Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verification of Information Flow Security in the Presence of Unverified Code

Master Thesis

Nicolas Gabriel Kaletsch

September 15, 2025

Advisors: Prof. Dr. Peter Müller, Dr. Marco Eilers
Department of Computer Science, ETH Zürich

Acknowledgments

First, I want to thank my supervisor Marco Eilers for his support and feedback throughout this thesis. Many hours of interesting discussions have added significant value to this thesis. In addition to great technical advice, he found the right words and encouraged me in moments of doubt.
I would like to thank Prof. Dr. Peter Müller and his group for providing a supportive environment and giving me the opportunity to work in such an exciting research field.
Finally, I want to thank my family and friends for their continuous support at all times.

Abstract

Information flow security is an important security property that ensures that no secret information is leaked to outside observers. Most existing approaches to proving information flow security are either imprecise by statically partitioning heap memory into sections of low and high confidentiality or do not support unverified code. One such approach is program verification, which usually requires the entire code base to be verified. In some scenarios, this is not possible. For example, in a plugin system, parts of the code are only added at runtime and therefore cannot be verified.

In this thesis, we bridge this gap by presenting a novel permission-based program verification technique that supports precise reasoning about information flow security in the presence of unverified code. Our approach works on a subset of Java and is secure against an adversary that controls parts of our code base and has access to dynamically changing heap memory. We provide an encoding and implement our design on top of the VerCors verification tool. With the evaluation of our implementation, we demonstrate that our technique achieves the desired security guarantees and can be efficiently automated.

Contents

Chapter 1

Introduction

As software systems become increasingly more complex, so does the task of ensuring their correctness. There are several ways to tackle this problem and increase the trustworthiness of a program. The most widely used practice is software testing. While this is sufficient for many applications, there are cases where one would like to have even stronger guarantees. In those cases, program verification allows us to formulate and mathematically prove properties about programs, in order to ensure their correctness.

In addition to proving specifications of what the program is intended to do, program verification is especially useful in providing security guarantees. An important security property that a program should satisfy is that secret information such as passwords or encryption keys are not leaked to outside observers. This property is called ***information flow security*** or ***secure information flow***. The motivation behind it is self-evident: confidentiality of data must be preserved. For such important properties it is especially desirable to have formal proofs.

There are various approaches to proving information flow security for different programming languages. However, most of these techniques are either imprecise or only work when the entire code base of the program can be verified. For example, many existing techniques, such as type systems ([14]), are partitioning the memory locations statically into high and low confidentiality. That is, a variable can either store secret data or never store secret data. This leads to a loss of completeness, as can be seen in the example in Figure 1.1.

A different approach that does not have this kind of incompleteness problem is using program verification to prove information flow security. There exist techniques and tools specifically developed for proving information flow security such as SecCSL [10] and its implementation SecC (available at [9]) for concurrent C code. There has also been previous work of extending an existing program verifier to support information flow security proofs: Eilers

```
1 class MyClass {
2
3     public void myMethod(int secret){
4         int x = secret;
5         x = 0;
6         System.out.println(x);
7         //secure, since x does not contain secret data anymore
8     }
9 }
```

Figure 1.1: Example of an information flow that is secure but a type system would not allow.

et al. [6] have extended the automated Python program verifier Nagini [7] and the underlying verification infrastructure Viper [13] to support verification of information flow security. Both of these techniques allow programs such as in Figure 1.1 but have a different downside: Verification tools usually require that the entire code base of the program can be verified. There are cases where this is not possible. For example, let us consider a plugin system that can add arbitrary plugins at runtime. In this setting, parts of the code base are not present at compile time and therefore cannot be verified.
In this thesis, we bridge this gap by presenting a verification technique for information flow security with support for unverified code in the code base, while still not sacrificing completeness as type system approaches would.

When trying to prove properties about a program with some parts of code unverifiable, new problems arise. The unverified code could basically do anything, as no properties can be proven about this code. In his master thesis [15], Necker has proposed a technique that allows the verification of method specifications and class invariants in the presence of unverified code for Java-like languages.

In the context of verifying secure information flow in the presence of unverified code, we need to take into account that the unverified code base may contain adversarial code. Thus, we can never leak information about secret inputs to the unverified code base. Compared to the type system approach where the heap locations accessible to the adversary are statically determined, in this scenario, it changes dynamically. Whenever we leak a new object to the unverified code, the adversary can access more heap locations.

Thanks to access control mechanisms such as access modifiers in Java, it is still possible to store secret information in objects that are accessible from unverified code. For example, in the program in Figure 1.2, line 10 is considered an insecure information flow. The unverified function could store the reference to object `a` and then later read the secret that is stored in the **public** field `x`. On the other hand, the assignment on line 11 is secure, since

```
1  class MyClass {
2      public int x;
3      private int y;
4
5      public void myMethod(int secret){
6          MyClass a = new MyClass();
7          a.x = 0;
8          // call to unverified library
9          unverifiedFunction(a);
10         a.x = secret; // insecure
11         a.y = secret; // secure
12     }
13 }
```

Figure 1.2: Example of storing a secret in a leaked object securely and insecurely

field `y` is **private**, and therefore cannot be accessed outside of the `MyClass` class.

In this thesis, we propose a technique to verify secure information flow in the presence of unverified code for a subset of Java. For that we build upon the previously mentioned encoding of Necker [15]. We implement our design for the Java frontend of the VerCors verification tool [1] and evaluate it on a number of examples. Since VerCors also uses Viper [13] as a verification infrastructure, we reuse and adapt the aforementioned work of Eilers et al. [6] for proving secure information flow specifications.

We start by giving the necessary background knowledge in Chapter 2. Specifically, we cover important concepts from the previous works of Necker [15] and Eilers et al. [6] that are needed for our design. We then present the design of our verification technique and illustrate it with examples in Chapter 3. In Chapter 4 we document our implementation in VerCors and evaluate it in Chapter 5. Finally, we conclude the thesis in Chapter 6.

Chapter 2

Background

2.1 Information Flow Security

In this section, we give a quick overview of information flow security illustrated with examples. We then discuss the technique of Eilers et al. [6] to prove this property.

As described by Sabelfeld et al. [20], there are different ways of violating information flow security, that is, leaking a secret (see examples in Figure 2.1):

- *explicit* information flow: Secret data is directly leaked to a public output.
- *implicit* information flow: Secret data is leaked through the control flow of a program.
- *timing channels*: Through measuring the time until some action occurs (e.g. program termination, a print statement or setting a variable that the adversary can read) the adversary gains information about a secret value.
- *termination channels*: It depends on a secret value, whether the program terminates.

A common approach to verifying information flow security is to verify an equivalent property: ***noninterference***. This is also the approach that Eilers et al. [6] have taken. ***Noninterference*** states that two executions with the same public inputs but different secret inputs will have the same public outputs. In other words, secret input of the program does not influence public output. It follows that by observing the public output, an adversary does not learn any information about the secret input. This property relates two executions of the same program and is hence called a 2-hyperproperty [4].

```
1  class MyClass {
2
3      static int secret = 100; // secret value
4
5      // explicit
6      public void explicit(){
7          System.out.print(secret/2);
8      }
9
10     // implicit
11     public void implicit(){
12         if(secret > 50){
13             System.out.print(0);
14         } else {
15             System.out.print(1);
16         }
17     }
18
19     // Timing channel
20     public void timing(){
21         while(secret-- > 0){
22             doSth();
23         }
24         System.out.print(0);
25     }
26
27
28     // Termination channel
29     public static void main(String[] args){
30         while(secret > 100){
31             doSth();
32         }
33     }
34 }
```

Figure 2.1: Examples for the different ways of violating secure information flow.

Most often, program verification tools are only able to reason about one execution of a program. In their extension of Nagini and Viper, Eilers et al. used a ***modular product program (MPP)*** construction [8] to allow reasoning about hyperproperties, specifically noninterference.

The idea is to transform the given program into a new composite program that corresponds to two executions of the original program. To achieve this, every variable is duplicated and two ***activation variables*** are introduced. The latter are used to track whether a point in the program is reached by the corresponding execution. That is, if an activation variable `p1` (`p2`) evaluates to `true`, then the first (second) execution is active at that point. The duplicated variables are used to have two disjoint states of the program for each execution. This way, for example, a variable assignment can only be

```
1  class MyClass {
2
3    void bar(boolean z){
4        ...
5    }
6
7    void foo(int x){
8      if(x > 0){
9        y = 1;
10     } else {
11       y = 2;
12     }
13     bar(y > 0);
14   }
15 }
```

```
1  class MyClass {
2
3    void bar(boolean p1, boolean p2,
4      boolean z1, boolean z2){
5      ...
6    }
7
8    void foo(boolean p1, boolean p2,
9        int x1, int x2){
10     boolean p1t = p1 && x1 > 0;
11     boolean p2t = p2 && x2 > 0;
12     boolean p1e = p1 && !(x1 > 0);
13     boolean p2e = p2 && !(x2 > 0);
14     if(p1t) y1 = 1;
15     if(p2t) y2 = 1;
16     if(p1e) y1 = 2;
17     if(p2e) y2 = 2;
18     if(p1 || p2)
19         bar(p1, p2, y1 > 0, y2 > 0);
20   }
21 }
```

Figure 2.2: Example for the MPP transformation (adapted from Eilers et al. [6])

applied to one version of the variable, depending on whether the corresponding execution reaches the assignment (that is, the corresponding activation variable evaluates to true). When encountering a conditional branch, new activation variables are created for each execution and branch, to correctly model which execution takes which branch. The transformation is illustrated in the example in 2.2 adapted from Eilers et al. [6].

In the following, we will denote by e_i the version of an expression e in execution i, for $i \in \{1, 2\}$. We will use a similar notation for variables and instance fields.

Specifications about one execution of the program are called ***unary*** specifications. During the MPP construction these are transformed into the conjunction of two logical implications to hold for each execution only if the corresponding activation variables are true. That is, for example, the precondition $x < 100$ with activation variables p_1, p_2 (we write $[\![x < 100]\!]^p_{MPP}$ for the MPP transformation) would become $(p_1 \implies x_1 < 100) \wedge (p_2 \implies x_2 < 100)$.

In addition to the unary specifications, the MPP transformation enables us to express ***relational*** specifications, that relate the two simulated executions. For our case, we use two relational secure information flow specifications from Eilers et al. [6]: ***low assertions*** and ***low event assertions***.

The ***low assertion*** $low(e)$ holds if and only if the expression e is public. Using

the noninterference formulation, this is the case iff both executions have the same value for the expression. Hence, the low expression is resolved as $[\![low(e)]\!]^p_{MPP} := p_1 \wedge p_2 \implies e_1 = e_2$. Note that if an input is not specified as low, by default it is considered high. The low expression is needed to prevent ***explicit*** insecure information flows, as in the first method in Figure 2.1. The print method would require its argument to be low, which would lead the method `explicit` to fail verification as expected.

The ***low event assertion*** *lowEvent* holds if and only if the current point of the program is reached by both executions or neither, that is, independently of secret (high) values. The respective encoding is therefore $[\![lowEvent]\!]^p_{MPP} := p_1 = p_2$. The low event assertion is crucial to prevent ***implicit*** insecure information flows, as in the second method in Figure 2.1. In this example, the print method would have *lowEvent* as a precondition, to ensure that its call does not leak high values. As intended, the method `implicit` would then be rejected.

Sometimes it is intended to leak secret depending data. For example, consider a program that checks whether the user entered the correct password. Then it is desirable for the program to leak the expression $userInput == correctPW$, by behaving differently depending on whether the expression evaluates to true or false. For such cases Eilers et al. [6] have implemented the `declassify` statement, which for an expression, makes the transition from high to low. It is equivalent to assuming that the expression is low and translated as $[\![declassify(e)]\!]^p_{MPP} :=$ `assume` $p_1 \wedge p_2 \implies e_1 = e_2$. That way, it is possible to leak high values if one explicitly states that it is intended.

2.2 Viper and VerCors

This section briefly describes the VerCors verification tool [1] and Viper [13], the underlying verification infrastructure.

2.2.1 Viper

Viper [13] is a verification infrastructure for permission-based reasoning (see Section 2.2.3). It enables modular verification of programs specified in the Viper intermediate language. This includes the verification of method pre- and postconditions and loop invariants in isolation. For verification, Viper offers two different backends: Silicon, which is based on symbolic execution, and Carbon, which is based on verification condition generation.

We will highlight important features of Viper, such as the permission system and the support for predicates in Sections 2.2.3 and 2.2.4, respectively.

2.2.2 VerCors

VerCors [1] is a verification tool that provides a frontend for multiple programming languages such as Java, OpenCL and OpenMP for C. VerCors functions by transforming the input program and its specification first into an intermediate language and then into the Viper intermediate language. The Viper program is then verified by using one of the two Viper backends.

Since VerCors builds on top of Viper, their concepts and features are very similar. We therefore discuss them together in the following sections.

