n: usize)` specification functions. During the encoding of Rust programs, Prusti translates them into access predicates for the corresponding abstract predicates as described in Section 3.1.1. Listing 22 shows the annotated version of the array comparison function presented in the introduction.

```
#[requires(a1.len() == a2.len())]
#[requires(time_credits(a1.len() + 1)]
#[ensures(time_receipts(a1.len() + 1)]
fn array_cmp<T: Eq>(a1: &[T], a2: &[T]) -> bool {
    let mut i = 0;
    let mut res = true;
    while i < a1.len() {
        body_invariant!(time_credits(a1.len() - i));
        body_invariant!(time_receipts(i));
        res &= a1[i] == a2[i];
        i += 1;
    }
    return res;
}
```

Listing 22: Equal size arrays comparison function with runtime annotations.

4.1.2 Loops

As Prusti works on the CFG generated by the Rust compiler, it does not use Viper's loops encoding. Note that in contrast to Viper's loop encoding, Prusti's loop encoding does not create a new context for the permission amounts. Thus, the time credits and time receipts seen in the loop are not from the loop invariant but from the whole function, and with Prusti's encoding of loops, each execution will execute at most one execution of the body of the loop and thus only consume one time credit and produce one time receipt. Listing 23 shows a wrong specification that still verifies, and also explains why.

To fix this issue, we have to create new counters that are initialized at zero for each loop in the function so that these counters are only influenced by the loop invariants and bodies. To do so, we have modified the time credits and receipts predicates, as shown in Listing 24, to have an extra integer argument which is the block id of the head of the loop, and -1 for the global function to be sure no conflict is possible. To match this change and to improve error reporting, we no longer call the `tick()` method but instead use `exhale` and `inhale` statements to easily change the id used. Then with this change, the function from Listing 23 will not verify. Indeed, the loop's time credit count will start at zero and as the loop invariant does not specify anything it will stay at zero. Consequently, we will not be able to consume a time credit in the loop body and thus fail the verification. To summarize, Listing 25 shows the updated example from Listing 23. On loop entry, the global function time credits count is decreased by the amounts specified by the loop invariant (line 10). Then `i` and `res` are havocked and the loop's count is initialized using the loop invariant (lines 12-13). In the loop's body,

```
#[requires(time_credits(2))]
fn sum(n: usize) -> usize {


    let mut res = 0;
    let mut i = 0;
    while i < n {

        body_invariant!(true);


        res += i;
        i += 1;


    }
    res
}
```

```
#[requires(time_credits(2))]
fn sum(n: usize) -> usize {
    // time_credits: 2
    tick();
    // time_credits: 1
    let mut res = 0;
    let mut i = 0;
    if i < n {
        exhale true // invariant
        havoc(i, res)
        inhale true // invariant
        tick();
        // time_credits: 0
        res += i;
        i += 1;
        if i < n {
            exhale true // invariant
            inhale false
        }
    }
    res
}
```

Listing 23: On the left, a simple function that sums all integers from `0` to `n`. Its correct runtime is `n+1`, but the specification wrongly says `2`, and a trivial loop invariant is given. Prusti should complain that there are not enough time credits to execute the function but instead, does not report any error. On the right, the transformed program with the time credits count at key instances. Prusti does not report any error because no time credit gets consumed after the count reaches zero.

```
1 predicate time_credits(id: Int)
2 predicate time_receipts(id: Int)
```

Listing 24: Updated time credits and receipts predicates.

we only need to update the local loop's count (line 15) and finally, the loop invariant is checked for looping traces using the loop's time credits count (line 22).

However, this approach is not enough to reason about time receipts because on loop entry the time receipts amount specified in the loop invariant is equal to zero as the loop has not yet produced any. To reason about time receipts, we need to find a way to add to the global time receipts count the amount that the loop has produced on exit. Additionally, with the encoding explained so far, if a loop exits without consuming all the time credits specified in its invariant on entry then it will not gain the remaining credits back. To tackle these problems, instead of only inhaling and exhaling the loop invariant with the local loop's scope id, we inhale and exhale it with all the ids of the enclosing scopes of the loop. This works because if we are in a looping iteration of the loop then the trace will assume false, killing this trace and otherwise we exit the loop with the produced time receipts and the remaining time credits added to the global counts as wanted. Listing 26 shows the same example with the reasoning of time receipts.

