Verification of Object Invariants in The Presence of Unverified Code

by

Frédéric Necker

May 2024

Advisors: Dr. Marco Eilers, Prof. Dr. Peter Müller

Programming Methodology Group

Department of Computer Science, ETH Zurich

Acknowledgments

I'd like to thank Marco Eilers. Without his support, advice, and many hours spent working with me over the course of this thesis, this thesis would not have been possible. I appreciate the length he went to to help me on this project, the trust he put in me and the reassurance he gave me at moments where I thought this would not be possible. I thank also him for helping me for the implementation of the plugin.

I would also like to thank Prof. Dr. Peter Müller for giving me the opportunity to work with his group and for the good facilities he has provided me with as well as letting me work with such a powerful tool that is Viper.

Finally I'd like to thank my family and friends, this project would not have been born without their support during my studies and during this thesis.

Table of Contents

Abstract

Deductive program verification is a field whose goal is to construct mathematical proofs of the correctness of programs with respect to a given specification. An assumption often made in the field is that when a given program is verified, all the code it interacts with is also verified. Verification techniques that rely on this assumption become unsound if this assumption is broken.

Thus, to remedy that problem, we develop in this thesis a verification technique that remains sound in presence of unverified code. The properties we verify include methods specifications, and class invariants. They are able to be proven even under interference from unverified code. We then apply the verification technique by creating an encoding from the specified Java subset to the Viper language. Finally, we implement the encoding into a Viper [1] plugin which we evaluate using a number of examples.

Chapter 1

Introduction

Deductive program verification is a field whose goal is to construct mathematical proofs of the correctness of programs with respect to a given specification. Obstacles to that goal include concurrency and heap-based memory management. In a multi-threaded environment, two threads trying to write to the same memory location at the same time could cause very complex behaviors that are nearly impossible to analyze, especially if the analysis is supposed to be modular, with modular meaning that the analysis only analyse parts of the code-base at a time. Some of the tools that try to deal with those obstacles are permission logics like Separation Logic[2] and a variation of it named Implicit Dynamic Frames [3]. By using permissions on memory location, the Implicit Dynamic Frames logic is capable of enforcing that only one thread at a time can overwrite a memory location, thus making sure that the behavior of writing on a location in memory much more predictable. This is crucial to verifying programs modularly because then the effect of a write or read in a memory location can be determined without having to take into account the behavior of every other thread in our environment. Many other useful properties arise from such a system. And thus, usage of the above-mentioned system has grown in the deductive program verification domain over the past 20 years. Notably, the Viper tool set [1] uses Implicit Dynamic Frames

However, permission logics depends on a very strong assumption: the assumption that a thread will only write to a memory location if it has permission to do so. This is fairly simple to enforce in verified code, as you simply check at every write statement if permission

to do so is possessed. But unverified code gives no such guarantee.

Properties proved about some piece of code using a permission logic may not actually hold if this code runs in the presence of other, unverified code as said unverified code could interfere in the execution of the verified code.

Due to access control mechanisms in languages, classes can be written in such a way that strong properties hold for them no matter what environment they are executed in, i.e., even in the presence of unverified code.

For a fairly trivial example, say we wanted a class `EvenHolder` with a field `f` that we would want to always be even.

```
1  class EvenHolder{
2      private int f;
3      public EvenHolder()
4          requires
5      {
6          this.f = 0;
7      }
8      private void setf(int i)
9          requires acc(this.f) && i%2 == 0
10         ensures acc(this.f) && this.f == i
11     {
12         this.f = i;
13     }
14
15 }
```

Without access modifiers, an unverified client code could either change the value of the field foo by assigning to it or call the method `setf()` with an argument different from 0 since it can ignore specification. But since both field `f` and the method bad are private, the only access to the field and methods possible is by verified code.

This touches on an other challenge. With unverified code interfering, many properties

about a class can only be maintained if all methods of said class respect said property. Thus it is needed to verify *Invariants*. *Invariants* are a set of properties about a class that are respected by all its methods and thus that are safe to assume. By verifying invariants it is possible to use them to prove some more advanced method-specific properties.

Of course, all of this is now assuming that the unverified and potentially malicious code is respecting the access modifiers, an assumption we were not making before. This is fine because the assumption we are replacing is stronger than the newer one. Effectively, we are replacing “all code running in our environment obeys language rules and is verified” by “all code running in our environment obeys language rules” and "all code we verify is verified." We explain the details of the new weaker assumptions that we make, as well as the capability of unverified code in chapter 3

Knowing all this, in this thesis, we aim to design a more robust verification system that should be sound even in the presence of unverified and potentially malicious code. This verification system should both be able to prove both method specifications and class *Invarians*. We then aim to implement a Viper plugin that will apply this tool to verify programs in a concrete manner.

1.1 Outline

We first discuss in Chapter 2 the background knowledge one would need to read this thesis. In Chapter 3, we present the environment we designed our thesis around, what we tried to achieve, the challenges of what we tried to achieve as well as the design of the solutions we created to face those challenges. Chapter 4 describes how, once we had a completed design, we turned our theoretical design into an implemented concrete tool. In Chapter 5 we evaluate how successful the tool is at both being a useful and sound tool. We conclude the thesis in Chapter 6 by restating what we have achieved.

Chapter 2

Background

2.1 Viper

Viper [1] is a tool-chain for deductive program verification. It consists of an intermediate language also named Viper and back-ends for that language.

The Viper language, as per the original Viper paper [1], "is a sequential, imperative, object-based intermediate language. Programs in this language consist of a sequence of global declarations for fields, methods, functions, predicates, and custom domains". In this thesis, fields, methods, and predicates are mainly used, so we will detail them more.

Viper fields are somewhat similar to Java fields; they are declared with a type and are always accessed from a receiver. But, contrary to Java, every object has every field. There is no notion of class, and thus no notion of a field belonging to a class.

Viper methods are also somewhat similar to Java methods. They have parameters, can optionally return values, including references, and can have a body . They differ in that they do not have a receiver and do not belong to a class. Additionally, methods can be specified with pre- and post-conditions to specify their intended behaviour. Here is an example:

```
field f1: Int
field f2: Int

method foo(bar: Ref)
    requires acc(bar.f1,write) && acc(bar.f2,wildcard)
    ensures acc(bar.f1) && bar.f1 == 0
{
    bar.f1 := 0
}
```

Listing 2.1: an exemple field and method

Here we have both a field declaration for the fields `f1` ,and `f2` and a method declaration for the method `foo` which takes a reference `bar` as an argument. The method has both pre- and post-conditions as well as a body. The pre- and post-conditions indicate the method's specifications, while the body is the actual implementation of the method. It can be noted that one could declare the same function without a body, as Viper only looks at the specification and signature of a method, not its body, when it is called. Such abstract methods are convenient if the declared method has already been verified elsewhere and does not need to be verified again or if the user of the verifier has only access to the specification of a yet to be implemented method.

In 2.1 there is the keyword `acc()` . It represent a *Viper permission*. *Viper permissions* are themselves based on Implicit Dynamic Frames [3] .Permissions are, as their name indicate, a way to regulate heap memory accesses and specify what memory locations a method touches. As per the original Viper paper [1] " A method or loop body may access the location only if the appropriate permission is held at the corresponding program point." Permissions are held by method executions and or loop body and are generally transferred during the transition between the execution of one holder to the next, this could be for exemple when a method returns and give out its preconditions to its caller, or when a loop body is entered. Permissions are not necessarily held whole, one can hold a positive fraction

of a permission. Viper enforces that the sum of all fraction of the permissions to a single heap location never exceeds 1 which leads to some interesting properties. Say we have, as in the example above, an object `bar` with a field `f1`. Holding the whole permission to that field, noted `acc(bar.f1, write)`, `acc(bar.f1, 1/1)` or shortened as `acc(bar.f1)`, allows one to write in the field `f1` of `bar` and be sure that nobody else will be writing or reading to the field `f1` of `bar` at the same time because anyone else holding any non-zero amount of permission would break the above mentioned enforced rule of sum not exceeding 1. Additionally, holding a non-zero fraction of the permission to the field `f1` of `bar`, noted as `acc(bar1.f1, a/b)`, with `a` and `b` being non zero integers and `b` being greater or equal than `a`, lets its holder read the field and give the guarantee that as long as the holder keep a non-zero amount of permission, content of the field will not be changed by another method and or thread.

Finally, there is a special permission amount that can be specified, the wildcard amount. A wildcard amount of permission,noted `acc(bar.f2,wildcard)` indicates holding a non-specified non-zero amount of permission. This lets Viper decide how much permission should be transferred to method and/or loop body that is specified with it. It is useful when one wants to read a field but does not want to specify manually the amount of permission the method needs.

To see them in action here is an example:

```
field fex: Int

method good1(bar: Ref)
    requires acc(bar.fex, write)
{
    var x : Int
    x := 1
    bar.fex := x
}

```

```
11 method good2(bar: Ref)
12     requires acc(bar.fex, wildcard)
13 {
14     var x : Int
15     x := bar.fex
16 }
17 method bad1(bar: Ref)
18     requires requires acc(bar.fex, 1/2)
19 {
20     var x : Int
21     x := 1
22     bar.fex := x
23 }
24 method bad2(bar: Ref)
25     requires true
26 {
27     var x : Int
28     x := bar.fex
29 }
```

Listing 2.2: an exemple field and method

`good1` and `good2` can be successfully verified because the correct permissions are held when one tries to write or read to `bar.fex`, notably in `good2` a wildcard permission is used. `bad1` and `bad2` on the other hand are not verified because the method tries to write with less than the whole permission in `bad1` and with no permissions at all in `bad2`.