2.2.3 Permission System

In order to prove the absence of deadlocks and data races, Viper uses a permission system based on implicit dynamic frames [21, 2] with fractional permissions following the idea of Boyland [3]. Implicit dynamic frames is a variant of separation logic [19] that supports heap-dependent expressions inside assertions. Every heap location has a corresponding permission. This permission can be split into arbitrary fractions between 0 and 1 that can be given to functions in different threads. Viper ensures that the sum of all threads for a permission is at most 1. Having non-zero permission for a memory location permits reading from the location, whereas full permission (1) allows to write to that location. This way, multiple threads may read a heap location at the same time. Writing requires full permission, guaranteeing that no other thread can read or write the heap location. Permissions are transferred from the caller to the callee via preconditions and from the current context to the loop body via the loop invariant.

In the VerCors Java frontend, a permission to a field `f` of an object `o` is denoted `Perm(o.f, `*val*`)` where *val* can be a fraction, 1 or one of the keywords `write` or `read`. `Perm(o.f, write)` is equivalent to `Perm(o.f, 1)` and denotes full permission. `Perm(o.f, read)` (also called ***wildcard permission***) denotes some positive non-zero amount of permission.
The `inhale` and `exhale` statements can be used to add, respectively remove permissions from the program state.
Furthermore, `perm(o.f)` can be used to obtain the amount of permissions that are currently held.

2.2.4 Predicates

As an abstraction over assertions, both VerCors and Viper support predicates [18]. A predicate encapsulates an assertion, denoted its ***body***. In place of the assertion, the predicate can then be transferred between different methods and loop executions, similarly to a permission. As with permissions, it is also possible to hold only a fraction of a predicate.

```
1  class MyClass {
2
3      public int x;
4
5      public void foo(){
6          MyClass obj = new MyClass();
7          unverifiedFunction(obj);
8          obj.x = 3;
9          assert obj.x == 3;
10     }
11 }
```

Figure 2.3: Example how unverified code leads to failing assertion.

The body of a predicate can be recursive. However, it must be ***self-framing***, that is, it must provide permissions for all heap locations it contains. For example, assertion `o.f > 0` is not self-framing, while `Perm(o.f, read) ** o.f > 0` is. It is also possible to specify an ***abstract predicate*** without a body.

2.3 Verification in the Presence of Unverified Code

In this section, we will provide an overview of Necker's [15] design for verification in the presence of unverified code for a subset of Java.

The key problem with verification of a code base with unverified code is that said code does not need to respect any rules of the verification technique. For example, for verification techniques based on permission logics, the unverified code can ignore the permission system. That is, it can access heap locations without having the required permissions, rendering the verification technique unsound. Consider the example in Figure 2.3. If we assume that the code in `unverifiedFunction` respects the permission system and returns the permissions to `obj.x`, the assertion on line 9 should verify. However, if it contains arbitrary Java code then the assertion might not hold: Inside `unverifiedFunction` a different thread could be spawned in which the value of the field x is repeatedly changed to an arbitrary value different from 3.

As can be seen, the presence of unverified code makes it much more difficult to prove specifications about the verified code. Necker [15] has tackled this problem for a subset of Java for the Viper verification infrastructure. Since Viper is based on implicit dynamic frames [21, 2], it suffers exactly from the problem illustrated in 2.3.

2.3.1 Hidden and Leakable

In order to keep track of which heap locations could be changed by unverified code, Necker introduces the notion of ***hidden*** and ***leakable*** objects. Every object in the program is either ***hidden***, meaning that the unverified code cannot have access to it, or ***leakable***, implying that unverified code could potentially have access to the object. For primitive types hidden and leakable are not defined. A key observation is that once an object has been leaked to unverified code, it is always accessible from it. Hence, Necker's encoding makes sure that an object can transition from hidden to leakable, but not the other way around. To simplify handling of nested objects, Necker forces the user to explicitly state when an object transitions from hidden to leakable. To do so, they use the ***leak operation*** $leak(obj)$. This statement changes obj to be leakable and no longer hidden. Necker demonstrates multiple ways of how an object can be passed to unverified code. His encoding enforces that before an object is passed to unverified code, it must be leakable. In other words, the user must use the leak operation before exposing the object to unverified code.

Hidden and leakable are modeled as abstract predicates for which fractional permissions can be held. For the leakable predicate, the distinction of having fractional or full permissions does not make a difference. For the hidden predicate, on the other hand, full permissions are required to be able to use the leak operation and make the object leakable. That way, it is ensured that there are no parts in the code that still have fractional hidden permissions on an object while it is being leaked by other code running in parallel.

2.3.2 Class invariant

Hidden objects can be handled exactly the same way as if there was no unverified code in the code base. Leaked objects are different: For example, a field read from a leaked object gives us significantly less guarantees than usual, as we have seen in Figure 2.3. Necker presents the following guarantees for field reads of leakable objects.

In the case of a ***public*** field read, the only information we can learn is that the value is of the correct type and is leakable. This is due to the fact that the unverified code can change the content of the field at any time to any leakable value. This is modeled by removing all public fields of a leaked object from the permission system. A read to such a field is then modeled as reading an arbitrary leakable value.

For ***private*** field reads, Necker makes the assumption, that the unverified code cannot access them. That is, he assumes that the unverified code ***respects the Java language rules*** and ***reflection is disallowed***. The latter assumption is justified by the fact that it can be enforced, for example, by the security

```
1  class EvenHolder {
2
3      private int f;
4
5      public void setF(int i){
6          int correct = 0;
7          if(i%2 == 0){
8              correct = i;
9          }
10         this.f = correct;
11     }
12 }
```

Figure 2.4: EvenHolder example adapted from Necker [15].

manager [17]. This way, there are some more guarantees that a private field read gives us. For that, Necker partitions the private fields into ***modifiable*** and ***non-modifiable***.

A private field is called ***modifiable*** if the unverified code can change its content by calling public methods of the object. Similarly to public fields, modifiable fields are removed from the permission system once an object becomes leakable. Depending on the public methods of a class, there are properties that always hold for the modifiable fields. Consider the example in Figure 2.4. In this example, the unverified code can only change the value of `f` by calling the `setF` method. Therefore, we can be sure that `f` can only be set to even values. To be able to specify information like this, Necker presents the concept of the ***class invariant***. The class invariant is an assertion that contains ***only private fields*** and must always hold ***from the moment the object is leakable***. A suitable class invariant for the `EvenHolder` class in Figure 2.4 would be `this.f % 2 == 0`. Note that no permission is needed, since `f` has been removed from the permission system.

From a modifiable field read, the only thing that we learn is that the class invariant holds. A leakable modifiable field read is thus modeled as reading an arbitrary value for which the class invariant holds.

Non-modifiable fields are the only fields that are not removed from the permission system when the object becomes leakable. Reading and writing to them works as before. They can also occur in the class invariant but must be self-framing, that is, the invariant has to contain a read permission to every non-modifiable field that appears in it. Furthermore, only wildcard permissions (`Perm(x.f, read)`) are allowed inside the class invariant.

It is the user's responsibility to specify a suitable class invariant and mark the modifiable fields. Necker's verification technique then checks that for a leakable object, the specified invariant always holds and that non-modifiable

fields cannot be changed by unverified code. For this purpose, Necker presents the double verification of methods.

2.3.3 Double Verification of Methods

It is possible for unverified code to call any public method of a leaked object at any time. Therefore, every public method is verified twice.

The ***first verification*** is normal: Assuming the preconditions of the method, the postconditions must be proven. This corresponds to the case where the method is called from verified code and the preconditions are checked on the caller side.

The ***second verification*** corresponds to the case where the method is called from unverified code, which does not need to respect the preconditions. In this second verification, there are no preconditions, but in turn also no postconditions to be proven. The only thing that can be assumed is that the receiver as well as all arguments are leakable. This is justified by the fact that the unverified code does not have access to hidden objects. As there are no preconditions, there are also no permissions in the second verification of a method. As a consequence, non-modifiable fields cannot be written to. Public and modifiable fields on the other hand can, since they have been excluded from the permission system and are encoded differently.

The complete encoding of Necker's [15] verification technique will be presented together with our extensions in Chapter 3.

Chapter 3

Design

In this chapter, we explain the design of our verification technique to prove information flow security in the presence of unverified code. In our design, we build upon the technique of Necker [15], which we have covered in Section 2.3.

We start by describing our adversary capabilities and the verification guarantees that our method provides. We then present the core properties of Necker's verification technique, as well as our additions. Next, we describe the Java subset that we consider. For every construct in this subset, we then present the proof obligations and guarantees on a high level, illustrated with motivating examples, before discussing extensions that we have added to Necker's [15] technique. Finally, we conclude this chapter by giving the concrete encoding, derived from our proof obligations and extensions.

For a sound verification in the presence of unverified code, we reuse Necker's [15] encoding. While he presented it as a transformation in the Viper intermediate language, we adapt it to be applicable directly within our Java subset. To that encoding, we add proof obligations that ensure secure information flow. These proof obligations use the relational specifications of Eilers et al. [6] plus an additional specification we developed for our use case described in Section 3.6.7. The product program transformation of Eilers et al. [6] can then be applied after our unverified code and secure information flow encoding.

3.1 Adversary Capabilities and Verification Guarantees

3.1.1 Adversary Capabilities

As Necker [15] does for his verification technique, we assume that the unverified code:

- respects the Java language rules
- cannot use reflection

Essentially, this means that the adversary has no way to access private fields and methods.

Except for that, the adversary in the unverified code is not restricted. We allow calling public methods of leaked objects, spawning new threads, and measuring execution times. Furthermore, the adversary can create subclasses of verified classes. If our verified code calls an unverified method, the adversary could, for example, call a public method on a leaked object in a different thread and measure its execution time.

3.1.2 Verification Guarantees

With the design of our verification technique and the described adversary capabilities, we provide similar verification guarantees as Necker [15]:

1 Sound method verification

2 Crash-freedom of verified code, when ***called from verified code***, but ***not when called from unverified code.***

Our main contribution is the addition of secure information flow as a verification guarantee:

3 Information flow security: The adversary does not learn any information about the high inputs. We guarantee the absence of

a) Explicit insecure information flow

b) Implicit insecure information flow

c) Covert channels: Timing and termination channels

As noted by Necker [15], unverified code can crash at any time. There is no way to guarantee that an unverified method does not crash. Therefore, verification guarantee **2** only prevents crashes that occur directly in verified code, and not those that occur inside an unverified method called from verified code.

Our verification guarantee **1** is exactly the same as the one from Necker [15], whereas **2** is slightly different. Necker [15] provided the guarantee that there are no crashes directly in verified code, ***regardless*** of whether called from verified or unverified code.

The motivation for our adaptation is that, as noted, we cannot prevent the unverified code from crashing. Therefore, it provides little benefit to guarantee that a verified method called from unverified code will not crash. In the information flow security setting, instead of guaranteeing crash-freedom

when called from unverified code, it is much more important that a verified method does not leak any secret information. So, as part of verification guarantee **3**, we verify the absence of insecure information flow also indirectly through crashes or exceptions. This means that we allow a method to throw an exception or crash when called from unverified code, as long as this does not leak information about high inputs.

We will explain this extension in more detail in Section 3.5.3. Furthermore, we explain our encoding that enables this addition in Section 3.6.2.

3.2 Core Properties

In this section, we provide an overview of the properties that our verification technique ensures. We first start by stating the properties ensured by Necker's design [15]. Then we present our additional properties that ensure the information flow guarantees (**3**).

3.2.1 Properties ensured by Necker's Verification Technique

Recall from Section 2.3, that for a sound verification (verification guarantee **1**) Necker [15] introduced new verification constructs such as hidden and leakable objects, the class invariant and modifiable, respectively non-modifiable fields. In this section, we restate the core properties that Necker's [15] verification technique ensures, so that in the rest of this chapter we can build up on them.

Hidden and Leakable Objects

Recall from Section 2.3, that an object can either be hidden or leakable. An object is hidden if the unverified code cannot have access to it and is leakable otherwise.

(HL) Every object is either hidden or leakable.

Necker [15] forces the user to explicitly make an object leakable by using the `leak` operation before an object can be passed to unverified code.

Since hidden objects cannot be accessed by unverified code, verification can proceed in the standard way.

(H1) A hidden object cannot be accessed by unverified code.
and can be treated as if there was no unverified code.

For leakable objects, on the other hand, the following properties hold. Public fields of leakable objects may be changed by the unverified code at any given

point. Therefore,

(L1) All ***public fields*** of a leakable object
are ***removed from the permission system***.

This means, that once an object becomes leakable, all permissions to its public fields are given up. From now on reads and writes are no longer field accesses, but handled differently:

(L2) Public fields of a leakable object
contain an ***arbitrary leakable*** value at all times.

Property (L2) has as a consequence, that all written values to such a field must be leakable. A read, on the other hand, corresponds to reading an arbitrary leakable value.