4.1.3 Pure Functions

Pure functions in Prusti are functions that can be used both in specification and normal code and thus, like Viper's functions, they cannot have any side effects. As a consequence, when used in normal code, they should consume time credits and produce time receipts as a normal function. However, when used in specification code, the runtime specification should be ignored. We did not have the time to implement this distinction and so currently the time reasoning extension of Prusti does not support pure functions. Listing 27 shows an implementation of a linked list in Rust that uses a pure function – `len` – to specify the runtime of its `sum` function.

4.2 Secure Information Flow

As explained in Section 3.2, Eilers et al. already implemented a Modular Product Programs transformation in Viper. After updating it to handle magic wands, Prusti needs to

```
#[requires(time_credits(n + 1))]
fn sum(n: usize) -> usize {
    // time_credits(-1): n + 1
    exhale acc(time_credits(-1), 1 / 1)
    // time_credits(-1): n
    let mut res = 0;
    let mut i = 0;
    if i < n {
        // loop invariant
        exhale acc(time_credits(-1), (n - i) / 1)
        // i == 0 ==> time_credits(-1): 0
        havoc(i, res)
        inhale acc(time_credits(1), (n - i) / 1)
        // time_credits(1): n - i
        exhale acc(time_credits(1), 1 / 1)
        // time_credits(-1): n - i - 1
        res += i;
        i += 1;
        // after the update to i:
        // time_credits(-1): n - i
        if i < n {
            exhale acc(time_credits(1), (n - i) / 1)
            // the previous exhale statement verifies
            // proving the loop invariant correct
            exhale false
        }
    }
    // the global time_credits count has not been changed
    // in the loop's body so it remains at :
    // time_credits(-1): 0
    res
}
```

Listing 25: Time credit reasoning for loops. The original code from Listing 23 has only been changed to have correct time credit specifications so it is not shown here. Note that the id used as an argument to the `time_credits` predicate is `-1` for the global function count and `1` for the loop count.

```
#[requires(time_credits(n + 1))]
#[ensures(time_receipts(n + 1))]
fn sum(n: usize) -> usize {
    // time_receipts(-1): 0
    inhale acc(time_receipts(-1), 1 / 1)
    // time_receipts(-1): 1
    let mut res = 0;
    let mut i = 0;
    if i < n {
        exhale acc(time_receipts(-1), i / 1)
        // i == 0 ==>
        // time_receipts(-1): 1
        havoc(i, res)
        inhale acc(time_receipts(1), i / 1)
            && acc(time_receipts(-1), i / 1)
        // time_receipts(-1): i + 1
        // time_receipts(1): i
        inhale acc(time_receipts(1), 1 / 1)
            && acc(time_receipts(-1), 1 / 1)
        // time_receipts(-1): i + 2
        // time_receipts(1): i + 1
        res += i;
        i += 1;
        // after the update to i:
        // time_receipts(-1): i + 1
        // time_receipts(1): i
        if i < n {
            exhale acc(time_receipts(1), i / 1)
            // the previous exhale statement verifies
            // proving the loop invariant correct
            exhale false
        }
        // i == n ==>
        // time_receipts(-1): n + 1
    }
    res
    // the post-condition is satisfied!
}
```

Listing 26: Time receipts reasoning for loops. The time credits reasoning has already been shown in Listing 25, so it is not shown here. Note that the id used as an argument to the `time_receipts` predicate is `-1` for the global function count and `1` for the loop count.

```
enum List {
    Nil,
    Cons(i32, Box<List>),
}

impl List {
    #[pure]
    // The runtime of len is its returned value.
    fn len(&self) -> usize {
        match self {
            Nil => 0,
            Cons(_, tail) => tail.len() + 1,
        }
    }

    // In specification code, we are only interested in the
    // returned value pure functions and not their runtime.
    #[requires(time_credits(self.len() + 1))]
    #[ensures(time_receipts(self.len() + 1))]
    fn sum(&self) -> i32 {
        match self {
            Nil => 0,
            Cons(i, tail) => i + tail.sum(),
        }
    }
}
```

Listing 27: Linked list implemented in Rust. The `len` function needs to be pure to be used in the specification of the `sum` function.

call it before calling its verifier backend. However, in contrast to Nagini's SIF extension [8] which can load different Viper backends and instantiate wrappers for the extension's new AST nodes at runtime as Nagini is programmed in Python, Prusti is programmed in Rust and thus needs to know their type at compile time and require the extension files to be present when Prusti is compiled. Consequently, it changes Prusti's deployment procedures.