Viper predicates, again, as per the original paper, "can be used both to abstract over concrete assertions and to write recursive specifications of heap data structures." This description is not overly clear, so let us look again at an example:

```
field f: Int

predicate foo(bar: Ref)
{
    acc(bar.f) && bar.f == 0
}
```

Here, the predicate `foo` abstracts the fact that the field `f` of `bar` is equal to zero. Practically, that means that if one knows the predicate to be true for a reference, then one also knows that the field `f` of that same reference is equal to 0. One can, using the `fold` and `unfold` instructions, to go from one to the other. For such a trivial predicate, this is fairly useless. But a predicate can be recursive and be used in very powerful ways to specify data structures, among other things. As our work does not use this aspect of predicates, we will not go into further details. But the reader is encouraged to read the original Viper[1] paper, which goes into greater detail on what can be done with them. Another use of predicates is that they, similarly to methods, can be specified without a body. This is useful in cases where one wishes to represent a more abstract property that could not necessarily be represented as a logical expression in a simple manner. We use this to our advantage to represent very important abstract concepts later in this thesis.

Finally, the *Viper back-ends* are the tools one can use to work with the language. It includes a type-checker for the language as well as two verifiers, which can both be used to deductively verify that a Viper program fulfills its own specification. This includes that every asserted expression must be true, no matter the execution path the program takes. As long as the pre-conditions of a method hold when the method is called, then the post-conditions of that method will hold when it returns. And that for every field read and field write of the program, the correct permissions are held. This also includes verifying the presence of permissions when they are needed and much more. The verifier work by transforming the Viper programs into an equivalent SMT problem that is then sent to an SMT solver to solve. How exactly the verifier encodes the program as an SMT program depends on which

back-end is selected.

Chapter 3

Design

3.1 Target Language

The language we picked as a target for verification is a subset of Java we created. The target language need to have such features as classes, fields, methods, variables, access modifiers and branching. Middleweight Java [4] filled all of the needs above with the exception of access modifiers. And so this thesis uses an modified version of Middleweight Java

Its syntax is described in Figure 3.1. Apart from standard Java features and syntax, the target language allow for specifications. Classes have *invariants* and a list of *modifiable* fields, while methods have pre- and post-conditions. The *invariants* and list of *modifiable* fields will be detailed in Section3.5.

(*Programs*)	*Prog*	::=	$cd_1 ... cd_n$
(*Class definition*)	*cd*	::=	`class` C `{` $inv\ mod\ fd_1 ... fd_k\ \ cnd\ \ md_1 ... md_n$ `}`
(*Modifiable list*)	*mod*	::=	`modifiable := {` $fieldname_1, ..., fieldname_k$ `}`
(*Field definition*)	*fd*	::=	$accmod\ C\ fname$
(*Constructor definition*)	*cnd*	::=	$accmod\ C$ `(` $C_1\ arg_1, ..., C_n\ arg_n$ `)` $spec$ `{` $s_1 ... s_n$ `}`
(*Method definition*)	*md*	::=	$accmod\ C\ mname$ `(` $C_1\ arg_1, ..., C_n\ arg_n$ `)` $spec$ `{` $s_1 ... s_n$ `}`
(*Statements*)	*s*	::=	$(ifstmt \mid fieldacc$ `=` $x_2 \mid C\ x \mid\ x_1$ `=` $exp \mid x_3 = mcall \mid leakstmt \mid$ `return` $x);$
(*If statements*)	*ifstmt*	::=	`if(` x `)` `{` $s_1 ... s_k$ `}` `else {` $s_{k+1} ... s_n$ `}`
(*Leak statements*)	*leakstmt*	::=	`leak` x
(*Method call*)	*mcall*	::=	$x.mname$ `(` $x_1, ..., x_k$ `)` | `new` C `(` $x_1, ..., x_n$ `)`
(*Specification*)	*spec*	::=	`requires` `assert` `ensures` $assertion$
(*Invariant*)	*inv*	::=	`invariant :=` $assertion$
(*Access modifier*)	*accmod*	::=	`private` | `public`
(*Field access*)	*fieldacc*	::=	$x.fieldname$
(*Assertion*)	*assertion*	::=	$(assertion\ op\ assertion) \mid sing$
(*Operation*)	*op*	::=	`+` | `-` | `|` | `&` | `*` | `/` | `%` | `!=` | `==` | `>` | `>=`
(*Singleton*)	*sing*	::=	$exp \mid spex$
(*Expression*)	*exp*	::=	$(exp\ op\ exp) \mid fieldacc \mid x \mid$ `true` | `false` | $intLit$
(*Specification expression*)	*spex*	::=	`hidden(` $x, perm$ `)` | `leakable(` $x, perm$ `)` | `acc(` $x, perm$ `)`
(*Permission amount*)	*perm*	::=	`write` | `none` | `wildcard` | $frac$
(*Fraction*)	*frac*	::=	$intLit$ `/` $intLit$

Figure 3.1: Target language syntax

In this syntax, C can range over all class names. For example, that means that for a class declared with `class foo` the constructor declaration will have to start with *accmod foo*. All mention to x on the other hand are local variable accesses. As for *fieldname*, they refer to field names and can range over all declared field names in the class. With, of course, the exception of when they are found in field declarations where they can be any valid Java identifier. Finally, *intLit* refers to int literals, that is to say integers like `1`, `500` and `42`.

A noticeable fact is that we have removed some options offered in Middleweight Java.

For example, `a.f1 = a.f2` is no longer possible despite it being possible in Middleweight Java because neither `a.f1` nor `a.f2` are variables. `a.f1.f2 = x` is also no longer possible because `a.f1` is not itself a variable. In general we only allow one expression in each assignment statement to be non-variables. This restriction does not hamper the expressivity of our target language as one can simply split a desired statement that would be impossible in our target language but possible in Middleweight Java into multiple local variable assignments to create statements that result in the same behaviour and that are valid in our target language. For example, our above `a.f1 = a.f2` would be turned into `var1 = a.f2` where var1 is a fresh variable and `a.f1 = var1` while `a.f1.f2 = x` would become `var1 = a.f1` and `var1.f2 = x`.

The presented target language does not support inheritance. However we do intend to support it Adding inheritance significantly raises the power of the unverified code. As such, to focus on the core aspects of this thesis, we have decided to first verify a language without inheritance, and then re-add it back once the initial concepts are detailed. Thus, inheritance is treated in Section 3.9

3.2 Assumptions and unverified code

As mentioned in the introduction, the goal of this work is to verify properties under a new set of assumptions. The main change in environment we make is that there is now unverified code running alongside the code we verify. If in our new assumptions the code could ignore language rules and directly affect memory C-style, it would be very difficult to prove anything at all. From that comes the need to restrict what the unverified code can do. Thus, we assume the following properties on unverified code:

The unverified code is written in our target language.

The unverified code is type-checked and compiles.

The receiver of field accesses to a private field has to be syntactically "this" for both verified

and unverified code.

The unverified code can run concurrently with verified code in one or more threads. Access modifiers are enforced and thus,the code cannot directly write or read private fields of verified classes.

We still keep the following assumptions:

Verified code is also written in our target language

Verified code is type-checked, compiles and respects access modifiers

Verified code is verified, and thus respects its specification.

Verified code can interact with unverified code through unverified classes, which are classes whose signatures are known but whose specifications are not

In practice, the last assumption on unverified code means that we assume no reflection is allowed to happen. This can, for example, be enforced through the Security Manager[5].

Those assumptions could be true in, for example, a scenario where one would like to verify some client code that uses an unverified library. The library is written in Java and has been compiled to Java byte-code that is thus type-checked but the library has not been annotated with specifications and thus,it is impossible to verify said library. Another possible scenario would be verifying a general library that is expected to receive plugins to interact with. At the time of writing the library the plugins have not been written and what could one want to write in a plugin is not yet known. The plugin will be compiled with general library so the assumptions can be enforced.

For nomenclature, we will call classes that are not specified as unverified classes. We will call instances of unverified classes unverified objects.

Those assumptions notably do not include that the unverified code will not quit exceptionally by, for example, creating too many objects and thus taking too much memory or by dividing a number by 0. The unverified code can crash at any time and thus,providing guarantees for crash-free executions makes little sense.

What those assumptions guarantee however is that unverified code cannot access objects

that it was not passed, either directly through assigning to an unverified object's field or indirectly through the return of a method the unverified code. If an object is ´´hidden" that way then unverified code cannot interact with it. That means that if one can prove that an object was not passed to unverified code in any way, then one can use that object in an almost identical way as it could be in a setting where all code is verified. One can access its public fields without assuming that the content of such a field will be changed.

It also means that, even for objects that have been passed to unverified code, unverified code will never be able to directly call a private method of an object of a verified class. Unverified code can only ever call public methods of those objects and since those methods have been verified we can be certain that all private methods will be called in ways that respect their specifications and that their return values will never be directly passed to unverified code before going through verified code first. And finally, it means that the only way to affect private fields will be through verified methods. In short, it allow us to reason about private fields and methods because we can trust that all interactions with them will pass through verified code.