Recall from 2.3 that private fields are divided into modifiable fields, which can be modified by unverified code through method calls, and non-modifiable fields, which cannot be modified by unverified code. Therefore, similar to public fields,

(L3) All ***modifiable*** fields of a leakable object
are ***removed from the permission system***.

In order to specify properties of private fields of a leakable object, Necker [15] introduced the class invariant.

(L4) For a ***leakable object***
the class invariant holds at all times.

Since the class invariant specifies a property about the state of the class, it can only refer to constants, class fields and expressions derived from them. Since public fields of a leakable object have an arbitrary value, it does not make sense to specify anything about them in the class invariant.

(CI1) The class invariant only refers to constants,
private fields and expression built from these.

Non-modifiable fields are the only fields that are still part of the permission system. Therefore,

(CI2) The only permissions the class invariant contains
are ***wildcard*** permissions to ***non-modifiable*** fields.

(CI3) The class invariant must be ***self-framing***.

```
1  class RelInv {
2      class_invariant: low(this.f);
3
4      private int f;
5
6      public void relInv(int secret){
7          UnverifiedClass uc = new UnverifiedClass();
8          RelInv obj1 = new RelInv();
9          obj1.f = 1;
10         RelInv obj2 = new RelInv();
11         obj2.f = 2;
12         leak(obj1);
13         leak(obj2);
14         uc.unverifiedMethod(obj1, obj2);
15         RelInv highObj = secret > 0 ? obj1 : obj2;
16         // low(highObj.f) does not hold!
17     }
18 }
```

Figure 3.1: Example of a secure scenario where the relational part of the class invariant of a high leakable object does not hold.

3.2.2 Properties for Information Flow Security

In addition to the properties of Necker [15], we present the properties that are needed for information flow security, that is, verification guarantee **3**.

In order to ensure that the adversary does not have access to secret data, a key property of leakable objects is

(L5) All public fields of a leakable object are ***low*** at all times.

With the addition of information flow security proofs, we added relational assertions: the low assertion and the low event assertion. We support low assertions inside the class invariant. Low event assertions, on the other hand, do not make sense in the invariant.

(CI4) The only relational assertions
a class invariant may contain are ***low assertions***.

With relational assertions introduced, it is possible that a reference-typed expression is high. Following the noninterference formulation (see Section 2.1) this means that in both executions the expression references a ***different actual object***. For both of those objects separately, the class invariant holds. For the high reference-typed expression, on the other hand, the relational parts of the class invariant would ***relate two different objects*** and therefore might not hold.

For an example, consider Figure 3.1. For both objects `obj1` and `obj2` the invariant holds at all times: `f` is low. However, after the simple variable assignment on line 16, with `highObj`, we have a reference-typed expression for which the relational part of the class invariant does not hold, even though it is leakable. For every actual object, the invariant holds. The only reason why `highObj.f` is not low is because the receiver expression `highObj` itself is high.

With the addition of low expressions to the class invariant, (L4) still holds for all ***actual objects***. However, in order to appropriately support class invariants on high reference-typed expressions, we formulate two new properties.

If a reference-typed expression is low, then it refers to the ***same actual object*** in both executions. Thus, it follows from (L4) that the relational assertions hold.

(L4.rel) For a ***low*** leakable reference-typed expression the ***relational part*** of the class invariant holds at all times.

Since unary assertions specify properties for each execution (and thus for each actual object) separately, they also hold for high reference-typed expressions.

(L4.unary) For a leakable reference-typed expression the ***unary part*** of the class invariant holds at all times.

In the following, we will call the whole invariant, that is, both the unary and relational parts of the invariant, the ***standard invariant***.

For example, for the `RelInv` class in Figure 3.1, the unary part of the class invariant is empty, that is, `true`, and the relational part is `low(this.f)`. In this case, the standard invariant is equal to the relational part: `low(this.f)`.

3.3 Target Language

For our encoding, we consider a similar Java subset as Necker [15] with slight adaptations. Our subset contains classes with instance methods and instance fields with all possible Java access modifiers. We support method and constructor calls, if statements, while loops, field assignments, and field reads. However, we have the following constraints for field accesses:

- The receiver of a field access (reading or writing) must be a variable or parameter.
- Field reads must assign directly to a variable or parameter.

That is, `x.f.g = x.f.g + 1;` must be rewritten into

```
MyCls tempF = x.f;
int tempG = tempF.g;
tempF.g = tempG + 1;
```

Similarly, we require method and constructor calls to be directly to a variable. That is, `int x = callMethod() + 1;` is not supported, while `int x = callMethod(); x = x + 1;` is. We furthermore support all kind of Java primitive types, and, as an extension to Necker's target language, arrays. The constraints for array accesses are analogous to those for field accesses:

- The array expression of an array access (reading or writing) must be a variable or parameter.
- Array reads must assign directly to a variable or parameter.

We support all built-in Java operators on primitive types.

Moreover, our design supports subclassing as follows. We allow subclassing between verified classes and, as stated in Section 3.1.1, we allow unverified subclasses of verified classes. However, similarly to Necker [15], we assume that there are no verified subclasses of unverified classes.

Moreover, we support return statements if they are outside constructors.

3.4 Proof Obligations and Guarantees

In this section, we consider every language construct in our Java subset and illustrate with examples how it can lead to an insecure information flow. We then present our proof obligations that ensure that the violating examples are ruled out. Note that we keep the proof obligations and guarantees from Necker [15] in our encoding, but do not explicitly restate them in this section.

In addition to proof obligations, we formulate information flow security guarantees per language construct.

Since we build upon Necker's [15] verification technique, we use concepts from his work that we introduced in Section 2.3. For a reminder of these concepts, refer back to that section or Necker's master thesis [15] for more details. Moreover, the corresponding core properties can be found in Section 3.2.1.

3.4.1 Method Definitions

Public Method Definitions

Recall from Section 2.3 Necker's [15] double verification technique for all public methods. For this, a public method definition is verified twice. The

```
1  class MethodDef {
2      class_invariant: true;
3
4      @modifiable
5      private int f;
6
7      public int getF(){
8          return this.f;
9      }
10
11     public void emptyInvariant(int secret){
12         UnverifiedClass uc = new UnverifiedClass();
13         MethodDef myObj = new MethodDef();
14         leak(myObj);
15         uc.unverifiedMethod(myObj);
16         myObj.f = secret;
17     }
18 }
```

Figure 3.2: Example of a modifiable field with an accessor method but without a low assertion in the class invariant. Leads to a explicit insecure information flow. Prevented by (PO1).

first verification remains as specified by the user and does not need any new proof obligations or guarantees. The second verification is used to model an invocation from unverified code. In this case, we provide some guarantees. Since our technique ensures that the adversary does not have access to secret information, we know that the current object `this`, as well as all parameters, are low. Furthermore, the method invocation can only be a low event, since it originates from an adversary.

(G1.1) In the second verification of a public method the current object `this` is low.

(G1.2) In the second verification of a public method all parameters are low.

(G1.3) In the second verification of a public method the invocation can be assumed to be a low event.

From an information flow perspective, we have to ensure that the method does not reveal secrets when called from unverified code. Consider the example in Figure 3.2. In `unverifiedMethod` the adversary can store the reference to `myObj` and in a new thread repeatadly call the public `getF` method on `myObj`, while returning from the `unverifiedMethod`. After the assignment on line 16 the adversary learns the secret that way. The solution

is the following proof obligation that disallows the method `getF` which leaks a private field with potentially private information.

(PO1) The value returned by the second verification of a public method must be low.

With this rule, `getF` fails the second verification, as it cannot be proven that `this.f` is low. For that, the class invariant would need to be strengthened to `low(this.f)`. With that modification `getF` would verify, since from (H1) and (G1.1) it follows that during the second verification `this` is low and leakable and therefore due to (L4.rel) `low(this.f)` holds. However, now the assignment on line 16 would fail, since it does not maintain the class invariant. More details on the proof obligations for field assignments are given in Section 3.4.6.

Private Method Definitions

Since private methods cannot be called from unverified code, they do not have a second verification. Therefore, there are no proof obligations or guarantees that Necker [15] or we add.

3.4.2 Constructor Definitions

Public Constructor Definitions

The guarantees for the second verification of a constructor definition are analogous to those for method definitions (see Section 3.4.1). However, compared to methods, we can strengthen the first guarantee (G1.1): Since at the beginning of a constructor the current object `this` is new, it cannot depend on a secret. Therefore, we may assume that `this` is low, also at the start of the ***first*** verification of a constructor. Therefore, we formulate (G2.1) stronger than the method definition analogue.

(G2.1) In the ***first and the second*** verification of a public constructor the current object `this` is low.

(G2.2) In the second verification of a public constructor all parameters are low.

(G2.3) In the second verification of a public constructor the invocation can be assumed to be a low event.

Proof obligations are not needed, since a constructor call from unverified code cannot leak any secret information.

```
1 class Explicit {
2     public void explicit(int secret){
3         UnverifiedClass uc = new UnverifiedClass();
4         uc.callMethod(secret);
5     }
6 }
```

Figure 3.3: Example of explicitly leaking a secret by passing it as an argument to an unverified method. Prevented by (PO2).

Private Constructor Definitions

The same logic that applies to the first verification of a public constructor definition also applies to a private constructor definition. Thus, we can formulate a version of (G2.1) for private constructors.

(G3) In a private constructor the current object `this` is low.

3.4.3 Method Invocations

We differentiate two different types of method invocations: Those on a receiver with verified runtime class and those on a receiver with unverified runtime class. Note that for method calls, it is crucial that we consider the ***runtime class*** of the receiver and not the static class. Since, as pointed out in Section 3.1.1, the adversary is able to create unverified subclasses of verified classes, we cannot infer from a receiver with a verified static type that a verified method is actually called. If there is no information on the runtime class of an object, we must treat the object as unverified. In the following, we will refer to a method call as a ***verified method call*** if the receiver's runtime class is known to be verified, and as an ***unverified method call***, otherwise.

Our future proof obligations will ensure that a ***verified method call*** is not visible to the adversary in any way. However, if the receiver's ***runtime class is unverified***, there are multiple ways in which information flow security can be violated. In the following sections, we discuss these ways and provide illustrative examples.

Explicit Insecure Information Flow

The simplest way to violate secure information flow is by directly passing a secret value as an argument to the unverified method call. In the example in Figure 3.3 clearly the adversary learns the value of the parameter `secret`. In order to disallow this, we formulate the following proof obligation:

(PO2) All arguments of an unverified method call must be low.

```
1 class HighEvent {
2     public void highEvent(int secret){
3         UnverifiedClass uc = new UnverifiedClass();
4         if(secret > 0){
5             uc.callMethod();
6         }
7     }
8 }
```

Figure 3.4: Example of implicitly leaking information about a secret value by an unverified method invocation that is not a low event. Prevented by (PO3).

```
1 class HighReceiver {
2     public void highReceiver(int secret){
3         UnverifiedClass uc = new UnverifiedClass();
4         if(secret > 0){
5             uc = new UnverifiedClass();
6         }
7         uc.callMethod();
8     }
9 }
```

Figure 3.5: Example of implicitly leaking information about a secret value by a method invocation on a high unverified receiver. Prevented by (PO4).

Implicit Insecure Information Flow: High Event

In addition to the explicit way, it can also happen that secret information is leaked implicitly.

One way this can happen is, if it depends on a secret value, whether the method is invoked. In the example in Figure 3.4, the unverified method is invoked only if `secret` is greater than zero. The adversary can observe whether the method is called (for example, by setting a flag). In doing so, they can infer whether `secret > 0`. We disallow this kind of information flow with the following rule:

(PO3) An unverified method call must be a low event.

Implicit Insecure Information Flow: High Receiver

The second kind of implicit insecure information flow occurs if the receiver itself is high (that is, secret-dependent). Consider the example in Figure 3.5. The adversary can, inside the constructor, store all created objects (for example, by adding them to a linked list). They can then track the method invocation and distinguish which object it happened on. As in the example from before the adversary then finds out whether `secret > 0`. The following

```
1  class VerifiedClass {
2
3      public void highConstructorCall(int secret){
4          UnverifiedClass uc = new UnverifiedClass();
5          if(secret > 0){
6              new VerifiedClass();
7          }
8          VerifiedClass obj = new VerifiedClass();
9          leak(obj);
10         uc.callMethod(obj);
11     }
12 }
```

Figure 3.6: Example of a high verified constructor call, that, depending on the hashCode implementation, can lead to an implicit insecure information flow. Prevented by (PO5).

proof obligation ensures that this does not happen.

(PO4) The receiver of an unverified method call must be low.

Lowness Guarantee

For unverified method calls, we provide the following security guarantee:

(G4) The result of an unverified method call is low.

The justification is again that the adversary does not have access to high data.