As explained in Section 2.4, the MPP transformation duplicates each field and thus also each function that uses references. Prusti uses abstract functions in its encoding for array slices. Indeed, Prusti encodes array slice access using an abstract function `lookup(array: Ref, idx: Int)` that receives a reference for the array and an integer for the index. Thus, using the MPP transformation will create a copy of this function for each execution and so even if the array is low and the access index is also low, Viper will not be able to prove that the result is also low. To fix that, we removed the duplication of each field, function, and predicate.

Even if the MPP transformation handles declassify statements using it would require adding it to Prusti's statements and handling it in all stages of Prusti's pipeline. So to avoid it, declassify statements are directly translated into `prusti_assume!(low(expr));` by Prusti and thus the pipeline did not have to be modified.

4.2.1 Loops

Prusti's loop encoding is not compatible with the MPP transformation explained thus far. To see this, let us consider the function `foo` from Section 4.2.1. This function should not verify as if `y1 == 2` for the first execution and `y2 == 4` for the second execution, then at the end of the loop `i1 == 3` and `i2 == 5` and as a result, the precondition for the `print` calls are not verified. However, with the MPP transformation as explained so far, Prusti verifies this code. To understand why, let us consider all the combinations of traces, either both executions loops and so reach the `assume false` statement, either they both exit the loop at the same time and so reach the call sites, either one of the execution loops and the other exits the loop, in this case, the loop invariant will not check that both executions still have the same value of `i` whereas it would have been checked by using Viper's while loop and their MPP transformation. As a workaround for this issue, we automatically add a `low_event` constraint to the loop invariants when it contains SIF specifications which force both executions to leave the loop at the same time, preventing the problem.

```
#[trusted]
#[requires(low(i))]
fn print(i: u32) {}

fn foo(y: u32) {
    let i = 0;
    while i < y + 1 {
        body_invariant!(low(i));
        i += 1;
    }
    print(i);
    print(y);
}
```

Listing 28: The loop invariant should not verify as both executions might exit the loop with different values for `i`.

4.3 Simplifications

One issue related to how Prusti encodes Rust's expression in Viper and which is common to both parts of this project is that Prusti translates implications `P ==> Q` into `!P || Q`. This translation is sound if Q does not contain any permission assertion and if P is side-effect free which is the case in normal Prusti assertions. However, as time reasoning assertions are encoded using the permission mechanism of Viper, it will not accept programs such as `b || acc(time_credits(), 1/1)` generated from `b ==> time_credits(1)`. The MPP transformation also does not support such encoding. As a result, before passing programs to Viper or the MPP transformation, we have to transform disjunctions back into implications.

Another related issue is that Prusti adds some double negations during its encoding. This is sound as in Viper $P \equiv \neg\neg P$ when P does not use access permission predicate. But for time reasoning, Prusti uses them and consequently, Viper rejects the programs in which this happens. To fix this, during the same transformation as for the implications, we also remove double negations.

Chapter 5

Evaluation

5.1 Time Reasoning

We will present how the time reasoning ability combines with other capacities of Prusti using two examples. The first example (Listing 29) showcases that implications can be used to specify different runtimes for functions when different execution paths yield different runtime. Listing 30 showcases how one could use the result of functions to define its runtime.

5.1.1 Non-Linear Runtime

Viper is using real numbers to represent permission amounts in its SMT encoding and SMT solvers are sometimes limited by non-linear arithmetic. As a consequence, Prusti is sometimes not able to prove non-linear runtime as in Listing 31. On the other hand, Prusti is able to prove the exponential runtime of the recursive Fibonacci function from Listing 32.