3.3 General Design

As mentioned above, objects that are never passed to unverified code have stronger guarantees in behaviour. Thus,it is useful to know which references are also held by unknown code and which are not. In this thesis we will work with a model where we separate references that we are sure are not held by the unverified code, calling them *hidden* references, and references where unverified code could possibly hold them, calling them *leakable* references.

But even if we pass a reference to unverified code, we still want to be able to use the object afterwards. We will have weaker guarantees about the object but we want to be able to guarantee a minimum set of properties that holds for the object even when the unverified code holds a reference to it. For those reasons, we want to specify *class invariants*. A *class*

invariant is an assertion about a class that is established by the class before the object is ever passed to unverified code and remains true at all times after it has been passed. They can then be used by the class to help prove public methods specifications. *Invariants* are explained in more detail in Section 3.5.

Thus,the objective will be to verify that the following properties hold: When an object is held by unverified code, its invariant holds and remains true no matter what the unverified code does within the bounds of the assumptions and for every method of that class, if a given method is called while respecting its pre-conditions, then its post-conditions will held.

Enforcing that model is non-trivial as we have to assume that our verification is sound no matter what the unverified code does, and the unverified code has a wide array of options to be disruptive.

To showcase this, we will see a list of challenges this thesis is faced with, and the solution we bring to fix those problems. This will also serve as an introduction to those concepts before we describe them in greater details in Section 3.4 and the sections following it.

```
1  class Node{
2      public Node f;
3      public Node(int n){
4          if( n == 0){
5              this.f = Null;
6          } else {
7              this.f = new Node(n-1);
8          }
9      }
10 }
```

Listing 3.1: Because f is public, passing a Node will indirectly pass other Node's

The class `Node` in the Figure 3.1 has the problematic property that when an object of class `Node` is passed to unverified code, the content of its public fields are also indirectly passed to the same unverified code. By transitivity, every other `Node` in the list is also

passed. In general, which objects exactly are given to the unverified code when a reference is passed to said code is difficult to determine as it requires determining which objects are *reachable* by the passed object.

The solution we bring to this problem is both in the leak operation and how we treat leakable objects. Instead of objects transforming from hidden to leakable as they are passed to the non-verified code, we enforce that the transition is made **before** they are passed. To do so we use the leak operation. The leak operation ensures that every public field of an object has to be leakable themselves before said object can be leaked and thus ensures that even the objects we pass indirectly are safe to pass. It is not the only function of the *leak* operation and we describe it in detail in Section 3.7

But this is not the last challenge we face, let us look at an other example that demonstrates an obstacle to this thesis:

```
1  class Node{
2      private Node f;
3      public Node(int n){
4          if( n == 0){
5              this.f = null;
6          } else {
7              this.f = new Node(n-1);
8          }
9      }
10     public Node getf(){
11         return this.f;
12     }
13 }
```

Listing 3.2: The public getf method can be called by unverified code to optain this.f

Here the field `f` is private instead of public, so we do not have the above-mentioned challenge. But the method `getf()` is now problematic. Should an object of class `Node` be given to unverified code, because the unverified code can simply call `getf()`, the unverified

code will then be able to obtain the content of the field `f` of the passed `Node` object. As things are, when we turn an object from hidden to leakable using the *leak* operation, which we will write as the verb "leak", we would have to take into account every method of the class of the object we wish to leak to check if they do not pass an object to unverified code and thus,for every method in which we we would wish to "leak" an object we would need to take into account all other methods of said object. This would result in a much less modular and thus much more inefficient verification process.

We wish to keep the ability to verify our program one method at a time without taking into account other methods of the same object. To solve that challenge the class invariants are useful. By capturing the behaviour of leakable objects of that class into the invariant, we remove the need to take into account about every method and replace it with the need to take into account the single invariant.

For example, in Figure 3.2, if the invariant states that `this.f` is leakable then since we only pass leakable objects to unverified code, it will also be true for. On the other hand, if the invariant does not states that `this.f` is leakable then the verification of `getf` should not succeed as `getf` could be used to pass an item to unverified code that is marked as hidden. One of the function of the invariant is specifying what fields are safe to pass and which are not. What exactly we need to do to prove the invariant to hold and how it is used is described in greater detail in Section 3.5.

The third major challenge is the ability of unverified code to completely ignore specifications. Here is an example to demonstrate:

```
1 class EvenHolder{
2     invariant := this.f%2 == 0
3     private int f;
4     public EvenHolder()
5         ensures acc(this.f)
6     {
7         this.f = 0;
```

```
8       }
9       public void setf(int i)
10          requires acc(this.f) && i%2 == 0
11          ensures acc(this.f) && this.f == i
12      {
13          this.f = i;
14      }
15
16  }
```

Listing 3.3: Here the specification tries to enforce the invariants and fails to do so.

As this is the first example of specified code in our target language we will describe the specifications. The invariant specifies that the field `f` of an object of class `EvenHolder` should always be even. The specification on the `setf` method specifies that the argument `i` of it should always be even and that the caller of `setf` need to have permission to write to the field `f` of `this` to call `setf`. The problem here is that unverified code can just ignore this pre-condition and can call `setf` with an argument of 1, breaking the invariant of EvenHolder by having `setf` assign an odd value to `this.f`.

We wish to verify for every class that even when unverified code calls a method of an object of said class, the invariant of the class holds still for that object. To do so, we verify every method two times. The first pass is standard in that we verify that, if the methods pre-conditions hold, the methods post-condition hold if the method returns. We also verify in the first pass that invariant holds if the object is or becomes leakable during the method's execution. The second pass however will verify that even if the methods pre-conditions are ignored, the invariant of the receiver object will be maintained by the method. That way we verify both that the class respects its own specification both in regard to its invariant and its methods pre- and post-conditions. As the private fields of a given object cannot be directly modified by other objects, we do not need to verify if methods break the invariant of other objects than their receiver's invariant. We explain this double verification in Section 3.6

The fourth challenge is closely related to the third one. Unverified code ignores the specifications of methods. Notably, they ignore the permissions needed in method pre-conditions. Thus, let us consider a corrected `EvenHolder` class and an example of client code that tries to use the class `EvenHolder`

```
class EvenHolder{
    invariant := this.f%2 == 0
    private int f;
    public EvenHolder()
        ensures acc(this.f)
    {
        this.f = 0;
    }
    public void setf(int i)
        requires acc(this.f) && i%2 == 0
        ensures acc(this.f) && this.f == i
    {
        int correct = 0;
        if (i%2 == 0){
            correct = i
        }
        this.f = correct;
    }

}
```

Listing 3.4: Here EvenHolder is not at fault, its the client code that is wrong

```
public void main()
{
    BadLibrary bad = new BadLibrary(); //with BadLibrary being unverified
   code
    EvenHolder ph = new EvenHolder();
    leak ph;
```

```
6       bad.pubfield = ph;
7       ph.setf(4);
8       assert ph.f == 4;
9   }
```

Listing 3.5: An inocent client code

As we see it for the first time, we focus on the `leak ph`. This statement is a call to the leak operation. The leak operation verify if hidden objects are safe to be passed to unverified code and then makes said objects leakable objects. What exactly need to be proven and the details of the leak operation are described in more detail in Section 3.7.

Considering the specifications of the method `EvenHolder` , without unverified code interference, this statement would be verified because the client code would retain permission to `ph.f` between the call to `setf` and the assertion. But with the implementation of `EvenHolder` as seen in Figure 3.3, the unverified code, here the object `bad`, could call `setf` between the call of the client to `setf` and the assertion to change the value of `ph.f` and thus making the assertion fail. From that we can conclude that verification should fail at one (or multiple) of three points:

Either *the setf implementation*,

the write to bad.pubfield,

the leak statement

the call to setf,

or *the assertion*.

This is a design choice. If we decided that the assertion needed to fail at the `setf` implementation or the assertion then it would have required a change in the permission model as the method does in fact obey its specification and the assertion being verified comes directly from the specification of `setf`. The assertion should not fail because the call to `setf` establishes the assertion. The leak statement should not fail as the invariant is established. Thus,the conclusion that we came to is that the verification should fail at

the call to `setf`. Simply put, when we call `setf` we need to have permissions on the field `this.f`. But permissions to a field imply it can not be modified by non-permission holders while said permission is held. Since it is possible for `this.f` to be modified, we should not be able to hold permission on `this.f`. To maintain the functionality of the permission model we use we therefor need to make sure that, when an object is leaked, the permissions to fields that now can be changed by unverified code be taken away. To do this we introduce *modifiable fields*, and for every class we specify which fields are *modifiable* or not with the *modifiable field list*. All fields that can be modified directly or indirectly by unverified code should be modifiable and the leak operation should take away permissions to those fields. *Modifiable fields* are described in detail in Section 3.5

The above is just a feature list along with the problems those features try to solve. Let us now look at the total picture and see how the different features and tools we presented above work together to solve those problems in a cohesive way.

3.4 Hidden, Leakable

As mentioned before we wish to track for every object whether or not we know if the given object may be accessible by unverified code. We do so by the way of two predicates on references: *hidden* and *leakable* which we will enforce to be mutually exclusive. Like all predicates, one can hold a fraction of the predicate. For leakable, holding a fraction of the predicates mean the same thing as holding the full thing, but for hidden, holding the full permission of the hidden predicate means that, in addition to knowing that the hidden object is not held by unverified code because we are holding a non-zero amount of permission, we have the permission to leak the item. We do not use hidden or leakable on primitive types, as the concept doesn't apply to primitives.