3.4.4 Constructor Invocations

Verified Class

At first sight, a constructor call of a verified class poses little threat to information flow security. However, depending on how the `hashCode` method of the corresponding class is implemented, a high event constructor call could still lead to secret leaks. Specifically, if a class does not override the `hashCode` method, `Object.hashCode` is used. This is a native method, and its implementation is JVM-dependent. Since the Java language rules [16] do not specify a concrete implementation, valid implementations include calculations on the object's memory location or simply increasing a counter for every new object. Consider the class in Figure 3.6 and assume that `hashCode` is implemented as a simple counter starting from 0. In this scenario, the adversary can call the `hashCode` method on `obj` and depending on whether the result is equal to 1 or 2 find out if the constructor on line 5 was called. Thus, they can infer if `secret` is greater than zero. We want our verification technique to be sound for any possible `hashCode` implementation. Therefore, we formulate the following proof obligation, to reject examples

similar to the one in Figure 3.6.

(PO5) A verified constructor call must be a low event.

Unverified Class

The problem illustrated in Figure 3.6 clearly also exists if the constructor calls on lines 5 and 7 were to constructors of unverified classes. Thus, we need a proof obligation analogous to (PO5) for unverified constructor calls.

Additionally, we have proof rules similar to those for unverified method invocations (see Section 3.4.3). In fact, the examples in that section can be seen as direct analogues obtained by replacing the method invocation with a constructor call. The only exception is rule (PO4): a constructor invocation does not have a receiver that could be high. Due to the close similarity, we omit the examples in this section and only present the resulting rules.

(PO6) All arguments of an unverified constructor call must be low.

(PO7) An unverified constructor call must be a low event.

(G5) The result of an unverified constructor call is low.

Note that Rule (PO7) is the unverified analogue of Rule (PO5) for verified constructor calls.

3.4.5 Field Reads

Field reads are not observable for the adversary, regardless of whether they occur on an object with verified or unverified runtime class. Hence, there is no way how information flow security could be violated, and therefore there are no new proof obligations for field reads. From property (H1) it follows that field reads of a hidden receiver can be left untouched. However, for reads with a leakable receiver, we provide new information flow guarantees.

Public Field Read

Recall that public field reads of leakable objects are treated as reading an arbitrary value that is leakable (see (L2)). With relational specifications introduced, we can add another guarantee: If the receiver of the field access is low, then so is the read value. The justification is similar to the one for (G4): By the definition of information flow, all values or objects accessible to the adversary are low. However, this only holds if the receiver is low. To

```
1  class PublicRead {
2      private int secretField;
3      public int f;
4
5      public void highReceiver(PublicRead obj1, PublicRead obj2){
6          UnverifiedClass uc = new UnverifiedClass();
7          PublicRead myObj = obj1.secretField > 0 ? obj1 : obj2;
8          int value = myObj.f;
9          uc.callMethod(value);
10     }
11 }
```

Figure 3.7: Example of a public field read with a leakable high receiver. Assuming that the read value is low would lead to an explicit insecure information flow. Prevented by (G6).

illustrate why, consider the example in Figure 3.7. An adversary could call the `highReceiver` function with `obj1` and `obj2` that have different values in `f`. Observing the call to the unverified `callMethod` method, they can then learn whether `obj1.secretField > 0`. Since `secretField` is private, it may contain secret data even when the object is leakable. Thus, we have found an insecure information flow.

(G6) A value read from a public field of a leakable object is low if the receiver is low.

Modifiable Field Read

After we have discussed public field reads, what remains are private field reads. We start by covering modifiable fields. Recall that similar to public fields, modifiable fields are removed from the permission system (see (L3)). Therefore, following Necker's [15] design, reading from a modifiable field is equivalent to reading an arbitrary value for which the class invariant holds. Recall from Section 3.2, that we have introduced new properties about the class invariant for reference-typed expressions. One for the unary (L4.unary) and one for the relational part (L4.rel). The latter holds only if the reference-typed expression is low. Thus, we provide the following guarantees.

(G7.1) A value read from a modifiable field of a leakable object satisfies the unary part of the class invariant.

(G7.2) A value read from a modifiable field of a leakable object satisfies the ***relational part*** of the class invariant ***if the receiver is low***.

This prevents insecure information flows such as in the example in Figure 3.8. The value read from `myObj.f` is not low, as `myObj` itself is high. A similar ad-

```
1  class ModRead {
2      class_invariant: low(this.f);
3
4      private int secretField;
5
6      @modifiable
7      private int f;
8
9      public void highReceiver(ModRead obj1, ModRead obj2){
10         UnverifiedClass uc = new UnverifiedClass();
11         ModRead myObj = obj1.secretField > 0 ? obj1 : obj2;
12         int value = myObj.f;
13         uc.callMethod(value);
14     }
15 }
```

Figure 3.8: Example of a modifiable field read with a leakable high receiver. Assuming the class invariant after the field read would lead to an explicit insecure information flow. Prevented by transforming Necker's guarantee into (G7.1) and (G7.2).

versary as described for the example in Figure 3.7 learns if `obj1.secretField > 0`.

Non-Modifiable Field Read

The last type of field we have not discussed yet are non-modifiable fields. Recall that these fields are still part of the permission system, even if the object is leakable. Except for that, non-modifiable field reads can be treated exactly as modifiable fields. The only change is that the read value is not an arbitrary value, but the field can be read normally, since it is still part of the permission system. That is, the usual proof obligations regarding permissions still remain. The motivating example and our new sound guarantee are analogous to the modifiable case.

(G8.1) A value read from a non-modifiable field of a leakable object satisfies the unary part of the class invariant.

(G8.2) A value read from a non-modifiable field of a leakable object satisfies the ***relational part*** of the class invariant ***if the receiver is low***.

3.4.6 Field Assignments

As long as the adversary does not have access to an object, no secret information can be leaked through field assignments and no invariant has to be maintained for that object. Therefore, similar to the section on field reads, we only need to add proof obligations for leakable receiver objects. Again

```
1  class PublicWrite {
2
3      public int f;
4
5      public void highValue(int secret){
6          UnverifiedClass uc = new UnverifiedClass();
7          PublicWrite myObj = new PublicWrite();
8          leak(myObj);
9          uc.unverifiedMethod(myObj);
10         myObj.f = secret;
11     }
12 }
```

Figure 3.9: Public field write to a leakable object. Leads to an explicit insecure information flow. Prevented by (PO8).

we will start with the public case and then move on to the modifiable and non-modifiable private case.

Public Field Write

Once the adversary has access to an object, they can continuously check the contents of its public fields. Therefore, there are multiple types of insecure information flow. They fall into the same categories as those in Section 3.4.3 on method invocations: explicit, implicit through a high event, and implicit through a high receiver.

The simplest is the ***explicit insecure information flow***, where a secret value is stored in a public field of a leakable object. Consider the example in Figure 3.9. After the call to the unverified method, the adversary has access to `myObj`. Thus, in a different thread, they can simply read `myObj.f` and learn the secret value. (PO8) prevents this information flow.

(PO8) The value written to a public field of a leakable object must be low.

We also have to avoid the two types of implicit insecure information flow. For the first, consider the example in Figure 3.10. In this scenario, the adversary can find out if `secret > 0` by continuously reading `myObj.f` and checking whether at some point the value is equal to 42. The resulting proof obligation is the following.

(PO9) A write to a public field of a leakable object must be a low event.

A similar adversary also finds out `secret > 0` in the other type of implicit insecure information flow. For that, consider the example in Figure (3.11).

```
1  class PublicWrite {
2
3      public int f;
4
5      public void highEvent(int secret){
6          UnverifiedClass uc = new UnverifiedClass();
7          PublicWrite myObj = new PublicWrite();
8          leak(myObj);
9          uc.unverifiedMethod(myObj);
10         if(secret > 0){
11             myObj.f = 42;
12         }
13     }
14 }
```

Figure 3.10: Public field write to a leakable object. Leads to an implicit insecure information flow, since the assignment is not a low event. Prevented by (PO9).

```
1  class PublicWrite {
2
3      public int f;
4
5      public void highReceiver(int secret){
6          UnverifiedClass uc = new UnverifiedClass();
7          PublicWrite myObj1 = new PublicWrite();
8          PublicWrite myObj2 = new PublicWrite();
9          leak(myObj1);
10         leak(myObj2);
11         uc.unverifiedMethod(myObj1);
12         uc.unverifiedMethod(myObj2);
13         PublicWrite receiver = secret > 0 ? myObj1 : myObj2;
14         receiver.f = 42;
15     }
16 }
```

Figure 3.11: Public field write to a leakable object. Leads to an implicit insecure information flow, since the receiver is not low. Prevented by (PO10).

```
1  class ModWrite {
2      class_invariant: low(this.f);
3
4      @modifiable
5      private int f;
6
7      public int getF(){
8          return this.f;
9      }
10
11     public void highEvent(int secret){
12         UnverifiedClass uc = new UnverifiedClass();
13         ModWrite myObj = new ModWrite();
14         leak(myObj);
15         uc.unverifiedMethod(myObj);
16         if(secret > 0){
17             myObj.f = 42;
18         }
19     }
20 }
```

Figure 3.12: Modifiable field write to a leakable object. Leads to an implicit insecure information flow, since the assignment is not a low event and the field has a getter. Prevented by (PO11).

The problem is the same as in Figure 3.10. Whether `myObj1.f` (respectively `myObj2.f`) gets set to 42 depends on a secret value. Therefore,

(PO10) The receiver of a public field assignment must be low.

Modifiable Field Write

Necker's [15] verification technique ensures that after a modifiable field write the invariant still holds. With our new invariant guarantees (L4.unary) and (L4.rel) we run into a problem. We consider two examples that illustrate the reasons why with low expressions in the class invariant it is not enough to

1. require the invariant to hold in the current branch.
2. require the invariant to hold for the receiver expression of the assignment.

Instead, the class invariant has to hold ***for each actual object***.

The first problem can be seen in the example in Figure 3.12. The `ModWrite` class is reminiscent of the `MethodDef` class in Figure 3.2. In the corresponding Section 3.4.1, we have seen that because of the getter method `getF`, the field `f` can be accessed by the adversary. Therefore, a high event assignment as on line 17 must be prevented. Otherwise, the adversary can find out if `secret > 0` depending on whether `myObj.getF()` returns 42, at some point after a call to `highEvent`. When requiring the normal class invariant to

```
1  class ModWrite {
2      class_invariant: low(this.f);
3
4      @modifiable
5      private int f;
6
7      public int getF(){
8          return this.f;
9      }
10
11     public void highReceiver(int secret){
12         UnverifiedClass uc = new UnverifiedClass();
13         ModWrite myObj1 = new ModWrite();
14         ModWrite myObj2 = new ModWrite();
15         leak(myObj1);
16         leak(myObj2);
17         uc.unverifiedMethod(myObj1);
18         uc.unverifiedMethod(myObj2);
19         ModWrite receiver = secret > 0 ? myObj1 : myObj2;
20         receiver.f = 42;
21     }
22 }
```

Figure 3.13: Modifiable field write to a field leakable object. Leads to an implicit insecure information flow, since the receiver is not low and the field has an accessor method. Prevented by (PO11).

hold after line 17 the assignment passes verification. Recall how lowness is formulated as a relational noninterference assertion: An expression is low if both activation variables being true implies that the values of both expressions are the same. This holds on line 17: If both executions take the if-branch, then the `receiver.f` is updated in both executions to 42. If one of the executions does not take the branch, then the low assertion holds vacuously at line 17.

For the second problem, consider a scenario in which the receiver of the field assignment is high. Following the noninterference formulation, this means that we have two different objects in the two executions. This leaves us with two class invariants that we want both to hold after the assignment. For unary assertions, the proof obligation with one invariant is fine, since the noninterference technique checks the assertion for both versions of the receiver separately. However, as soon as the class invariant contains low assertions, this is not enough. Consider the example in Figure 3.13. The example is similar to the previous one (Figure 3.12). Only this time the assignment has a high receiver instead of being a high event. The outcome is similar: After the field assignment on line 20 `low(receiver.f)` holds, since this time both activation variables are active, and 42 is low. What we would rather like to check is whether for each individual object, the field remains

low. That is, we observe that `myObj1` has the same values for `f` before line 20, regardless of whether `secret > 0`. After line 20 this is no longer the case: If `secret > 0` then `myObj1.f` is equal to 42. Otherwise, it is equal to 0.

The easiest solution to prevent those insecure information flows would be to only allow field writes on low receivers and require them to be a low event. However, this is incomplete. As an example, consider a modifiable field that is not required to be low in the invariant. In this case, we want to allow high event and high receiver assignments as they cannot lead to insecure information flows.

As a next solution one could think of disallowing high event and high receiver assignments only for fields that are specified low in the class invariant. This way, the previously mentioned scenario would be allowed. However, generally this is non-trivial, since the invariant can contain predicates in which the low assertion might be hidden. Moreover, it would still be incomplete. To see why, consider the example in Figure 3.14. The assignment on line 28 is on a high receiver and to a modifiable field which is specified as low in the invariant. However, because the read to the non-modifiable copy `copyF` of `f` is guaranteed to return the value of `f`, after the subsequent assignment to `f` the invariant still holds for ***both actual objects*** `myObj1`, `myObj2` (see (L4)). Furthermore, (L4.unary) and (L4.rel) hold for all expressions of type `ModWrite`: `myObj1`, `myObj2` and `receiver`. Note that an analogous example can be created for the scenario of a high event field assignment.