5.1.2 Limitations of the Runtime Model

The runtime model for this project is not precise enough to guarantee the absence of timing attacks. For example, the function from Listing 33 has a runtime of `4000 * loop_count + 1` for its upper and lower bounds so based on the runtime model,

```rust
enum OptionI32 {
    Some(i32),
    None,
}

#[requires(time_credits(1))]
#[ensures(time_receipts(1))]
fn double_int(i: i32) -> i32 {
    2 * i
}

impl OptionI32 {
    #[requires(matches!(self, OptionI32::Some(_)) ==> time_credits(2))]
    #[requires(matches!(self, OptionI32::None) ==> time_credits(1))]
    #[ensures(matches!(self, OptionI32::Some(_)) ==> time_receipts(2))]
    #[ensures(matches!(self, OptionI32::None) ==> time_receipts(1))]
    fn double(&self) -> i32 {
        match self {
            OptionI32::Some(i) => double_int(*i),
            OptionI32::None => 0,
        }
    }
}

```

Listing 29: Time reasoning with enums, pattern matching, and implications.

```rust
#[requires(time_credits(bound + 1))]
#[ensures(time_receipts(result.0 + 1))]
#[ensures(a.len() >= bound ==> result.0 == bound)]
#[ensures(a.len() <= bound ==> result.0 == a.len())]
fn bounded_sum(a: &[usize], bound: usize) -> (usize, usize) {
    let mut i = 0;
    let mut sum = 0;
    while i < a.len() && i < bound {
        body_invariant!(time_credits(bound - i));
        body_invariant!(time_receipts(i));
        sum += a[i];
        i += 1;
    }
    (i, sum)
}

#[requires(time_credits(a.len() + 2))]
#[ensures(time_receipts(a.len() + 2))]
fn sum(a: &[usize]) -> usize {
    bounded_sum(a, a.len()).1
}

#[requires(a.len() >= 1)]
#[requires(time_credits(3))]
#[ensures(time_receipts(3))]
fn first(a: &[usize]) -> usize {
    bounded_sum(a, 1).1
}

#[requires(time_credits(9))]
#[ensures(time_receipts(9))]
fn main() {
    let a = [1, 2, 3];
    let _s = sum(&a);
    let _f = first(&a);
}
```

Listing 30: Using the resulting value of functions to define the exact runtime of functions.

```
#[requires(time_credits((n + 1) * n + 1))]
#[ensures(time_receipts((n + 1) * n + 1))]
fn two_level_loop(n: usize) -> usize {
    let mut i = 0;
    let mut res = 0;
    while i < n {
        // body_invariant!(i < n);
        body_invariant!(time_credits((n + 1) * (n - i)));
        body_invariant!(time_receipts((n + 1) * i));
        let mut j = 0;
        let mut inner_res = 0;
        while j < n {
            // Prusti is not able to verify the
            // following invariant on loop entry
            body_invariant!(time_credits(n - j));
            body_invariant!(time_receipts(j));
            res += j;
            j += 1;
        }
        res += inner_res;
        i += 1;
    }
    res
}
```

Listing 31: Quadratic function where Prusti is not able to prove the runtime.

```rust
#[requires(time_credits(2_usize.pow(n as u32)))]
fn fib_rec(n: usize) -> usize {
    if n <= 1 {
        1
    } else {
        fib_rec(n - 1) + fib_rec(n - 2)
    }
}

// define specifications for external functions
#[extern_spec]
impl usize {
    #[pure]
    #[ensures(self.pow(n) * self == self.pow(n + 1))]
    #[ensures(self >= 1 ==> result >= 1)]
    #[ensures(self >= 1 ==> self.pow(0) == 1)]
    fn pow(self, n: u32) -> usize;
}

#[requires(time_credits(50))]
fn use_fib_rec() {
    // consumes 32 credits
    fib_rec(5);
}
```

Listing 32: Recursive Fibonacci function with exponential runtime.

```
fn fancy_sum(secret: bool, loop_count: u32,
             a1: [u64; 4000], a2: [u64; 4000],
) -> u64 {
    let mut l = 0;
    let mut res = 0;
    while l < loop_count {
        let mut i = 0;
        while i < 4000 {
            res += a1[i];
            if secret {
                res += a1[i];
            } else {
                res += a2[i];
            }
            i += 1;
        }
        l += 1;
    }
    res
}
```