At the beginning of every constructor call we assume the predicate hidden of `this` to hold as unverified code will only get hold of the object when the constructor returns or if

the constructor passes the object to the unverified code before returning.

On the other hand, we assume the predicate leakable of `this` to hold for every object we would directly get from an unverified object, that being any object we got from a public field of said object, any object as the return of a method of said unverified object, any object we got from the public field of an object for whom the leakable predicate holds or finally from the return of a method that specified its return would be leakable.

Since there are no way to get a new object other than using a constructor in verified code or getting the object from unverified code. And since, as we will see further, the leak operation consumes the hidden predicate to create the leakable predicate. And since, finally, no other operation can give either of those predicates to a reference, then we have successfully enforced that all objects can be either hidden or leakable but never both.

We can conclude then that all object are either hidden or leakable but never both as we know that:

- There are no way to get a new object for verified code other than using a constructor in verified code or getting the object from unverified code which both only give one or the other,
- the the leak operation consumes the hidden predicate to create the leakable predicate and thus maintains their mutual exclusivity,
- and no other operation can create the predicate.

Another important property is that since the only way an object can obtain the predicate hidden is at creation, an object can go from hidden to leakable but the reverse is impossible. We will call this property "monotonicity of the hidden predicate".

We now must enforce that having the existence of predicate hidden of a given object means that unverified code does not have access to a pointer to said object. As leakable and hidden are mutually exclusive and since we have the monotony of the hidden predicate , it

is sufficient to prove that all object passed to an adversary are leakable as they are passed. There are five different ways an object can be passed to the unverified code.

- Case 1 *A reference to the object is returned by a method unverified code called.*
- Case 2 *A reference to the object is assigned to the public field of an unverified object*
- Case 3 *A reference to the object is assigned to the public field of a an object held by unverified code*
- Case 4 *A reference to the object is passed as the argument to a method of an unverified object*
- Case 5 *A reference to the object is indirectly passed by being in a public field of another object when this other object is itself passed to the unverified code*

Let us see case by case how we avoid an object for which the hidden predicate is held from being passed to unverified code:

- Case 1 will be dealt with in Section 3.6.
- The solution to prevent case 2 and 3 from happening is to ensure that at every point where an assignment `a.f = b` is done where `f` is a public field of `a` we assert that either we have the a non-zero amount of permission on either the predicate hidden of `a` or the predicate leakable of `b`. That way, either the assignment does not pass an object to unverified code as said code does not hold `a` or `b` is already something that can be safely given to unverified code.
- To prevent case 4 we assert that for every object passed as argument to a method of an unverified object we must hold the leakable predicate.
- Finally, case 5 is dealt with in the leak operation. In the leak operation, amongst other things, it is asserted that the leakable predicate of every public field of the leaked

object must be held, making it so that, when the object is later passed to unverified code, the public fields of the passed object will be themselves safe to pass. The details of the leak operation are in Section 3.7.

As we now know that objects are all created hidden, and that we enforce that every object has to be rendered leakable before being passed to unverified code, we now also know that all objects held for which verified code holds the hidden predicate are not held by unverified code.

For verification, this is very useful. Most of the further changes we do in comparison to normal specification will be exclusive to objects for which we do not hold the hidden predicate. That is because hidden objects behave in way that is similar to how they would behave in an environment where every piece of code is assumed to be verified.

However, we still want to be able to work with leakable objects, and thus the invariant we show in the next section is a specialized tool to work with an object while it is leakable.

3.5 Invariants and the modifiable fields list

In this work, every class that we wish to verify must be specified with an invariant. The invariant specifies how instances of the class behave when they are leaked to and then held by unverified code.

Invariants are declared as a single assertion. The assertion must only concern the private fields of the class it specifies. The reason why no specifications is made about public field is that since the *invariant* specifies behaviour while leakable, the unverified code can easily assign to the public field to break the invariant at any time.

Another concept we must introduce before we can finish specifying what can or cannot be in the invariant is the concept of *modifiable* fields. As was shown with the example in Figure 3.4, it is necessary to capture which private fields unverified code could freely modify indirectly through calls of methods. Since unverified code can call any method of all the

objects it holds at any time, if a field can be modified by a public method call, then that field can be modified at any time.

Thus,we define *modifiable* fields to be all fields whose values can potentially be changed by unverified code when their object is leakable. For every class we define a set of such fields, called *the modifiable field set*.

With modifiable fields introduced we can go back to the invariant. The invariant must not contain a permission or a predicate on a modifiable field, we explain why later in this section.

As any number of methods can hold the same invariant, we enforce that all permissions amount in the invariant be set to wildcard so that we do not break soundness by having the total permission amount for a predicate that is larger than 1/1.

Let us see an example to demonstrate what the invariant and the modifiable fields accomplish:

```
1  class EvenHolder{
2      invariant := this.f%2 == 0
3      modifiable := {f}
4      private int f;
5      public EvenHolder()
6          ensures acc(this.f)
7      {
8          this.f = 0;
9      }
10     public void setf(int i)
11         requires acc(this.f) && i%2 == 0
12         ensures acc(this.f) && this.f == i
13     {
14         int correct = 0;
15         if (i%2 == 0){
16             correct = i
17         }
```

```
18          this.f = correct;
19      }
20  }
```

Listing 3.6: A correct EvenHolder

This third variation of `EvenHolder` has only two difference with the one shown in Figure 3.4, that being the modifiable field list in line 3 containing the field `f` and the fact that `setf` requires `hidden(this)`. Note that this specification only means that verified code can only call `setf` while `this` is hidden. As unverified code ignores specification, when we leak an object of class `EvenHolder`, it will ignore this specification like all others. When an object of class `EvenHolder` is leaked, the field `f` can be modified at all time during the execution of verified code but the field `f` could not be changed in such a way that the invariant of `f` being even is violated. Thus, when reading the field, all one can say is that the invariant is true and will remain true but no other information on the field `f` can be learned.

Thus, for modifiable fields, because they can be modified at any time, holding permission on such a field makes no sense. Because of this, the leak operation must remove all permissions from all such fields of an object before said object is leaked. And that is why we prevent the existence of access permissions on modifiable fields in the invariant.

However,this creates another problem: without permission to a field it is impossible to interact with said field, either by writing or reading. This is overly restrictive and thus we must find a solution to interact with modifiable fields.

The solution is the following: once an object is leakable, we remove its field from the permission system altogether. Instead we rely on the invariant, and the fact that, in our target language, only local variables can be assigned to a field, leading us to those verification steps:

For every assignment to a modifiable field of an object, knowing every field write is of the form `a.f = x`, if object `a` is leakable, we verify that if the invariant held before the

assignment then replacing the content of `a.f` with `x` does not break the invariant. In short, we verify that the assignment maintains the invariant of the receiver of the field.

Because of this verification and the fact that the invariant must be established when an object is leaked, we can now be certain that the invariant will always hold for at least the modifiable fields. Thus,we can do the following:

Thus, for every field read of the form x := a.f where a is leakable and f is modifiable, we know that we get some arbitrary value (which may change at any moment) for which the invariant of a holds.

Let us look at an example to clarify and show how we can use those properties to our advantage:

```
1  class EvenHolder{
2      invariant := this.f%2 == 0
3      modifiable := {f}
4      private int f;
5      public EvenHolder()
6          ensures acc(this.f)
7      {
8          this.f = 0;
9      }
10     public void setf(int i)
11         requires leakable(this)
12         ensures leakable(this)
13     {
14         int correct = 0;
15         if (i%2 == 0){
16             correct = i
17         }
18         this.f = correct;
19     }
20     public void doNothing()
21         requires leakable(this)
```

```
22          ensures leakable(this)
23      {
24          int cont;
25          cont = this.f;
26          this.f = cont;
27
28      }
29  }
```

Listing 3.7: An EvenHolder with a method that demonstrate both reads and writes

Because `this` is leakable, we must maintain the invariant of `this`. To do so, we must prove that the invariant of EvenHolder, `this % 2 == 0`, holds after we assign the value `correct` to `this.f`. Thus, we have to prove that `correct` $\% 2 == 0$. Since this is true no matter which path in the method is taken, we can verify this property and prove the method correct.

Now let us look at the method `doNothing`. The method does not do much but is a good example to demonstrate how we can verify that the invariant of an object is maintained through a method and how we can use the invariant obtained from reading a value from a field to verify to maintain the same invariant when we write to that field. First we focus on the statement `cont = this.f`. As `t his.f` is modifiable we do not need permissions to read it. Instead we simply learns that `cont` is some unknown value that satisfies `cont%2 == 0`. Thus,when we verify the statement `this.f = cont` maintains the invariant as after the statement `cont%2 == 0` still holds.

But what about non-modifiable fields? Since we wish for them to continue using the permission system, we need to ensure that the unverified code can not freely change the field. To do so we need to introduce the *double verification of methods*. We explain informally here but will explain more rigorously in Section 3.6. In addition to verifying the invariant and the method conditions when verified code calls a method, we also verify that if the receiver is leakable and unverified code calls the method without establishing the pre-conditions of the

method then the invariant will not be broken. And since during this part of the verification the caller does not have the permissions to any field of the receiver then if the method tries to modify any non-modifiable field the verification will fail. In practice, this enforces that non-modifiable fields of leakable objects can not be written to.