Thus, we allow assignments that are a high event or on a high leakable receiver if after the assignment the standard invariant still holds for ***all actual objects***. In this case, we denote the assignment as ***invariant-preserving***.

In order to check whether an assignment is invariant-preserving, we introduce the ***split-invariant***. The split-invariant is a transformed version of the standard class invariant. Similarly to (L4.unary) and (L4.rel) our technique guarantees that for a leakable expression the split-invariant holds at all times.

(SI1) For a leakable reference-typed expression the split-invariant holds at all times.

The idea behind the split-invariant is to transform a low assertion in the class invariant in such a way that after an assignment it can be checked if the lowness is preserved for the ***actual receiver object*** of ***every execution***. This then implies the invariant-preserving property for the assignment.

(SI2) If the split-invariant holds after an assignment the assignment is invariant-preserving.

```
1  class ModWrite {
2      class_invariant: Perm(fCopy, read) ** fCopy == f ** low(f);
3
4      @modifiable
5      private int f;
6
7      private int fCopy;
8
9      public ModWrite(int f){
10         this.f = f;
11         this.fCopy = f;
12     }
13
14     public int getF(){
15         return this.f;
16     }
17
18     public void highReceiver(int secret){
19         UnverifiedClass uc = new UnverifiedClass();
20         ModWrite myObj1 = new ModWrite(1);
21         ModWrite myObj2 = new ModWrite(2);
22         leak(myObj1);
23         leak(myObj2);
24         uc.unverifiedMethod(myObj1);
25         uc.unverifiedMethod(myObj2);
26         ModWrite receiver = secret > 0 ? myObj1 : myObj2;
27         int fValue = receiver.fCopy;
28         receiver.f = fValue;
29     }
30 }
```

Figure 3.14: Secure example of an assignment on a high receiver to modifiable field which is specified low in the invariant.

We present the concrete split-invariant transformation in Section 3.6.7 where we discuss our encoding.

Note that the problems depicted in Figures 3.12 and 3.13 occur only if the assignment is a high event or the receiver is high. If the field assignment has a low receiver and is a low event, we can use the standard class invariant as in Necker's [15] design. With the concept of the split-invariant introduced, we formulate the updated proof obligation for modifiable field writes.

(PO11) If a modifiable field write is a high event or has a high receiver the split-invariant must hold after the assignment.
Otherwise, the standard class invariant must hold.

Non-Modifiable Field Write

As in Section 3.4.5 on field reads, non-modifiable fields are similar to modifiable fields. The only difference is again that non-modifiable fields remain in the permission system when the receiver is leakable. That is, in this case, additionally, there are the proof obligations from the permission systems. However, our information flow security proof obligations are not affected by this.

(PO12)
If a non-modifiable field write is a high event or has a high receiver
the split-invariant must hold after the assignment.
Otherwise, the standard class invariant must hold.

3.4.7 Leak Operation

Recall from Section 2.3, that the leak operation is used to make a hidden object leakable. Therefore, Necker [15] introduces proof obligations for the leak operation that establish properties (L1) – (L4). We keep those proof obligations, but we do not restate them in this section. Rather, we only present our new information flow security proof obligations that ensure properties (L5), (L4.unary) and (L4.rel).

To establish (L5), we formulate the following proof obligation.

(PO13) The leak operation requires that
all public fields of the given object are low.

Proof obligation (PO13) prevents examples as the one in Figure 3.15, where the adversary could in the `unverifiedMethod` simply read the secret from the public field `f`.

In order to establish (L4), respectively (L4.unary) and (L4.rel), we additionally require the leak operation to be a low event and the object to be low. This way, we avoid complex situations, as in the case for private field assignments, where low assertions in the invariant can be vacuously true.

(PO14) The leak operation requires that the given object is low.

(PO15) The leak operation requires that it is a low event.

Unlike in the case for private field assignments, for the leak operation, these obligations do not make our approach incomplete. The reason is that an expression to be passed to unverified code must be low either way. Moreover,

```
1  class LeakOperation {
2
3      public int f;
4
5      public void publicFieldExplicit(int secret){
6          UnverifiedClass uc = new UnverifiedClass();
7          LeakOperation myObj = new LeakOperation();
8          myObj.f = secret;
9          leak(myObj);
10         uc.unverifiedMethod(myObj);
11     }
12 }
```

Figure 3.15: An explicit insecure information flow, caused by leaking an object with a public field containing a secret. Prevented by (PO13).

leaking a new object to unverified code must always be a low event. Therefore, requiring both low event and lowness already during the leak operation does not negatively affect the completeness of our technique.

With our previous proof obligations (PO14) and (PO15), we know that the object is low and both executions are active. Thus, we do not run into the problems that we have seen for private field assignments. The leak operation can simply check the standard invariant. Therefore, this proof obligation remains unchanged compared to Necker's design [15]. For clarity, we nevertheless restate it here.

(PO16) The leak operation requires the ***standard invariant*** to hold.

Note that a similar example to the one in Figure 3.15 could be thought of, where `f` is a private field with a getter instead of a public field. As we have seen for the example in Figure 3.2, proof obligation (PO1) requires the class invariant to include `low(this.f)`, in this scenario. Therefore, proof obligation (PO16) prevents such an example.

3.4.8 Covert Channels

With these proof obligations, we successfully detect explicit and implicit insecure information flows. We have not yet considered what the adversary can learn from timing and termination channels. In this section, we provide proof obligations for preventing such covert channels.

Volpano and Smith [23] distinguish between ***internal*** and ***external*** timing channels. ***External timing channels*** arise when the execution time of a program depends on a secret and the adversary can ***measure this time***. ***Internal timing channels*** are introduced when such secret-dependent time differences are visible in the program state. For that consider the example in

```
1  class CovertChannel {
2
3      public int f;
4
5      public void run1(){
6          for(int i=0; i < 20; i++){
7          }
8          this.f = 1;
9      }
10
11     public void run2(int secret){
12         for(int i=0; i < secret || secret == -1; i++){
13         }
14         this.f = 2;
15     }
16
17     public void main(int secret){
18        Thread t1 = new Thread(() -> this.run1());
19        Thread t2 = new Thread(() -> this.run2(secret));
20        t1.start();
21        t2.start();
22     }
23 }
```

Figure 3.16: Example of an internal timing channel, that would be prevented by the permission system.

Figure 3.16, where the two methods `run1` and `run2` run in parallel. Clearly, the execution time of `run2` depends on a secret. Even worse: if `secret == -1` then `run2` does not terminate. Depending on what value `secret` has, the probability of which thread finishes first changes and, with that, the probability of field `f` having value 1 or 2. Even though a data race as in the main method would be prevented by the permission system, our verification technique would still allow similar side channels. The adversary is free to spawn additional threads and can, therefore, call the method `run1` in parallel to calling `run2`. Furthermore, if the write accesses to `f` are guarded by a lock, then a similar program as in Figure 3.16 respects the permission system, while still introducing an internal timing channel.

Since our adversary also has the capabilities to call public methods and measure their execution time, we are also susceptible to external timing channel and termination attacks. The adversary can call the method `run2` and then continuously read the value of the public field `f`. They can differentiate the cases where it takes a small amount of time, a large amount of time, or the public field never gets assigned the value 2. This way, the adversary can infer information about the high input `secret`.

We can prevent internal and external timing channels, as well as termination channels by taking the same approach as Eilers et al. [6] and other previous

work: By disallowing high branching, we force every execution with the same low inputs to take the exact same execution path. If two executions take the exact same execution path, then both execution times are the same, and either both executions terminate or do not.

(PO17) All loop conditions and if conditions must be low.

As Eilers et al. [6] remark, we must also disallow implicit branches. That is, we must prevent dynamically-bound calls, where the method called at runtime depends on high data.

(PO18) For every method call,
the runtime class of the receiver must be low.

With those two proof obligations, we essentially ensure that secrets do not influence what execution paths a program takes. In other words, every point in the program is a low event. This means that the obligations (PO17) and (PO18) imply that all previous proof obligations that required a statement to be a low event are satisfied. We can therefore remove the following rules: (PO3), (PO5), (PO7), (PO9), (PO15), (G1.3), (G2.3). Furthermore, (PO11) and (PO12) can be simplified.

Note that even without high branching, it is still possible to have high reference-typed expressions, for example, due to the use of arrays (see Section 3.5.2). Therefore, we have to keep the rules that require a low receiver.

3.5 Extensions

In this section we present additions to Necker's [15] verification technique on top of information flow security.

3.5.1 Access Modifiers

The only access modifiers that Necker's technique supports are `public` and `private`. We add support for the remaining two: `protected` and `package-private` (default). Both must be treated exactly the same way as `public`. In this section, we explain why.

Protected and package-private can both be accessed from code within the same package. Although this might seem as if this would protect against access from unverified code, since that part of the code will certainly not be running in the same folder as the verified code, this is not the case. The adversary can, in his own module, declare a package with the same name as the one containing the target class. For Java, these two packages are the same. Therefore, the adversary can access protected and package-private code in the verified code base.

3.5.2 Arrays

As a further extension to Necker's [15] verification technique, we add support for arrays.

Arrays can be seen as classes without private fields, and hence without class invariant. Every array location can be seen as a public field. However, there are some more restrictions. In the following, we provide the proof obligations and guarantees for arrays. These include both rules for the hidden/leakable system and information flow security related rules. Furthermore, we present a version of the leak operation for arrays.

Constructor

We only support the array constructor without instantiation. That is, the one of the form `new int[length]`. We get simple guarantees that follow the logic for object constructors (see Section 3.4.2): A newly created array cannot be high.

(G9.1) A newly created array is low.

Furthermore, we know that the adversary cannot have access to a newly created array.

(G9.2) A newly created array is hidden.

Since in Java arrays are objects, the same reasoning about the `hashCode` method for constructor invocations (see Section 3.4.4) also applies to arrays. Thus, we would need an analogue to the proof rule (PO5) that requires an array constructor call to be a low event. However, as we have seen in Section 3.4.8, the rules (PO17) and (PO18), which prevent high branching, imply that every point in the program is a low event. We can therefore omit the corresponding proof obligation.

Array Writes

For writes to leakable arrays, we have to require that the index is low, in order to prevent implicit secret leakage.

(PO19) The index of a leakable array write must be low.

The other proof obligations are similar to the ones for writes to a public field of a leakable object.

(PO20.1) The value written into a leakable array must be low.

(PO20.2) If the value written into a leakable array is of non-primitive type it must be leakable.

(PO21) An array write of a leakable array must be a low event.

The only difference is that an array write does not have a receiver. As an analogue to (PO10) we require the array in question to be low.

(PO22) The array expression affected by a leakable array write must be low.

Array Reads

For array reads, the guarantees are similar but slightly weaker than in the case of a public field read (see (G6)). Clearly, we only learn that the read value is low if in addition to the array, the index is low too.

(G10.1) A value read from a leakable array is low if the array ***and the index*** are low.

Since the adversary has access to every element of a leakable array, each element itself is also leakable.

(G10.2) A value read from a leakable array is leakable.

Length

The length of an array is a special case: Its value can be read, but cannot be changed by the adversary. Therefore, reading the length of a leakable array is the same as for a hidden array.

We have to ensure that we do not leak a secret through the length of an array that we pass to unverified code.

Leak Operation for Arrays

We ensure this by adding a proof obligation for the leak operation.

(PO23) The leak operation requires that the length of a given array is low.

Except for this, the proof obligations for the array leak operation are similar to those for objects:

(PO24) The leak operation requires that the given array expression is low.

Since (PO15) remains exactly the same, we will not restate it for the array version. The analogue of (PO13) is

(PO25.1) The leak operation requires that all values inside the given array are low.

In addition to that, similar to property (L2) for objects,

(PO25.2) The leak operation requires that all values inside the given array are leakable.

Furthermore, as with public fields of a leakable object, the leak operation removes all array locations from the permission system (see property (L1)).

(PO26) The leak operation removes all locations of the given array from the permission system.

3.5.3 Weak Method Specifications

Recall from Section 3.1.2, that we provide the adapted verification guarantee **2**. That is, we allow a verified method to crash if it is called from unverified code and the crash does not leak information about high inputs.

We do this by giving the user the capability to specify two sets of pre- and postconditions. In addition to the standard ones we allow the user to formulate a set of weaker pre- and postconditions to specify the behavior of the method when called transitively from unverified code. The stronger standard pre- and postconditions guarantee crash-freedom and information flow security, whereas the weaker ones only need to guarantee information flow security. If a method call initially originates from verified code, the strong specifications will be used such that crash-freedom is ensured, whereas during calls originating transitively from unverified code, we may use the weak specifications that only ensure that no secrets are leaked.