Listing 33: Function used to increase the data cache pressure.

the runtime does not depend on `secret`. However, depending on the value of the secret, the data cache utilization will not be the same. Indeed, for `secret == true` only `a1` is accessed and for `secret == false` both `a1` and `a2` are accessed which increases the data cache pressure and indeed we can measure a few percent runtime difference depending on the value of `secret`. Additionally, the runtime model does not take into account how the compiler might optimize the program. Indeed, using the same program and optimizing it yields a runtime difference of up to 50% depending on the value of `secret`.

5.2 Secure Information Flow

The SIF extension presented in this project, works with function calls, control-flows statements such as if-then-else, and implications as shown in Listing 34.

```
#[requires(low(l))]
#[ensures(b ==> low(result))]
fn choose(b: bool, l: i32, h: i32) -> i32 {
    if b {
        l
    } else {
        h
    }
}

#[ensures(low(i) ==> low(result))]
fn add_one(i: i32) -> i32 {
    i + 1
}

fn main() {
    let h = 42;
    let l = produce_low();

    prusti_assert!(low(choose(true, l, h)));
    prusti_assert!(low(add_one(l)));

    // the following assertions fail
    prusti_assert!(low(choose(false, l, h)));
    prusti_assert!(low(add_one(h)));
}

#[trusted]
#[ensures(low(result))]
fn produce_low() -> i32 { todo!() }
```

Listing 34: Function calls, control-flow, and implication work with the SIF extension.

```
struct Wrapper(i32);

impl Wrapper {
    #[pure]
    fn get(&self) -> i32 {
        self.0
    }
    // The following specification states that if the value referenced
    // by the result of this function just before the caller
    // drops the reference is low, then the value returned by calling
    // `self.get()` is also low.
    #[after_expiry(low(before_expiry(*result)) ==> low(self.get()))]
    fn get_mut(&mut self) -> &mut i32 {
        &mut self.0
    }
}
```

Listing 35: This code will not verify as the MPP transformation for the magic wand does not work with relational expressions.

5.2.1 Reborrowing

The updated MPP transformation for magic wands only works for non-relational expressions so the code in Listing 35 will not work. However, the transformation works for magic wands containing non-relational expressions, so Prusti can verify the code in Listing 36.

5.2.2 Arrays and Forall

There are multiple ways to reason about the lowness of arrays. Firstly, we can specify that the reference to an array is low as in Listing 37 which makes all the elements of the array also low as both executions use the same reference. Secondly, if we do not have a low reference of an array, we can use forall expressions to specify that all the elements of an array are low as in Listing 38. Finally, we can also use forall expressions in loop invariants to fill an array with low values as in Listing 39.

```
struct Wrapper(i32);

impl Wrapper {
    #[pure]
    fn get(&self) -> i32 {
        self.0
    }
    // The following specification states that the value referenced by
    // the returned reference just before the caller drops it, is
    // the value returned by calling `self.get()`.
    #[after_expiry(before_expire(*result) == self.get())]
    fn get_mut(&mut self) -> &mut i32 {
        &mut self.0
    }
}

fn main() {
    let mut w = Wrapper(0);
    *w.get_mut() = produce_low();
    prusti_assert!(low(w.get()));
}
```

Listing 36: The magic wand used with non-relational expression works correctly.

```
#[requires(low(ts))]
#[ensures(low(result))]
fn sum_array_low(ts: &[i32]) -> i32 {
    let mut i = 0;
    let mut res = 0;
    while i < ts.len() {
        body_invariant!(low(i));
        body_invariant!(low(res));
        res += ts[i];
        i += 1;
    }
    res
}
```

Listing 37: Having a low reference to an array makes all elements of this array low.

```
#[requires(forall(|i: usize| i < ts.len() ==> low(ts[i])))]
#[ensures(low(result))]
fn sum_array_low2(ts: &[i32]) -> i32 {
    let mut i = 0;
    let mut res = 0;
    while i < ts.len() {
        body_invariant!(low(i));
        body_invariant!(low(res));
        res += ts[i];
        i += 1;
    }
    res
}
```