Finally, we can focus on public fields. Assigning to a public field while the receiver of said public field is leakable is equivalent to passing the assignee to unverified code and doing nothing else. That is because we can never learn more information about a public field than the class of its content and the fact that its content is leakable as the unverified code can easily control the content of the field. Thus, apart from verifying that what is being assigned is leakable, we treat such an assignment as a no-op. We explicitly can not take for granted that the field now has the value we assigned to it as the unverified code could have changed it again afterward. We do bot have to verify that this assignment break any invariant because we specified that invariants could only contain assertions on private fields.

When we read from a public field with a statement of the form `x = a.pub`, all that we know now about `x`, since the value of `a.pub` is entirely controlled by unverified code, is that `x` is leakable, thus,we can simply treat it as an assignment of the form `x = newvar1` with newvar being a leakable fresh variable.

Thus,we also need to remove the public fields from the permission system, because holding any amount of permission to a public field while its receiver is leakable would be nonsensical as unverified code can change directly the fields value.

To summarise:

- Modifiable private fields are removed from the permission system. We verify that the invariant holds for modifiable fields each time we assign to them. Each time we read from such a field we know that the invariant will hold for the read value.

- Public fields are removed from the permission system. We verify that all references assigned to such a field is *leakable*. Each time we read from such a field we know that the read value will be leakable

- Non-modifiable private fields are not removed from the permission systems, Reading and writing from such a field is done as usual.

3.6 Double verification of methods

We wish to verify that when an object is held by unverified code, its invariant holds no matter what the unverified code does within the bounds of the assumptions we made in Section 3.2 and, for every method of that class, if a given method is called while respecting its pre-conditions, then its post-conditions will hold.

The previous sections have explained how to verify individual statements and how for each statements we verify the invariant of the receiver of the method they are in. But we have not explained completely how we could verify said invariant when unverified code is calling the methods, nor have we entirely explained yet how we verify methods pre- and post- conditions.

Thus, in this section, we will be explaining first how we verify pre- and post-conditions as well as how we verify the invariant when unverified code calls methods. We will then explain how this ties into verifying field reads and writes on non-modifiable fields. Finally, we will explain how we verify that a method when called by unverified never pass any hidden object to said unverified code.

To start off, to verify that methods respect their specifications when called by verified code, we use standard verification of methods. That is, we verify that every verified method call establishes the methods pre-conditions. we verify that every method, if its pre-conditions are established at the call site, will establish its post-conditions when the method returns, we also verify for that case that invariants are maintained and that no runtime error can occur.

However, unverified code can call the methods of objects that it holds without respecting their pre-conditions and while establishing their post-conditions is not necessary because no

verified code will depend on its return, it is necessary that the call maintains the invariants of all objects it interacts with, since we assume that those invariants hold for all leakable objects when verifying other code.

To verify that the invariant is maintained even when unverified code is the one calling a method, we will prove that even a method call that ignores pre-conditions will still maintain the invariant.

We know that the receiver of such a call and all references in the method's arguments will be leakable because the receiver has to be an object that unverified call can access directly. We also know that unverified code cannot directly call private methods of verified objects.

Thus,we can do the following: for every public method, in addition to verifying the standard pre- and post- conditions, we verify that, assuming only that this and all arguments that are not of primitive types are leakable (i.e., not assuming any pre-conditions),

- we have all the required permissions for the fields we access we access, including the non-modifiable fields we interact with,
- we fulfill all pre-conditions of methods we call inside the method we verify,
- the invariant is maintained by the verified method.

We also verify that method returns will not pass a hidden object to unverified code even when called by unverified code by proving that every reference returned is leakable even when under the above conditions. By doing so we ensure that Case 5 of section 3.4 does not happen.

Incidentally, the above verification has also verified that non-modifiable fields cannot be modified by unverified code. Since, without pre-conditions, the method cannot obtain permissions to write on fields then if the call tries to write to a non-modifiable field the verification will fail as the required permission is not held by the call. This is not a problem for modifiable fields since we have removed them from the permission system.

It is to be noted however that since it is possible to have reading permissions in the invariant, a method call by unverified code could still read non-modifiable fields. They could not write to it however.

Another fact to be noted, is that while *locks* and *locks invariant* are not supported at the moment by this thesis, their addition could permit writing to non-modifiable fields by simulated call as one could store in the *lock invariant* the write permissions of such a field. We expect however that creating a sound systems with locks will not be too difficult as the locks would then themselves explicitly guard the different method calls from interfering with each other.

Finally, we have to verify the special case of the constructors. At the beginning of a call to a constructor, the created object is in fact hidden, no matter if verified or unverified code called the constructor because the created object will only be accessible to unverified code when the method returns or if the object is passed beforehand to said unverified code.

Thus, the constructor also actually has the permissions to its fields because the unverified code cannot interfere with its fields until it actually has access to the created object. Only when `this` will be held by unverified code do we need to remove the permissions to its modifiable and public fields.

Since the constructor call passes `this` to unverified code at return, we need to verify that `this` is leakable when the constructor returns. Thus,we use the leak operation at every return to enforce this. As we will see in Section 3.7, this in turns verifies that the method establishes the invariant, even when the object was created by unverified code because the leak operation needs that the invariant holds to be verified.

3.7 The Leak Operation

This section is about the *leak* operation we mentioned in multiple section before but did not explain in detail. The *leak* operation is not particularly complex, it simply uses all the other

concepts we introduced, with the exception of Section 3.6.

The leak operation, despite its name, is not actually a transformation or something that passes one or multiple objects to unverified code. Conceptually, the leak operation is actually just a change of labels and we verify that changing those labels is correct. It changes the label of an item from hidden to leakable Thus, at the point where an object is leaked, all properties that must hold for a leakable object must be proved.

In practice. this means that the leak operation

- exhale a full hidden permission,
- inhale a full leakable permission,
- and make a series of assertions

The assertions are the following:

- we assert that the invariant holds,
- we remove from the permissions system all modifiable fields by exhaling the entire permissions on all such fields,
- we assert that all public fields that are not of primitive types of the object are safe to be passed to unverified code by being leakable and we exhale the entire permissions on all public fields.

Let us see an example to demonstrate:

```
1 class EvenHolder{
2     invariant := this.f%2 == 0
3     modifiable := {f}
4     private int f;
5     public int a;
6     public EvenHolder()
7         ensures leakable(this)
```

```
8       {
9           this.f = 0;
10          leak this;
11      }
12
13  }
```

Listing 3.8: An exemple of the leak operation

Here we focus on `leak this`. When we verify it, we will :

- remove the hidden predicate of `this` and add the leakable predicate of `this`,
- assert the invariant holds, that is to say we assert that `this.f%2 == 0`,
- exhale the entire permissions of `f`,
- and exhale the entire permission of `a`, but since `a` is of primitive type we do not assert that it should be leakable.

3.8 Viper Encoding

With the theoretical design done, we can now put the concepts we introduced into practice. To verify programs we will be using the Viper language. To be precise, we will encode the target language in such a way that the if the resulting Viper program is verified with regard to its own specification then the target language will also be verified with regard to its own specification.

The encoding is specified formally in Figure 3.3. We put the syntax of our target language in 3.2 as a reminder for the reader.

A quick notation helper:

- $[\![cd_1]\!]$ means "the encoding of cd_1".

- $[\![cd_1]\!]_{inv,mod}$ means "the encoding of cd_1 if its invariant of the current class verified is *inv*, its set of modifiable fields is *mod* and the access modifier of the current method is *accmod*".

- $[\![cd_1]\!]^{accmod,C}_{inv,mod}$ means "the encoding of cd_1 if the current class is C, its invariant is *inv*, its set of modifiable fields is *mod* and the access modifier of the current method is *accmod*".

- $\forall$ *e S stmt* indicates that we we repeat the statement *stmt* for each element *e* of the set S. Not that a $\forall$ is encoded in Viper

- The fonts are used to differentiate between litteral text and identifiers. `assert` means that assert will be literally encoded while *fieldname* means that fieldname is an identifier or part of some formula.

- $[\![C]\!]$ where C is a class identifier encodes to `Int` and `Bool` for integers and boolean, it encodes to `Ref`

- by combining the last two rules we that: $\forall$ *e* $fieldofC, [\![e.class]\!] ==$ `Ref` *stmt* means that for each field *e* class C whose declared class encodes to `Ref` repeat the statement *stmt*.