Recall the double verification of methods discussed in Sections 2.3.3 and 3.4.1. The second verification corresponds to verifying the method for the case that it is called from unverified code and therefore cannot rely on any user-provided pre- and postconditions. In this second method verification, our approach diverges from Necker's technique: Whenever a method or constructor is invoked, we use the ***weak specifications***. For all subsequent transitive method or constructor calls within the method invoked, ***weak specifications*** are used. Only during the first verification, we use the strong specifications to ensure crash-freedom for methods called from verified code.

Note that in addition to user-defined methods, the two sets of method specifications can be implemented for the built-in operations of the underlying

```
1 class Division {
2
3     // weak-requires low(b != 0);
4     // strong-requires b != 0;
5     // strong-ensures \result == a / b;
6     public int div(int a, int b);
7
8 }
```

Figure 3.17: Example of weak and strong method specifications for integer division.

verification language. For example, for division, it is usually required that the divisor is nonzero. The two sets of methods specifications for integer division could look as depicted in Figure 3.17. The weak precondition `low(b != 0)` is needed, also when called from unverified code, in order to ensure that a crash does not leak information about high inputs. If we have a public method that divides some value by a secret integer, the adversary could, by calling this method and depending on whether the method crashes or not, infer whether `secret == 0`.

The weak preconditions depend on the error thrown by the operation or method. For the example in Figure 3.17, if the zero division error returned information about the dividend `a`, an additional weak precondition `weak-requires b == 0` $\implies$ `low(a)` would be needed. This way, no high information is leaked by the error.

Note that the concept of weak and strong specifications can also be applied to even more primitive built-in operations, such as field accesses or method invocations. The strong variant would typically require the receiver to be non-null, whereas for the weak specifications `low(receiver` $\neq$ `null)` suffices.

3.6 Encoding

In this section, we present the VerCors encoding that adds our proof obligations and guarantees, as well as a slightly adapted version of Necker's [15] proof rules. By discussing runtime class tracking and weak method specifications, we first cover general properties of our encoding. Then we clarify the notation used for our encoding, before we present and explain our encoding for selected language constructs. Finally, we provide the encoding for the remaining cases.

3.6.1 Runtime Class Tracking

We assume that our underlying verification infrastructure tracks the runtime classes of objects and can reason about them. As discussed in Section 3.4.3,

this is needed to distinguish verified from unverified method calls and additionally for encoding proof obligation (PO18).

3.6.2 Weak Method Specifications

In this section we briefly present how we encode the two sets of method specifications, discussed in Section 3.5.3. We explain the encoding for methods, but for constructors the encoding is identical. For the sake of readability, we limit the discussion to methods.

In order to support two sets of method specifications, we introduce a boolean ghost parameter `fromUC`, a parameter only needed for the proof (see Filliâtre et al. [11]). This ghost parameter is added to every method and evaluates to true iff the method is currently transitively called from unverified code.

Since the second verification of a public method corresponds to the case where the method is called directly by the adversary, we want `fromUC` to be true.

(E1) In the second verification the ghost parameter `fromUC` is `true`.

During a method invocation, regardless of whether it occurs in the first or second verification, the ghost parameter is passed on.

(E2) During a method or constructor invocation
the ghost parameter `fromUC` is passed on.

This way, whenever a method is called from unverified code, directly or indirectly, it always holds that `fromUC` is true.

With this parameter, the user can then specify the pre- and postconditions as logical implications. A strong specification can be specified as `!fromUC` $\implies$ *spec*. Since weak specifications must usually hold in both cases, if called from unverified or verified code, they can be specified in the usual way *spec*. However, it is also sound to specify a "weak" specification that holds only if called from unverified code: `fromUC` $\implies$ *spec*. In that case, the "weak" specification is not necessarily implied by the strong specification and thus might not actually be weaker.

For the division example in Figure 3.17 the specifications could be encoded as depicted in Figure 3.18. Our encoding declares parameter `fromUC` for every method and constructor, and passes it on as specified in this section.

3.6.3 Notation

In the following, we denote $[\![s]\!]_C$, the encoding of language construct s within class C. We assume that for every verified class C there is a class invariant

```
1 class Division {
2
3     // requires low(b != 0);
4     // requires !fromUC ⟹ b != 0;
5     // ensures !fromUC ⟹ a / b;
6     public int div(int a, int b);
7
8 }
```

Figure 3.18: Example how the ghost parameter could be used to specify the two sets of method specifications for integer division.

and a set of modifiable fields. We denote $C.mod$ the set of modifiable fields and $C.inv$ the class invariant of C. We write $C.fields$ for all fields of class C and $C.public\_fields$ for all non-private fields (that is, public, protected and package-private). With `x.class` we denote the ***static compile time class*** and with `x.getClass()` the ***runtime class*** of x. `primitive_types` denotes the set of all types that are not of class type. Note that this does not include array types. For an array `a`, we use `a.elem.class` to denote the static type of its elements. Furthermore, we use square brackets $[a/b]$ to denote substitution of all terms a with b. For example, `(low(this) ∧ low(this.f))[this/x]` = `low(x) ∧ low(x.f)`. We use $\wedge$ as the separating conjunction. Moreover, we use the universal quantifier $\forall$ to write multiple statements or multiple substitutions.

For clarity, parts of the encoding stemming from Necker [15] are depicted in *black*, whereas our contributions appear in *blue* and are annotated with the corresponding rule, if applicable.

3.6.4 Method Declarations

Figure 3.19 shows the encoding of a public method `mn`. The double verification is encoded by ***doubling the method declaration***.

The first method declaration `mn_h` corresponds to the case where `mn` is called from verified code. Thus, the pre- and postconditions for this method remain as specified by the user and no proof obligations or guarantees are added to the method body. The only thing our encoding adds is the ghost parameter `fromUC` in order to support weak method specifications.

The second method `mn_l` corresponds to the scenario where `mn` is directly called from unverified code. Therefore, there are no preconditions that can be assumed to hold. The user-specified postconditions are removed. Instead, we use Necker's [15] postcondition that requires the result to be leakable, and additionally, we add a postcondition that ensures our proof obligation (PO1). In the body of the method, we encode (E1) by assigning `fromUC` the value

$[\![am\ C_{ret}\ mn(C_1\ arg_1, \ldots, C_n\ arg_n)\ spec\ \{s_1, \ldots s_k\}]\!]_C$
where $am \in \{\texttt{public}, \texttt{protected}, \epsilon\}$
$:=$ $am\ C_{ret}\ mn\_h(C_1\ arg_1, \ldots, C_n\ arg_n,\ \texttt{boolean}\ fromUC)\ spec$
$\{[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C\}$

$am\ C_{ret}\ mn\_l(C_1\ arg_1, \ldots, C_n\ arg_n,\ \texttt{boolean}\ fromUC)$
ensures $C_{ret} \notin$ primitive_types $\implies$
Perm(leakable(\result), write)
ensures low(\result) (PO1)
{
$fromUC$ = true (E1)
inhale Perm(leakable(this), write)
assume low(this) (G1.1)
$\forall i \in \{1, \ldots, n\}$
inhale arg_i.class $\notin$ primitive_types
$\implies$ Perm(leakable(arg_i), write)
assume low(arg_i) (G1.2)
$[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C$
}

Figure 3.19: Encoding of a public method declaration.

true. In order to encode the remaining rules from Section 3.4.1, we transform the guarantees (G1.1) and (G1.2) into corresponding assume statements.

3.6.5 Leak Operation

Next, we discuss the encoding of the leak operation., which is shown in Figure 3.20. The majority of the encoding is part of Necker's [15] work. The leak operation checks property (HL), that is, that the object x is either leakable or hidden. If x is not yet leakable, the transfer from leakable to hidden is made by exhaling and inhaling the corresponding predicates. Afterwards, Necker's [15] encoding establishes property

- (L1) by exhaling all permissions to public fields.
- (L2) by asserting that all public fields are leakable.
- (L3) by exhaling all permissions to modifiable fields.
- (L4) by exhaling the class invariant.

Note that here exhaling the class invariant already covers our proof obligation (PO16). For the remaining proof obligations from Section 3.4.7, (PO14) and (PO13), we add corresponding assert statements: We assert that the object `x` and the values of its public fields `x.f'` are low. (PO15) can be omitted due to (PO17) and (PO18).

$[\![$`leak` $x]\!]_C$
  where x is not an array
:= `assert perm(leakable(`x`)) > 0 || perm(hidden(`x`)) > 0`
  `if(perm(leakable(`x`)) == 0) {`
    `exhale Perm(hidden(`x`), write)`
    `inhale Perm(leakable(`x`), write)`
    `assert low(`x`)` (PO14)
    `exhale` $C.inv$ (PO16)
    $\forall f \in$ `x.class.`mod, `exhale Perm(`$x.f$`, write)`
    $\forall f' \in$ `x.class.`$public\_fields$,
      `assert low(`$x.f'$`)` (PO13)
      `assert` $x.f'$`.class` $\notin$ `primitive_types` $\implies$
        `Perm(leakable(`$x.f'$`), wildcard)`
      `exhale Perm(`$x.f'$`), write)`
  }

Figure 3.20: Encoding of the leak operation.

3.6.6 Modifiable Field Reads

Another interesting case are modifiable field reads. For the corresponding encoding, see Figure 3.21. If the receiver `x` is hidden, the else branch is taken and the field is read in the standard way according to the permission system. However, if `x` is leakable, `fn` is no longer part of the permission system. Instead, for every modifiable field, Necker [15] declares a new variable. According to our guarantee (G7.1), we may inhale the unary part of the class invariant with all modifiable fields substituted with their corresponding fresh variable and with the receiver object `this` of the invariant replaced by x. This way, when reading `fn_temp`, the new variable corresponding to `x.fn`, we get an arbitrary value for which the invariant holds. Following (G7.2), if `x` is low, we may assume the relational part as well.

Note that in the leakable case the encoding no longer contains an actual field read. Thus, we have to explicitly check the preconditions of a field read. These are, as discussed in Section 3.5.3, if we are originating from verified

$[\![y = x.fn]\!]_C$
  where fn is a private field and $fn \in C.mod$
$:=$ `assert perm(leakable(`x`)) > 0 || perm(hidden(`x`)) > 0`
  `if(perm(leakable(`x`)) > 0) {`
    `assert low(`$x \neq$ `null)`
    `assert !fromUC` $\implies x \neq$ `null`
    $\forall f \in C.mod$, `f.class` f_temp;
    `inhale unary(`$C.inv$`)`$[\forall f' \in C.mod,\ this.f'/f'\_temp][this/x]$ (G7.1)
    `assume low(`x`)` $\implies$
      `rel(`$C.inv$`)`$[\forall f' \in C.mod,\ this.f'/f'\_temp][this/x]$ (G7.2)
    $y = fn\_temp$
  } `else {`
    $y = x.fn$
  }

Figure 3.21: Encoding of modifiable field reads.

code that `x` $\neq$ `null` and that this expression is low otherwise. In the else branch, we do not have to add these assert statements, as they are implied by the permission check.

3.6.7 Modifiable Field Writes

In the modifiable field writes encoding we use the ***split-invariant*** that we have introduced at a high level in Section 3.4.6. We denote the split-invariant of class C with substituted receiver x and substituted modifiable fields as $\langle\langle C.inv \rangle\rangle^{x}_{\{(f', f'\_new, f'\_old) \mid f' \in C.mod\}}$. We will cover this construct in a bit. First, we discuss the other parts of the encoding. For that consider Figure 3.22.

Similarly to the leak operation and modifiable field reads, it is first checked that the receiver `x` is hidden or leakable. Again similar to field reads, if `x` is hidden, the encoding is equivalent to a standard field write. If `x` is leakable, the field access preconditions are checked and the encoding follows proof obligation (PO11): We make a case distinction on whether the receiver `x` is low or not.

We first cover the simple case where `x` is low. Similarly to the field read case, we create new variables for all modifiable fields. According to properties (L4.unary) and (L4.rel) and since `x` is low, we may inhale the standard class

$[\![x.fn = y]\!]_C$
where fn is a private field and $fn \in C.mod$
:= `assert perm(leakable(`x`)) > 0 || perm(hidden(`x`)) > 0`
`if(perm(leakable(`x`)) > 0) {`
`assert low(`$x \neq$ `null)`
`assert !fromUC` $\implies x \neq$ `null`
`if(low(`x`)){` (PO11)
$\forall f \in C.mod$, `f.class` f_temp;
`inhale` $C.inv[\forall f' \in C.mod,\ this.f'/f'\_temp][this/x]$
$fn\_temp = y$
`assert` $C.inv[\forall f' \in C.mod,\ this.f'/f'\_temp][this/x]$
`} else {`
$\forall f \in C.mod$, `f.class` f_new, f_old;
`inhale` $\langle\langle C.inv \rangle\rangle^x_{\{(f', f'\_new, f'\_old)\ |\ f' \in C.mod\}}$
$fn\_new = y$
`assert` $\langle\langle C.inv \rangle\rangle^x_{\{(f', f'\_new, f'\_old)\ |\ f' \in C.mod\}}$
`}`
`} else {`
$x.fn = y$
`}`

Figure 3.22: Encoding for a modifiable field write.

invariant (both unary and relational part) with modifiable fields replaced by the corresponding variables. After that, we assign the variable that represents `x.fn`, the value `y` and check, if the invariant still holds after that assignment.