Listing 38: Using a forall expression to specify the lowness of every element of an array allows Prusti to reason about the lowness of their sum.

```
#[requires(low(array.len()))]
#[ensures(forall(|i: usize| i < array.len() ==> low(array[i])))]
fn fill_with_low(array: &mut [i32]) {
    let mut i = 0;
    while i < array.len() {
        body_invariant!(low(i));
        body_invariant!(forall(|j: usize| j < i ==> low(array[j])));
        array[i] = produce_low();
        i += 1;
    }
}

#[trusted]
#[ensures(low(result))]
fn produce_low() -> i32 { todo!() }
```

Listing 39: Filling the given mutable reference of an array with low values. Using low expression inside a forall expression in the loop invariant allows Prusti to prove the post-condition.

Chapter 6

Related Work

6.1 Time reasoning

In a previous Master's thesis based on Prusti, Lowis [11] built a similar time-reasoning extension that uses time credits. Lowis's extension is designed to reason about the upper bound of amortized and asymptotic runtimes in programs that do not use loops.

Mével et al. [16] built an extension of Iris [14] to allow reasoning about time credits and time receipts. Their implementation allows reasoning about upper and lower bounds using time credits and time receipts by adding calls to a *tick* function similar to this work. In addition, it can reason about amortized runtime.

Hoffmann et al. [13] used time credits to guarantee lock freedom. They call them tokens and are used to give an upper bound to the number of loop iterations before a lock is freed.

Charguéraud and Pottier [6] have used time credits in separation logic to verify the correctness and the amortized complexity of a union-find implication in OCaml.

6.2 Secure Information Flow

In this project, we have used Modular Product Programs [10] to prove Secure Information Flow in Rust programs. This is an adaptation of Eilers et al.'s work for the Nagini Python verifer [8].

Crichton et al. used the ownership type system of Rust to develop *Flowistry* [7] a system for analyzing information flow in Rust. They used their system to implement a program slicer in a VSCode extension, highlighting lines of code that influence a selected variable and fading out irrelevant ones. Additionally, they also used it to verify secure information flow using traits to specify security levels.

Myers developed *JFlow* [18], an extension of the Java language with security annotations that can automatically be translated into normal Java code after verification. It provides some automatic labeling mechanisms and a runtime labeling system when the static analysis is too restrictive.

Pottier and Simonet [19] presented a type-based information flow analysis for OCaml and provided formal proof of correctness.

Austin and Cornac created an information flow analysis [3] for JavaScript that is dynamic, allowing it to report fewer false negatives, and lightweight.

More closely related to this work, Barthe et al. showed how self-composition [5] of programs can be used to check for secure information flow. And they also showed how hyperproperties such as non-interference can be verified using product programs [4].

Chapter 7

Conclusion

In this project, we have extended the capabilities of Prusti, a formal verifier for Rust programs, to reason about the runtime of Rust functions and also to check for non-interference. These two functionalities are compatible with nearly all features of Prusti. Thus, users can use them in combination with other features of Prusti and prove other properties at the same time as verifying the runtime of functions or checking for secure information flow.

7.1 Future work

7.1.1 Time reasoning in pure functions

Currently, pure functions cannot be written with the time reasoning extension. To add them to Prusti, we should ignore runtime specifications of pure functions when they are used in specifications and use those runtime specifications when the pure functions are called in normal code.

7.1.2 Extend Modular Product Programs for Magic Wands

The Modular Product Programs transformation for the magic wand used in this project does not allow relational specification in the subexpression of the wand as in Listing 35. Thus, a natural extension of this project is to remove this constraint.

7.1.3 Combining Secure Information Flow and Time Reasoning

The time reasoning encoding and the MPP transformation used in this project are not yet compatible. Indeed, as predicates are not duplicated between the two executions in the MPP transformation, the permission amounts for the `time_credits` and `time_receipts` predicates will be shared between the two executions. To remedy, this we could introduce an additional argument to the time credits and receipts predicates that would be different for each execution.

Bibliography

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. "Leveraging rust types for modular specification and verification". en. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3360573. URL: https://dl.acm.org/doi/10.1145/3360573 (visited on 10/04/2022).