(*Programs*)	*Prog*	::=	$cd_1 ... cd_n$
(*Class definition*)	*cd*	::=	`class` C `{` $inv\ mod\ fd_1 ... fd_k\ \ cnd\ \ md_1 ... md_n$ `}`
(*Modifiable list*)	*mod*	::=	`modifiable := {` $fieldname_1, ..., fieldname_k$ `}`
(*Field definition*)	fd	::=	$accmod\ C\ fname$
(*Constructor definition*)	*cnd*	::=	$accmod\ C$ `(` $C_1\ arg_1, ..., C_n\ arg_n$ `)` $spec$ `{` $s_1 ... s_n$ `}`
(*Method definition*)	*md*	::=	$accmod\ C\ mname$ `(` $C_1\ arg_1, ..., C_n\ arg_n$ `)` $spec$ `{` $s_1 ... s_n$ `}`
(*Statements*)	*s*	::=	($ifstmt \mid fieldacc$ `=` $x_2 \mid C\ x \mid\ x_1$ `=` $exp \mid x_3 = mcall \mid leakstmt \mid$ `return` x);
(*If statements*)	$ifstmt$	::=	`if(` x `)` `{` $s_1 ... s_k$ `}` `else {` $s_{k+1} ... s_n$ `}`
(*Leak statements*)	*leakstmt*	::=	`leak` x
(*Method call*)	*mcall*	::=	$x.mname$ `(` $x_1, ..., x_k$ `)` | `new` C `(` $x_1, ..., x_n$ `)`
(*Specification*)	*spec*	::=	`requires` | `assert` | `ensures` $assertion$
(*Invariant*)	*inv*	::=	`invariant :=` $assertion$
(*Access modifier*)	*accmod*	::=	`private` | `public`
(*Field access*)	$fieldacc$	::=	$x.fieldname$
(*Assertion*)	*assertion*	::=	($assertion\ op\ assertion$) | $sing$
(*Operation*)	*op*	::=	`+` | `-` | `|` | `&` | `*` | `/` | `%` | `!=` | `==` | `>` | `>=`
(*Singleton*)	*sing*	::=	$exp \mid spex$
(*Expression*)	*exp*	::=	($exp\ op\ exp$) | $fieldacc \mid x \mid$ `true` | `false` | $intLit$
(*Specification expression*)	*spex*	::=	`hidden(` $x, perm$ `)` | `leakable(` $x, perm$ `)` | `acc(` $x, perm$ `)`
(*Permission amount*)	*perm*	::=	`write` | `none` | `wildcard` | $frac$
(*Fraction*)	$frac$	::=	$intLit$ `/` $intLit$

Figure 3.2: Target language syntax

$[\![cd_1 ... cd_n]\!]$ ::= $[\![cd_1]\!] ... [\![cd_n]\!]$

$[\![class\ C\ \{inv\ mod\ fd_1 ... fd_k\ \ cnd\ \ md_1 ... md_n\}]\!]$ ::= $[\![fd_1]\!] ... [\![fd_k]\!]\ \ [\![cnd]\!]_{inv,mod}$
$[\![md_1]\!]_{inv,mod} ... [\![md_n]\!]_{inv,mod}$

$[\![accmod\ C\ fn]\!]$ ::= `field` fn : $[\![C]\!]$

$[\![$ `public` $C\ (C_1\ arg_1, ..., C_n\ arg_n)\ spec\ \{s_1 ... s_n\}]\!]_{inv,mod}$ ::= `method` C `_cons_h`
$([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$
$spec\ \{bsch_C [\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}\}$

`method` C `_cons_l`
$([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$
$\{bscl^C_{arg1,..,argn} [\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}$
$[\![$ `leak this` $]\!]^{public,C}_{inv,mod}\}$

$[\![$ `private` $C\ (C_1\ arg_1, ..., C_n\ arg_n)\ spec\ \{s_1 ... s_n\}]\!]_{inv,mod}$ ::= `method` C `_cons`
$([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$
$spec\ \{bsch_C\ [\![s_1]\!]^{private,C}_{inv,mod} ... [\![s_n]\!]^{private,C}_{inv,mod}\}$

$[\![$ `public` $C\ mn\ (C_1\ arg_1, .., C_n\ arg_n)\ spec\ \{s_1 .. s_n\}]\!]_{inv,mod}$ ::= `method` mn `_h`
$([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$
$returns$ (`ret:` $[\![C]\!]$) $spec$
$\{[\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}\}$

`method` mn `_l`
$([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$
$returns$ (`ret:` $[\![C]\!]$)
$\{bsml_{arg_1,...,arg_n}$
$[\![s_1]\!]^{public,C}_{inv,mod} ... [\![s_n]\!]^{public,C}_{inv,mod}$
`assert perm(leakable(ret)> 0)` $\}$

$[\![$ `private` $C\ mn\ (C_1\ arg_1, .., C_n\ arg_n)\ spec\ \{s_1 .. s_n\}]\!]_{inv,mod}$ ::= `method` mn
$([\![C_1]\!]\ arg_1, ..., [\![C_n]\!]\ arg_n)$
$returns$ (`ret:` $[\![C]\!]$) $spec$
$\{[\![s_1]\!]^{private,C}_{inv,mod} ... [\![s_n]\!]^{private,C}_{inv,mod}\}$

where

$bsch_C$ = $\forall\ f\ fieldof\ C$ `inhale acc(this.` f `, write)`
`inhale acc(hidden(this),write)`

$bscl^C_{arg1,..,argn}$ = $\forall\ f\ fieldof\ C$
`inhale acc((this.` f `),write)`
$\forall\ arg\ in\ (arg_1, .., arg_n)\ , [\![arg.class]\!] ==$ `Ref`
`inhale acc(leakable(` arg `), write)`
`inhale acc(hidden(this),write)`

$bsml_{arg1,..,argn}$ = `inhale acc(leakable(` arg_1 `), write)`
...
`inhale acc(leakable(` arg_n `),write)`
`inhale acc(leakable(this),write)`

Figure 3.3: Here ::= means " the encoding of ... is" while = means "is"
$\forall$ f fieldof C stmt means "for each field f of the class C do stmt"

$[\![\texttt{if(}\ x\ \texttt{)}\ \ \texttt{\{}\ s_1 ... s_k\ \texttt{\}}\ \ \texttt{else \{}\ s_{k+1} ... s_n\ \texttt{\}}\ ]\!]^{accmod,C}_{inv,mod}$ ::=
$\texttt{if(}\ x\ \texttt{)}\ \ \texttt{\{}\ [\![s_1]\!]^{accmod,C}_{inv,mod} ... [\![s_k]\!]^{accmod,C}_{inv,mod}\ \texttt{\}}$
$\texttt{else \{}\ [\![s_{k+1}]\!]^{accmod,C}_{inv,mod} ... [\![s_n]\!]^{accmod,C}_{inv,mod}\ \texttt{\}}$

$[\![C\ x]\!]^{accmod,C}_{inv,mod}$ ::= $\texttt{var}\ x : [\![C]\!]$

$[\![x_0 = x.mn(x_1, ..., x_n)]\!]^{accmod,C}_{inv,mod}$
where x is verified ::= $x_0 \texttt{ = } x.mn(x_1, ..., x_n)$

$[\![x_0 = x.mn(x_1, ..., x_n)]\!]^{accmod,C}_{inv,mod}$
where x is not verified ::=
$\forall\ i\ in(1-n), [\![x_i.class]\!] == \texttt{Ref}$
$\texttt{assert perm((leakable(}\ x_i\ \texttt{)) > 0}$
$x_0 \texttt{ = } x.mn(x_1, ..., x_n)$

$[\![x_0 = x.fn]\!]^{accmod,C}_{inv,mod}$
where fn ∈ mod,
fn is a private field ::=
$\texttt{assert perm(leakable(}\ x\ \texttt{)) > 0}$
$\texttt{|| perm(hidden) > 0}$
$\texttt{if(perm(leakable(}\ x\ \texttt{)) > 0) \{}$
$\forall\ f,\ f \in mod$
$\texttt{var}\ f\_\texttt{temp} : [\![f.class]\!]$
$\texttt{inhale}\ inv[\forall f', f' \in mod, f'/f'\_\texttt{temp}]$
$x_0 = fn\_\texttt{temp}$
$\texttt{\} else \{}$
$x_0 \texttt{ = } x.fn$
$\texttt{\}}$

$[\![x_0 = x.fn]\!]^{accmod,C}_{inv,mod}$
where fn ∉ mod,
fn is a private field ::=
$\texttt{assert perm(leakable(}\ x\ \texttt{)) > 0}$
$\texttt{|| perm(hidden) > 0}$
$\texttt{if(perm(leakable(}\ x\ \texttt{)) > 0) \{}$
$\forall\ f,\ f \in mod$
$\texttt{var}\ f\_\texttt{temp} : [\![f.class]\!]$
$\texttt{inhale}\ inv[\forall f', f' \in mod, f'/f'\_\texttt{temp}]$
$x_0 = x.fn$
$\texttt{\} else \{}$
$x_0 \texttt{ = } x.fn$
$\texttt{\}}$

$[\![x_0 = x.fn]\!]^{accmod,C}_{inv,mod}$
where fn is a public field ::=
$\texttt{assert perm(leakable(}\ x\ \texttt{)) > 0}$
$\texttt{|| perm(hidden) > 0}$
$\texttt{if(perm(leakable(}\ x\ \texttt{)) > 0) \{}$
$\texttt{var pubvar : } [\![fn.class]\!]$
$\texttt{inhale acc(leakable(pubvar),wildcard)}$
$x = \texttt{ pubvar}$
$\texttt{\} else \{}$
$x_0 \texttt{ = } x.fn$
$\texttt{\}}$

$[\![x.fn = x_0]\!]^{accmod,C}_{inv,mod}$
where fn $\notin$ mod,
fn is a private field ::=

```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if(perm(leakable( x )) > 0) {
  ∀ f, f ∈ mod
     var f_temp : [[f.class]]
   inhale inv[∀f', f' ∈ mod, f'/f'_temp]
   fn_temp = x0
   assert inv[∀f', f' ∈ mod, f'/f'_temp]
} else {
  x.fn = x0
}
```