In the case where `x` is not low, according to (L4.rel) we may not inhale the relational part of the class invariant. Therefore, we have introduced the split-invariant, which after (SI1) always holds for a leakable reference-typed expression. In this case, we declare two new variables per modifiable field. The split-invariant transformation uses both variables to replace the corresponding modifiable field. We now inhale the split-invariant assertion, assign the first variable `fn_new` the value `y`. After (PO11) we now assert the split-invariant again, with the updated value of `fn_new`.

Split-Invariant

In the following, we show the encoding of the split-invariant assertion and give an intuition for why the formulation is what we want. That is, with this formulation, the previously shown modifiable field write encoding ensures that the write is invariant-preserving. In other words, that after the field assignment, the class invariant still holds for all actual objects.

The split-invariant assertion is a relational assertion that is an extension of the product program transformation of Eilers et al. [6]. Therefore, it will be encoded by this transformation.

We start by giving the encoding for the split-invariant without field replacements $\langle\langle inv \rangle\rangle^x$. In addition to the class invariant inv, the split-invariant assertion $\langle\langle inv \rangle\rangle^x$ takes as input a receiver variable x. We will define recursively how the split-invariant will be resolved by the product program transformation. For a ***non-atomic*** assertion, that is, one that contains logical connectives or quantifiers, the result of the transformation is the combined result of the underlying subformulas.

$$
\begin{aligned}
&\langle\langle A \odot B \rangle\rangle^x := \langle\langle A \rangle\rangle^x \odot \langle\langle B \rangle\rangle^x && \text{where } \odot \in \{\wedge, \vee, \implies, \iff\} \\
&\langle\langle \neg A \rangle\rangle^x := \neg \langle\langle A \rangle\rangle^x \\
&\langle\langle \mathcal{Q}\ x.\ A(x) \rangle\rangle := \mathcal{Q}\ x.\ \langle\langle A(x) \rangle\rangle && \text{where } \mathcal{Q} \in \{\forall, \exists\}
\end{aligned}
$$

Recall that from (CI1) we know that a class invariant can only contain expressions built from constants and private fields of the receiver `this`. For an ***atomic relational assertion***, the split-invariant will be resolved by splitting the underlying assertion a into the conjunction of two separate assertions for the two different executions and the corresponding receivers x_1, x_2. Recall from (CI4) that the low assertion is the only relational assertion a class invariant can contain. The split-invariant of a low expression is encoded as follows.

$$
\begin{aligned}
\langle\langle \text{low}(e) \rangle\rangle^x := (p_1 &\implies e_1[this_1/x_1] = e_2[this_2/x_1]) \\
\wedge\ (p_2 &\implies e_1[this_1/x_2] = e_2[this_2/x_2])
\end{aligned}
$$

So for the example invariant $\text{low}(this.f)$ the split-invariant would resolve as follows.

$$
\langle\langle \text{low}(this.f) \rangle\rangle^x = (p_1 \implies x_1.f_1 = x_1.f_2) \wedge (p_2 \implies x_2.f_1 = x_2.f_2)
$$

Note that we now read fields that we did not have with the standard encoding: $x_1.f_2$ and $x_2.f_1$. In order to keep the self-framing property of the invariant

(CI3), we need to also change the encoding of the non-relational parts of the invariant, such as permissions or other assertions. For a unary non-relational atomic assertion a we define

$$\begin{aligned}\langle\langle a \rangle\rangle^x := (p_1 \implies a_1[this_1/x_1] \land a_2[this_2/x_1]) \\ \land (p_2 \implies a_2[this_2/x_2] \land a_1[this_1/x_2])\end{aligned}$$

compared to the standard product program encoding which would be

$$a[this/x] := (p_1 \implies a_1[this_1/x_1]) \land (p_2 \implies a_2[this_2/x_2])$$

As an example why this is needed, consider the invariant $inv = Perm(this.f, \text{read}) \land \text{low}(this.f)$. The split-invariant version would be resolved as follows.

$$\begin{aligned}&\langle\langle Perm(this.f, \text{read}) \land \text{low}(this.f) \rangle\rangle^x \\ =\ &\langle\langle Perm(this.f, \text{read}) \rangle\rangle^x \land \langle\langle \text{low}(this.f) \rangle\rangle^x \\ =\ &(p_1 \implies Perm(x_1.f_1, \text{read}) \land Perm(x_1.f_2, \text{read})) \\ &\land (p_2 \implies Perm(x_2.f_2, \text{read}) \land Perm(x_2.f_1, \text{read})) \land \langle\langle \text{low}(this.f) \rangle\rangle^x \\ =\ &(p_1 \implies Perm(x_1.f_1, \text{read}) \land Perm(x_1.f_2, \text{read})) \\ &\land (p_2 \implies Perm(x_2.f_2, \text{read}) \land Perm(x_2.f_1, \text{read})) \\ &\land (p_1 \implies x_1.f_1 = x_1.f_2) \land (p_2 \implies x_2.f_1 = x_2.f_2)\end{aligned}$$

Without our adapted encoding for the unary assertions, we would be missing the permissions for $x_1.f_2$ and $x_2.f_1$. For other unary assertions, we can run into the same issue. For example, if the invariant is $this.g \neq 0 \land \text{low}(this.f/this.g)$ where f and g are modifiable fields. If we do not know whether $x_1.g_2 \neq 0$, the value of $x_1.f_2/x_1.g_2$ is undefined.

Split-Invariant with Field Replacements As we have seen, in the encoding we use a variant of the split-invariant where a modifiable field is substituted with two new variables. For this, we define a version of the split-invariant that handles those substitutions. This version additionally takes as parameter a set *repl* of 3-tuples. An element $(f, f\_new, f\_old) \in repl$ consists of the name of the modifiable field to be replaced f and two variables f_new, f_old. The idea is that for an assignment with a high receiver x, both executions have a ***different actual object*** as receiver. This means that for each actual object, the assignment will only take place in one of the executions. The variable f_new corresponds to the fields in the execution that get updated, whereas f_old corresponds to the fields that remain the same. We want to

check for both objects that the assignment does not break the relational part of the invariant. This is the case, iff for both receivers, for all low assertions, the expression inside the low assertion from before the assignment is equal to the one after the assignment. We define

$$\langle\langle a \rangle\rangle^x_{repl} := \langle\langle a \rangle\rangle^x [\forall (f, f\_new, f\_old) \in repl,\ x_1.f_1 / f\_new_1,\ x_1.f_2 / f\_old_1,\ x_2.f_1 / f\_old_2,\ x_2.f_2 / f\_new_2]$$

Consider an example modifiable field assignment `x.f = y` on a high receiver and a part of the class invariant `low(e(this.f))`, where `e(this.f)` is some expression containing `this.f`. Following the encoding from Figure 3.22, new variables `f_new` and `f_old` are introduced. Subsequently, we may assume the split-invariant with `this.f` replaced by `f_new` and `f_old`. This means that for the considered part `low(e(this.f))` we may assume,

$$\begin{aligned}\langle\langle low(e(this.f)) \rangle\rangle^x_{\{(f, f\_new, f\_old)\}} &= ((p_1 \implies e(x_1.f_1) = e(x_1.f_2)) \\ &\quad \wedge (p_2 \implies e(x_2.f_1) = e(x_2.f_2))) \\ &\quad [x_1.f_1 / f\_new_1,\ x_1.f_2 / f\_old_1, \\ &\quad\ x_2.f_1 / f\_old_2,\ x_2.f_2 / f\_new_2] \\ &= (p_1 \implies e(f\_new_1) = e(f\_old_1)) \\ &\quad \wedge (p_2 \implies e(f\_old_2) = e(f\_new_2))\end{aligned}$$

Then we assign `f_new = y`, that is, $\texttt{f\_new}_1$ `= y`$_1$ and $\texttt{f\_new}_2$ `= y`$_2$. After that, our considered part of the split-invariant is equivalent to

$$(p_1 \implies e(y_1) = e(f\_old_1)) \wedge (p_2 \implies e(f\_old_2) = e(y_2))$$

This can only hold if from the rest of the class invariant it can be proven that for each execution $i \in \{1, 2\}$ the new value y_i is the same as the value of $\texttt{f\_old}_i$, that is the value of $\texttt{x}_i\texttt{.f}_i$ before the assignment. This is exactly what we want. Since from that it follows that for both actual objects $\texttt{x}_1$ and $\texttt{x}_2$, the class invariant still holds. Thus, the assignment is invariant-preserving.

3.6.8 Encoding for the Remaining Cases

What follows are the remaining cases of our encoding.

$[\![$`private` $C\ (C_1\ arg_1, \ldots, C_n\ arg_n)\ spec\ \{s_1, \ldots s_k\}]\!]_C$
$:=$ `private` $C(C_1\ arg_1, \ldots, C_n\ arg_n,$ `boolean` $fromUC)\ spec\ \{$
`inhale Perm(this.`f`, write)` $\forall f \in C.fields,$
`inhale Perm(hidden(this), write)`
`assume low(this)` (G3)
$[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C$
$\}$

$[\![am\ C\ (C_1\ arg_1, \ldots, C_n\ arg_n)\ spec\ \{s_1, \ldots s_k\}]\!]_C$
where $am \in \{$`public`, `protected`, $\epsilon\}$
$:=$ $am\ C(C_1\ arg_1, \ldots, C_n\ arg_n,$ `boolean` $fromUC)\ spec\ \{$
`inhale Perm(this.`f`, write)` $\forall f \in C.fields,$
`inhale Perm(hidden(this), write)`
`assume low(this)` (G2.1)
$[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C$
$\}$

$am\ C(C_1\ arg_1, \ldots, C_n\ arg_n,$ `boolean` $fromUC)$
`ensures Perm(leakable(\result), write)`
$\{$
$fromUC$ `= true` (E1)
`inhale Perm(this.`f`, write)` $\forall f \in C.fields,$
`inhale Perm(hidden(this), write)`
`assume low(this)` (G2.1)
$\forall i \in \{1, \ldots, n\}$
`inhale` arg_i`.class` $\notin$ `primitive_types`
$\implies$ `Perm(leakable(`arg_i`), write)`
`assume low(`arg_i`)` (G2.2)
$[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C,$
$[\![$`leak this`$]\!]_C$
$\}$

$[\![$`private` C_{ret} $mn(C_1\ arg_1, \ldots, C_n\ arg_n)$ *spec* $\{s_1, \ldots s_k\}]\!]_C$
:= `private` C_{ret} $mn\_h(C_1\ arg_1, \ldots, C_n\ arg_n,$ `boolean` *fromUC*) *spec*
$\{[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C\}$

$[\![x =$ `new` $C_{obj}(arg_1, \ldots, arg_n)]\!]_C$
where C_{obj} is unverified
:= $\forall i \in \{1, \ldots, n\}$, `assert low(`$arg_i$`)` (PO6)
$\forall i \in \{1, \ldots, n\} \wedge arg_i$`.class` $\notin$ `primitive_types,`
`assert perm(leakable(`arg_i`)) > 0`
$x =$ `new` $C_{obj}(arg_1, \ldots, arg_n,$ *fromUC*) (E2)
`inhale Perm(leakable(`x`, write)`
`assume low(`x`)` (G5)

$[\![x =$ `new` $C_{obj}(arg_1, \ldots, arg_n)]\!]_C$
where C_{obj} is verified
:= $x =$ `new` $C_{obj}(arg_1, \ldots, arg_n,$ *fromUC*) (E2)

$[\![y = x.mn(arg_1, \ldots, arg_n)]\!]_C$
where the ***runtime class*** of x is unverified or unknown
:= `assert low(`x`)` (PO4), (PO18)
$\forall i \in \{1, \ldots, n\}$, `assert low(`$arg_i$`)` (PO2)
$\forall i \in \{1, \ldots, n\} \wedge arg_i$`.class` $\notin$ `primitive_types,`
`assert perm(leakable(`arg_i`)) > 0`
$y = x.mn(arg_1, \ldots, arg_n,$ *fromUC*) (E2)
`inhale` y`.class` $\notin$ `primitive_types` $\implies$ `Perm(leakable(`y`, write)`
`assume low(`y`)` (G4)

$[\![y = x.mn(arg_1, \ldots, arg_n)]\!]_C$
where the ***runtime class*** of x is verified
:= `assert low(`x`.getClass())` (PO18)
$y = x.mn\_h(arg_1, \ldots, arg_n,$ *fromUC*) (E2)