[2] Robert Atkey. "Amortised Resource Analysis with Separation Logic". In: *Programming Languages and Systems*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 85–103. ISBN: 978-3-642-11957-6.

[3] Thomas H. Austin and Cormac Flanagan. "Efficient Purely-Dynamic Information Flow Analysis". In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. PLAS '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 113–124. ISBN: 9781605586458. DOI: 10.1145/1554339.1554353. URL: https://doi.org/10.1145/1554339.1554353.

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational Verification Using Product Programs". In: *World Congress on Formal Methods*. 2011.

[5] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure information flow by self-composition". en. In: *Mathematical Structures in Computer Science* 21.6 (Dec. 2011), pp. 1207–1252. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129511000193. URL: https://www.cambridge.org/core/product/identifier/S0960129511000193/type/journal_article (visited on 10/19/2022).

[6] Arthur Charguéraud and François Pottier. "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits". en. In: *Journal of Automated Reasoning* 62.3 (Mar. 2019), pp. 331–365. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-017-9431-7. URL: http://link.springer.com/10.1007/s10817-017-9431-7 (visited on 03/24/2023).

[7] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". en. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implemen-*

tation. arXiv:2111.13662 [cs]. June 2022, pp. 1–14. DOI: 10.1145/3519939.3523445. URL: http://arxiv.org/abs/2111.13662 (visited on 11/12/2022).

[8] Marco Eilers, Severin Meier, and Peter Müller. "Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security". en. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 718–741. ISBN: 978-3-030-81684-1. DOI: 10.1007/978-3-030-81685-8_34. URL: https://link.springer.com/10.1007/978-3-030-81685-8_34 (visited on 09/28/2022).

[9] Marco Eilers and Peter Müller. "Nagini: A Static Verifier for Python". en. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 596–603. ISBN: 978-3-319-96144-6. DOI: 10.1007/978-3-319-96145-3_33. URL: http://link.springer.com/10.1007/978-3-319-96145-3_33 (visited on 03/02/2023).

[10] Marco Eilers, Peter Müller, and Samuel Hitz. "Modular Product Programs". en. In: *ACM Transactions on Programming Languages and Systems* 42.1 (Mar. 2020), pp. 1–37. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3324783. URL: https://dl.acm.org/doi/10.1145/3324783 (visited on 10/18/2022).

[11] Lowis Engel. "Reasoning about Complexities in a Rust Verifier". en. In: (), p. 79.

[12] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37036-6.

[13] Jan Hoffmann, Michael Marmar, and Zhong Shao. "Quantitative Reasoning for Proving Lock-Freedom". en. In: *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. New Orleans, LA, USA: IEEE, June 2013, pp. 124–133. ISBN: 978-1-4799-0413-6. DOI: 10.1109/LICS.2013.18. URL: http://ieeexplore.ieee.org/document/6571544/ (visited on 03/24/2023).

[14] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 637–650. ISSN: 0362-1340. DOI: 10.1145/2775051.2676980. URL: https://doi.org/10.1145/2775051.2676980.

[15] K. Rustan M. Leino. "This is Boogie 2". June 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[16] Glen Mével, Jacques-Henri Jourdan, and François Pottier. “Time Credits and Time Receipts in Iris”. en. In: *Programming Languages and Systems*. Ed. by Luís Caires. Vol. 11423. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 3–29. ISBN: 978-3-030-17184-1. DOI: 10.1007/978-3-030-17184-1_1. URL: http://link.springer.com/10.1007/978-3-030-17184-1_1 (visited on 10/18/2022).

[17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_2. URL: http://link.springer.com/10.1007/978-3-662-49122-5_2 (visited on 10/18/2022).

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

[19] François Pottier and Vincent Simonet. “Information Flow Inference for ML”. In: *ACM Trans. Program. Lang. Syst.* 25.1 (Jan. 2003), pp. 117–158. ISSN: 0164-0925. DOI: 10.1145/596980.596983. URL: https://doi.org/10.1145/596980.596983.

[20] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. “Gobra: Modular Specification and Verification of Go Programs”. In: *Computer Aided Verification (CAV)*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer International Publishing, 2021, pp. 367–379. URL: https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17.