$[\![x.fn = x_0]\!]^{accmod,C}_{inv,mod}$
where fn $\notin$ mod,
fn is a private field ::=

```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if(perm(leakable( x )) > 0) {
  ∀ f, f ∈ mod
     var f_temp : [[f.class]]
   inhale inv[∀f', f' ∈ mod, f'/f'_temp]
   x.fn = x0
   assert inv[∀f', f' ∈ mod, f'/f'_temp]
} else {
  x.fn = x0
}
```

$[\![x.fn = x_0]\!]^{accmod,C}_{inv,mod}$
where fn is a public field, ::=

```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if(perm(leakable( x )) > 0) {
   assert acc(leakable( x0 ),wildcard)
} else {
  x.fn = x0
}
```

Figure 3.5: ∀ f fieldof C, f ∈ mod means "for every field of C that is in the set mod"

$[\![$ `return` $x]\!]^{private,C}_{inv,mod}$ ::= `return` x

$[\![$ `return` $x]\!]^{public,C}_{inv,mod}$ ::=
```
assert perm(leakable( x )) > 0
return x
```

$[\![$ `leak` $x]\!]^{public,C}_{inv,mod}$ ::=
```
assert perm(leakable( x )) > 0
  || perm(hidden) > 0
if perm(leakable( x )) == 0 {
   exhale acc(hidden( x ),write)
   inhale acc(leakable( x ),write)
   assert inv
   ∀f fieldof C, f ∈ mod || f.accmod == public
      exhale acc( f ,write)
   ∀f′ fieldof C,  f′.accmod == public, [[f′.class]] == Ref
      assert acc(leakable( f′ ),wildcard)
```

$[\![predName(x)]\!]^{public,C}_{inv,mod}$ ::= `acc(predName(` x `),write)`

where *predName* is the name of a predicate `}`

otherwise

$[\![$ `e` $]\!]^{private,C}_{inv,mod}$ ::= e

Figure 3.6: $inv[\forall f \in mod/f\_temp]$ means
"the assertion inv with every field f in mod replaced by f_temp"

The encoding is mostly a 1 to 1 application of the design we presented in the earlier sections with the notable additions of the fact that every time we branch on whether or not a given object is leakable we first assert that we have non-zero permission on either leakable or hidden of said object. The fact that an object is either held or not by unverified code is trivially true, but a careless specifier could lose either predicate by calling a method that required one or the other and ensured neither. To remain sound we require that at any time we use an object hidden or leakable status we must know whether the object is hidden or leakable.

Apart from the above addition, we do apply the design described earlier. For every assignment to a private field we verify that the invariant is maintained by inhaling it before

the assignment and then asserting the invariant again after to verify that it still holds. Every time an object could be passed to unverified code we verify that the passed object is leakable. For every public method, to verify both possible contexts as shown in Section 3.6 we encode method declarations as two different methods, one for each context. *bsch*, *bscl*, *bsml* or "begining statement class hidden", "begining statment class leaked" and "begining statment method leaked" are the various inhalations the verifier does at the begining of the methods body to signify the assumed knowledge, *bsch* for example gives us the access permissions to the fields of this as we just created them. Finally the leak operation is as described in Section 3.7.

3.9 Subclassing

Finally, with the design and encoding complete for a target language without inheritance, we can now start expanding our design to support the addition of Inheritance to our target language.

First, we need to define precisely what "inheritance" means as computer languages have differing definitions of inheritance from one another. In this thesis, we will be speaking about Java-like subclassing [6]. That is to say, a class A can declare itself to be derived from another class B and so A would be the subclass of B and B would be the superclass of A. The opposite is not true, a class cannot declare itself the superclass of another.

We define the ancestor relationship as the following:

- If A is the superclass of B then A is also the ancestor of B.
- If A is the ancestor of B and B is the ancestor of C then A is the ancestor of C

A class is not allowed to declare itself the subclass of a class from which it is the ancestor.

A class can only declare itself subclass of a single other class but a class has no limits on how many subclasses it can have.

Thus, the graph inheritance relationship between classes is represented by a tree which Java calls the Class Hierachy.

A subclass inherits all public methods from its superclass as well as all its public fields. It does not inherit its constructor but the subclass's constructor must call the superclass's constructor before returning. It also does not inherit either private methods and classes. A class may override methods it inherits to change their implementation.

For our thesis, we will only allow a verified class to be a subclass of another verified class. Unverified code can however declare subclasses of verified classes.

Let us show an example to clarify, say we have declared the following classes:

```
1  class SuperC{
2      public int f;
3      SuperC(){
4          int f = 0;
5      }
6  }
7  class SubC extends SuperC{
8      SubC(){
9          super()
10     }
11 }
```

Listing 3.9: A class that inherit from another

Here `extends SuperC` is the declaration that `SubC` is a subclass of `SuperC`. In the constructor of `SubC` the statement `super()` is a call to the constructor of `SuperC` as a subclass must call the superclass's constructor before doing anything else in its constructor.

In Java, subclasses can be used to perform "subtype polymorphism", that is to say, at every place a given class is used one can instead use a subclass of the given class.

Let us again show an example, using the classes defined in Figure 3.9:

```
void main(){
    SubC a = SubC();
    SuperC b = a;
    b.f = 3;
}
```

Here, despite `a` not being the same class as `b` and `a`'s class not declaring an `f` field, because `b`'s class is a ancestor of `a`'s class and because `a` inherits the field `f` this is accepted.

While subclassing and polymorphism are very useful, for us they are also a challenge.

Again an example will let us illustrate why:

```
class SuperC{
    public int f;
    SuperC(){
        int f = 0;
    }
    public int getzero()
        ensures result == 0
    {
        return 0;
    }
}
class SubC extends SuperC{
    SubC(){
        super()
    }
    @Override
    public int getzero(){
        return 1;
    }
}
```

```
void main(){
    SubC a = SubC();
    SuperC b = a;
    a.getzero();
}
```

Listing 3.10: This is valid Java

By overriding `getzero()`, `SubC` can break the postconditions of `getzero()`. In general, with inheritance, calling a method becomes dangerous as if the receiver's class is different than what the verifier expects then the method call could have a very different behaviour than expected. In fact, if the receiver's class in an unverified class then that would mean that verified code called an unverified method. That is why we need to re-evaluate how we verify method calls.

Contrarily from method specifications, invariants are not affected by inheritance because subclasses have to call the constructor of their superclass and thus establish the invariant and because the invariants only specify the behaviour private fields which are not affected by subclassing as they not inherited and thus not accessible to the subclass.

Because of this, we do not have to change the invariant but we will have to adapt how we treat methods and method calls.

First of all, let us speak of subclassing within verified code.

For verified code we wish to use *behavioral subtyping*. Simply put, we wish for methods inherited and Overrided by subclasses to still respect the specifications of those methods as specified by their ancestors.

Let us see an example:

```
class SuperC{
    public int f;
    SuperC(){
        int f = 0;
    }
```

```
6       public int get()
7           ensures result %2 == 0
8       {
9           return 0;
10      }
11  }
12  class SubC extends SuperC{
13      SubC(){
14          super()
15      }
16      @Override
17      public int get()
18          ensures result%3 == 0
19      {
20          return 6;
21      }
22  }
```

Here, because `SubC` inherited `get` from `SuperC`, to respect behavioral subtyping we would need to require that `SubC` respects the specification of `get` as specified in `SuperC`. For this example, this means that `ensures result %2 == 0` must be respected. Of course, the method must also respect its own specification. Here it means that `get` must return a value that is both a multiple of 2 and 3. `get` does so by returning 6.

In general, to enforce behavioral subtyping we use a method similar to the double verification of methods we use in Section 3.6

For every class, we prove the following:

- For every method that is both inherited from an ancestor and overridden by the given class, for every ancestor from whom we have inherited the given method: we will verify, assuming only the pre-conditions the given method is specified with in the given ancestor, that the method post-conditions as specified in the given ancestor will

be established on method return and that the invariant of the given class will be maintained.

Behavioral subtyping is quite handy, because now, as long as we can confirm that an object's actual class, or in other terms its *runtime class*, is a class we verified then we do not have to worry about method contracts being broken when we call methods.

This is why we will also be tracking the *runtime class* of objects. The main tools to discover the runtime class of an object we will be using are the following:

- If verified code calls a constructor, then at the constructor's return the created object will be the exact class of the constructor's class.
- In Java, by using reflection, it is possible to obtain the runtime class of an object using the `getClass()` method. The returned value of that method will be the runtime class of the object that is used as argument. As `getClass()` is a public method of the `Object` class, which is the root of the Class Hierachy, all objects inherit it.

Because an object's runtime class does not change, we can track the runtime class of an object as it is moved and used.

Then, since we can track the runtime class of objects and we know that methods of object of a runtime class that is verified are safe to call, we can do the following:

For every method call:

- If we can prove that the runtime class of the receiver of the given call is a verified class then we treat the method call as normal.
- If we cannot, we treat the method call as a method call where the receiver is an unverified object. Thus, we assert that every argument passed to the method must be leakable and the only thing we can know of the returned object is that it itself is leakable.

And with that, we have incorporated inheritance into our design.

Chapter 4

Implementation

We have developed a Viper plugin that follow the design outlined in Chapter 3. Here are the steps we took in developing this tool. It should be noted that 3.9 has not been implemented.

4.1 Target language adaptation

Having designed a Java to Viper [1] encoding, the implementation process seems straightforward. At first glance implementing the encoding from our target language to Viper and then using the Viper [1] tool to verify the resulting Viper program is enough. Sadly, there exist no parser for our target language and since we use specification constructs, need to modify an existing Java parser fairly extensively to support our target language.

Parsing a Java subset is not the objective of this thesis, and we therefore decided to use a different strategy. Instead of writing a full parser for the target language we instead use a modified Viper language to support classes and other features of the target language. The result is a language similar enough to our target language that the encoding can still easily be applied.

To demonstrate, let us compare a program written in our target language and a program written in the modified Viper language. We will use the `EvenHolder` example we have used in Chapter and some client code3.

```
1 class EvenHolder{
```

```
    invariant := this.f%2 == 0
    modifiable := {f}
    private int f;
    public EvenHolder()
        ensures acc(this.f)
    {
        this.f = 0;
    }
    public void setf(int i)
        requires acc(this.f) && i%2 == 0
        ensures acc(this.f) && this.f == i
    {
        int correct = 0;
        if (i%2 == 0){
            correct = i
        }
        this.f = correct;
    }
    public void main(){
        EvenHolder ex = new EvenHolder();
        ex.setf(2);
        }
}
```

Listing 4.1: EvenHolder in Java

```
class EvenHolder{
    invariant this.f%2 == 0
    @private() field f: Int
    @public() method EvenHolder_cons(this : EvenHolder)
        ensures acc(this.f)
    {
        this.f := 0
    }
```

```
9     @public() setf(this : EvenHolder,i : Int)
10        requires acc(this.f) && i%2 == 0
11        ensures acc(this.f) && this.f == i
12    {
13        val correct : Int
14        int correct := 0
15        if (i%2 == 0){
16            correct := i
17        }
18        this.f := correct
19    }
20    @public method main(this: EvenHolder){
21        var ex: EvenHolder
22        EvenHolder_cons(ex)
23        setf(ex,2)
24    }
25 }
```

Listing 4.2: EvenHolder in Viper

While the syntax changes, all major elements that can be found in the target language version can also be found in the Viper version.

The only major difference is that Viper does not support methods having a receiver, so we instead required all methods to have a "this" argument that serves as a de-facto receiver. This is important because in constructors the created object cannot be an alias to the arguments to the constructor. Since we treat the created object as an argument in Viper, Viper does not know that "this" and the arguments of the constructor are not aliases of each other. This does not affect soundness because the number of provable facts does not increase. But it does affect completeness as proofs that could be done while supporting receiver in methods can't be done.

Additionally, to differentiate verified from unverified classes, we use the `@bad()` anno-

tation to mark unverified classes. Something which we did implicitly in the Java encoding.

Thus, our implementation is a Viper Plugin that does the following:

- Use the Viper parser, which we have modified slightly to support classes, to parse programs written in our modified Viper language.
- Encode the parsed program using the encoding we designed in the previous chapter.
- Pass the encoded program to the Viper back-end so that it can verify it.
- We report the result to the user.

Our implementation is, apart from the differences cited above, has all the features of the encoding and can be used to verify all programs that could have been verified using the encoding.

We evaluate how successful our plugin is in Chapter 5.

Chapter 5

Evaluation

According to the introduction of this thesis, the objective of this thesis was to design a more robust verification system that should be sound even in the presence of unverified and potentially malicious code. This verification system should both be able to prove both method specifications and class *Invariants*. We then aim to implement a Viper plugin that will apply this tool to verify programs in a concrete manner.

We can decompose this statement into three main objectives. They are:

- Designing a verification system that remains sound in the presence of unverified and potentially malicious code.
- Designing a verification system that is able to prove method specifications and class invariants in the presence of unverified and potentially malicious code.
- Implementing that verification system in a way that conserves the two first objectives.

Thus, to measure our success we will be using those three objectives as our main criteria. We will be evaluating:

- Is our verification system sound?
- Is our verification system useful in proving interesting properties?
- Does the implementation of our verification system maintain the soundness and completeness we have achieved in the theoretical design?

5.1 Soundness

Is our verification system sound?

As we have not provided a formal proof that our verification system is sound, it is possible that there exists a counter-example where one can prove incorrect properties with our verification technique.

However, we have not found such a counter-example to the current version of our verification technique at the time of writing of this thesis. To avoid soundness issues, we have:

- used an iterative design process. We have changed the design and implementation of our verification technique each time we have found a counter-example to an earlier version of our verification technique with the intent of adapting the verification technique to the new case.
- tried to be thorough in the justification of every design choice.
- tested the verification technique implementation with both examples that should be verified and examples that should not. And while we have found examples where we could not verify true properties, we have not found examples where we could verify false properties.

In conclusion, while we do not proove with complete certainty that our verification system is sound, we did not find any element indicating that it was unsound.

5.2 Completeness

Is our verification system useful in proving interesting properties?

To begin answering this question, the invariant expresses useful properties about minimal class behavior, that is to say properties that holds no matter what. This is very useful for

safety as one can guarantee that some unwanted behaviors cannot happen even when an object is accessible by unverified code, something that was not possible before this thesis.

In addition to that, the hidden/leakable system allows us to very precisely determine which objects can be trusted to not be interfered with by unverified code. This allow us to be more permissive with such objects and prove greater properties.

The invariant is however fairly restrictive in what it can do. The invariant proves properties about the state of a given object but not on its evolution. It is, for example, possible to prove that the field of an object contains only an even or odd number or that the content of the field will be always within a given range, but impossible to specify in the invariant that an integer field only increases.

Secondly, in theory one does not need to prove that method maintain invariants on hidden objects. However, in practice, a method's implementation is identical whether or not `this` is hidden or not. This results in the fact that methods also often need to maintain invariants when hidden. This is often overly restrictive. Especially with objects which are never meant to be leakable in the first place. While this does not affect soundness it leads to programs being rejected despite no breaking specification in practice because we are unable to specify in the current version of the verification technique that some objects will never be leakable.

Thirdly, as we do not support locks, it is difficult to prove useful properties for mutable structures that are leakable. For example, we cannot really implement without locks a mutable list structure with a method that provides the length of the list reliably as unverified code could modify said length before the method returns.

In conclusion, while there are restrictions on what properties can proved, useful properties can be proved in the presence of unverified code and, with the addition of further refinements, our technique could be used to prove more advanced properties

5.3 Implementation

To verify our implementation we have tested it on 12 examples: they include examples 3.5 , 3.8 and 3.7.

The verifier was able to verify all examples that were correct with regard to their specifications and did not verify examples that were incorrect with regard to their specifications.

Name of example	Loc	Time to Verify[s]	Loc of encoded program
ConditionalSetterSuccess	21	5,19	96
ContainerFail	18	5,28	92
EvenHolder_success	22	6,09	97
example_Design_5 3.5	22	6,28	78
example_double_1 3.8	12	5,25	66
example_modifiable_2 3.7	30	5,41	150
HiddenPropagationSuccess	34	5,27	232
LayeredLeakFail	27	5,44	128
LayeredSetSuccess	28	5,63	134
LayerSetFail	27	5,62	121
Leakable_basic_success	25	5,44	119
VeryBasic_success	19	5,39	91

The times were measured on a 13th Gen Intel(R) Core(TM) i7-13700HX and were measured as the average time of 10 runs for each examples rounded up to the 100th of a second.

In practice, for short examples the verification time is quick. This make us believe that the implementation is reasonably efficient.

As mentioned in Chapter 4, the only major difference between the implementation and the encoding does not affect soundness. Thus, the soundness of the theoretical design is maintained.

For completeness however there are proofs that cannot be achieved because of the difference, mainly in constructors as Viper does not recognise that `this` is not an alias of other arguments of the constructor that are of the same class as `this`. This in turns creates problems in the construction of recursive structures. This does not entirely prevent their use however but is an still an obstacle.

In addition to that, we have not implemented the extension of the verification technique to support sub-typing.

Thus, we conclude that, while the implementation does not maintain the entirety of the completeness of the theoretical design of the verification technique, it does preserve most of it . The implementation also completely maintain the soundness of the theoretical design.

Chapter 6

Conclusion

In this thesis, we defined a subset of Java which supports classes with methods, fields, access modifiers, variables and branching. We then created a set of assumptions on how this language was used, both by code we wish to verify and code that we did not wish to or cannot verify.

Next, we presented a verification technique that could prove properties on classes that we verified, even if they interacted with unverified code through unverified classes and objects that could run in parallel with verified code.

We defined an encoding that would apply this verification technique to create Viper programs that, if they were verified, would prove properties about programs written in the Java subset we specified.

We then extended our technique to prove properties under the same conditions but with the addition of subclassing and inheritance to the Java subset we specified.

Additionally, we implemented the encoding into a Viper [1] plugin which we evaluated using a number of examples.

Finally we evaluated the success of the verification technique and discussed its utility.

Bibliography

[1] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5.

[2] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[3] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit dynamic frames". In: *ACM Trans. Program. Lang. Syst.* 34.1 (May 2012). ISSN: 0164-0925. DOI: 10.1145/2160910.2160911. URL: https://doi.org/10.1145/2160910.2160911.

[4] Gavin Bierman, M. Parkinson, and A. Pitts. "MJ: An imperative core calculus for Java and Java with effects". In: (June 2003).

[5] Apr. 2024. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html.

[6] URL: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html.

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:

Verification of Object Invariants in The Presence of Unverified Code

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

Last name(s):	**First name(s):**
Necker	Frédéric

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)**
Zurich the 15.05.2024	Frédéric Necker

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