$[\![y = x.fn]\!]_C$
where fn is a public, protected or package-private field

```
:= assert perm(leakable(x)) > 0 || perm(hidden(x)) > 0
   if(perm(leakable(x)) > 0) {
       assert low(x ≠ null)
       assert !fromUC ⟹ x ≠ null
       f.class temp;
       inhale y.class ∉ primitive_types ⟹
                  Perm(leakable(temp), wildcard)
       assume low(x) ⟹ low(temp) (G6)
       y = temp
   } else {
       y = x.fn
   }
```

$[\![y = x.fn]\!]_C$
where fn is a private field and $fn \notin C.mod$

```
:= assert perm(leakable(x)) > 0 || perm(hidden(x)) > 0
   if(perm(leakable(x)) > 0) {
       ∀f ∈ C.mod, f.class f_temp;
       inhale unary(C.inv)[∀f' ∈ C.mod, this.f'/f'_temp][this/x] (G8.1)
       assume low(x) ⟹
                rel(C.inv)[∀f' ∈ C.mod, this.f'/f'_temp][this/x] (G8.2)
   }
   y = x.fn
```

$[\![x.fn = y]\!]_C$
where fn is a public, protected or package-private field

```
:= assert perm(leakable(x)) > 0 || perm(hidden(x)) > 0
   if(perm(leakable(x)) > 0) {
       assert low(x ≠ null)
       assert !fromUC ⟹ x ≠ null
       assert y.class ∉ primitive_types ⟹
                   Perm(leakable(y), wildcard)
       assert low(x) (PO10)
       assert low(y) (PO8)
   } else {
       x.fn = y
   }
```

$[\![x.fn = y]\!]_C$
where fn is a private field and $fn \notin C.mod$

```
:= assert perm(leakable(x)) > 0 || perm(hidden(x)) > 0
   if(perm(leakable(x)) > 0) {
       if(low(x)){(PO12)
           ∀f ∈ C.mod, f.class f_temp;
           inhale C.inv[∀f' ∈ C.mod, this.f'/f'_temp][this/x]
           x.fn = y
           assert C.inv[∀f' ∈ C.mod, this.f'/f'_temp][this/x]
       } else {
           ∀f ∈ C.mod, f.class f_new, f_old;
           inhale ⟨⟨C.inv⟩⟩^x_{(f',f'_new,f'_old) | f'∈C.mod}
           x.fn = y
           assert ⟨⟨C.inv⟩⟩^x_{(f',f'_new,f'_old) | f'∈C.mod}
       }
   } else {
       x.fn = y
   }
```

$[\![$`if(`*cond*`) {`$s_1, \ldots s_k$`} else {`$s_{k+1}, \ldots s_n$`}`$]\!]_C$
:= `assert low(`*cond*`)` (PO17)
`if(`*cond*`) {`$[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C$`} else {`$[\![s_{k+1}]\!]_C, \ldots, [\![s_n]\!]_C$`}`

$[\![$`while(`*cond*`) {`$s_1, \ldots s_k$`}`$]\!]_C$
:= `assert low(`*cond*`)` (PO17)
`while(`*cond*`) {`
$[\![s_1]\!]_C, \ldots, [\![s_k]\!]_C$
`assert low(`*cond*`)` (PO17)
`}`

$[\![$*type*`[] a = new` *type*`[i]`$]\!]_C$
:= *type*`[] a = new` *type*`[i]`
`assume low(`*a*`)` (G9.1)
`inhale Perm(hidden(`*a*`), write)` (G9.2)

$[\![y = a[i]]\!]_C$
:= `assert perm(leakable(`*a*`)) > 0 || perm(hidden(`*a*`)) > 0`
`if(perm(leakable(`*a*`)) > 0) {`
`assert` $(0 \leq i \wedge i \leq$ `a.length`$)$ $\vee$ $($`low(a.length)` $\wedge$ `low(`*i*`)`$)$
`assert !fromUC` $\implies 0 \leq i \wedge i \leq$ `a.length`
a`.elem.class` *temp*`;`
`inhale` *a*`.elem.class` $\notin$ `primitive_types` $\implies$
`Perm(leakable(`*temp*`), wildcard)` (G10.2)
`assume low(`*a*`)` $\wedge$ `low(`*i*`)` $\implies$ `low(`*temp*`)` (G10.1)
$y = temp$
`} else {`
$y = a[i]$
`}`

$[\![a[i] = y]\!]_C$
:= assert perm(leakable(a)) > 0 || perm(hidden(a)) > 0
    if(perm(leakable(a)) > 0) {
        assert $(0 \leq i \wedge i \leq a.\texttt{length}) \vee (\texttt{low}(a.\texttt{length}) \wedge \texttt{low}(i))$
        assert !fromUC $\implies 0 \leq i \wedge i \leq a.\texttt{length}$
        assert $y.\texttt{class} \notin$ primitive_types $\implies$
                Perm(leakable(y), wildcard) (PO20.2)
        assert low(a) (PO22)
        assert low(i) (PO19)
        assert low(y) (PO20.1)
    } else {
        $a[i] = y$
    }

$[\![\texttt{leak } a]\!]_C$
  where a is of array type
:= assert perm(leakable(a)) > 0 || perm(hidden(a)) > 0
    if(perm(leakable(a)) == 0) {
        exhale Perm(hidden(a), write)
        inhale Perm(leakable(a), write)
        assert low(a) (PO24)
        assert low(a.length) (PO23)
        $\forall i \in \{0,\ldots,a.\texttt{length}\}$,
            assert low($a[i]$) (PO25.1)
            assert $a.\texttt{elem.class} \notin$ primitive_types $\implies$
                    Perm(leakable($a[i]$), wildcard) (PO25.2)
            exhale Perm($a[i]$), write)
    }

otherwise
$[\![x]\!]_C := x$

Chapter 4

Implementation

In this chapter, we discuss the implementation of the design in Chapter 3. We first give an overview of the existing infrastructure and then present our extensions. Finally, we compare our implementation with the design by discussing its differences and limitations.

4.1 Existing Infrastructure

As mentioned in Chapter 1, we implemented our design in the Java frontend of the VerCors verification tool [1]. Internally, VerCors then uses the Viper backend for verification. Both are implemented in Scala.

VerCors takes as input a Java program with specification annotations. These are given in Java comments. The input is parsed into an abstract syntax tree (AST) which is subsequently simplified step by step by going through different transformations. One transformation typically removes one or more high-level language constructs by encoding them with more basic low-level constructs. In this way, the program is rewritten into a simpler language which is then converted to the Viper intermediate language. The Viper program can then be verified using one of Vipers backends (Silicon or Carbon). Both of them finally use the Z3 [5] SMT solver for their proofs.

The Viper tool supports running plugins on top of it. Eilers et al. [6] have

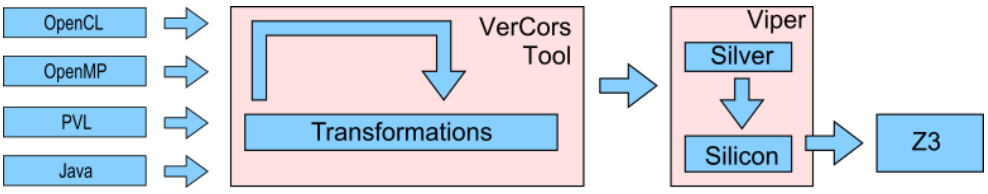

Figure 4.1: Overview of the VerCors tool (taken from [22]).

implemented their product program transformation for information flow security (see Section 2.1) as a Viper plugin. The plugin allows the user to specify information flow security constructs and encodes them into the Viper intermediate language, such that the standard Viper backends can be used for verification.

4.2 Extensions to VerCors

In this section, we discuss our extensions to VerCors in two parts. We first discuss our additions for supporting verification of information flow security and then for verification in the presence of unverified code.

4.2.1 Information Flow Security Extensions

We extended VerCors to be able to verify information flow security properties. For that, we added new specifications to VerCors' specification language. These include the `low` and `low event` assertion that we used in our encoding (see Section 3.6). We also added support for the `declassify` statement, which we have mentioned in Section 2.1. In order to resolve those constructs, we reuse the Viper plugin of Eilers et al. [6]. The new VerCors AST nodes are passed through all transformations and then converted into their Viper plugin analogue, before the MPP transformation encodes them into the standard Viper intermediate language.

In addition, we changed VerCors to support secure information flow verification for more complex language features. In VerCors `return`, `break`, `throw`, `try-catch` statements were resolved with `goto` statements, which are not fully supported by the MPP transformation. With our extension, VerCors converts them directly to the corresponding Viper AST nodes provided by the secure information flow Viper plugin.

With those additions, we created a VerCors version that supports standard verification of information flow security outside of the unverified code setting. This version can be used independently of the work in the remainder of the thesis.

4.2.2 Unverified Code Extensions

For supporting unverified code, we add ***hidden*** and ***leakable*** specifications to VerCors. To resolve them, we take the same approach as Necker [15]: we model them as abstract predicates. For modeling unverified code, we add the `unverified_class` annotation that can be used to mark a class as unverified code. For verified classes, we add the possibility of adding a ***class invariant*** and a ***modifiable*** annotation for private fields. Furthermore, we extend the

VerCors AST to support the ***leak operation***, as well as a ***split-invariant*** assertion, as discussed in Section 3.6.7.

With these extensions, we can implement our encoding as described in Section 3.6. We implement it as an additional transformation step in the VerCors transformation pipeline. Our encoding takes place approximately in the middle of the transformations. This way, Java type information (which is later removed but necessary for our design) is still present, while some simplifications (for example, the extraction of side effects) have already taken place.

After our transformation has finished, we have resolved all special nodes in the AST into standard VerCors AST nodes or into specific secure information flow nodes, described in Section 4.2.1. VerCors then uses the Viper plugin of Eilers et al. [6] to transform the secure information flow nodes of the program into the standard Viper nodes. Subsequently, the program can be verified by calling a standard Viper backend.

4.3 Extensions to Viper

Recall that with the split-invariant, we introduced a relational assertion that is transformed differently from other relational assertions. For that reason, we have extended the AST of the Viper plugin of Eilers et al. [6] with a split-invariant node. Furthermore, we extended the product program transformation of the plugin, to encode the split-invariant as described in Section 3.6.7.

4.4 Limitations and Extensions

4.4.1 Limitations

When comparing the implementation with our design in Chapter 3, there are a few limitations that are not supported by our implementation.

Runtime Class Tracking

In order to distinguish an unverified object from a verified object, we need to track the runtime class of an object. Our verification technique needs to treat every object for which the runtime class is not known as an unverified object. Furthermore, the runtime class is needed to ensure that no dynamically-bound calls depending on a secret occur (see (PO18)). However, VerCors currently does not enforce behavioral subtyping [12] and thus does not support reasoning about calls to overridden methods. Moreover, it does not implement the tracking of runtime classes. As a consequence, our implementation cannot reliably detect insecure information flows caused by

subclassing. Our implementation is sound, if the user only declares final classes, to completely disable subclassing. This prevents the adversary from creating unverified subclasses and providing them as arguments with verified static type. Furthermore, using only final classes ensures that there are no dynamically-bound calls that depend on high data, preventing the associated leakage via covert channels (see Section 3.4.8).

Weak Specifications for Primitive Operations

Our implementation does not include weak specifications for some primitive operations: method invocations and array accesses. In both cases, the strong method specifications are used, even when called from unverified code. Specifically, VerCors always checks that the receiver of a method invocation is non-null and that the index of an array access is in the correct range. Compared to our design in Section 3.5.3, our implementation is therefore less complete, as in those cases we do not allow our code to crash if called from unverified code. However, the soundness is not affected.

Primitive Type Arrays

We decided to not implement support for arrays with primitive type elements, since that would have complicated the implementation disproportionally compared to its limited benefit. Instead, the user can define a wrapper class and use arrays with elements of this class.

4.4.2 Extensions

Our implementation supports some language constructs that have not been explicitly covered in Chapter 3. However, our design as an idea covers them in principle. These include `for` loops, as well as the constructs mentioned in Section 4.2.1: `throw`, `try-catch`, simple `break` statements and `return` statements inside constructors. These are directly converted into the corresponding Viper node, which is then encoded by the modular product program transformation of Eilers et al. [6]. In the standard information flow setting, their correctness follows directly from the design of Eilers et al. [6]. However, we have not added explicit proof obligations for verification in the presence of unverified code. For the throw statement, it must be ensured that the thrown exception does not depend on high data. Except for that, the constructs do not need further proof obligations. (PO17) and (PO18) are enough to guarantee correctness in the scenario with unverified code.

Chapter 5

Evaluation

In this chapter, we evaluate our implementation and discuss the results.

We tested our implementation against all the examples presented in Chapter 3 as well as several adapted variants. In every case, our implementation successfully produced the expected verification result.

The examples were run on an HP ProBook 445 G8 Notebook with an AMD Ryzen 5800U processor and 16GB RAM running Ubuntu 22.04 on a warmed up JVM. As can be seen, the examples verify reasonably fast. For most examples, the verification time does not exceed three seconds. However, some array examples require more time, especially on the Carbon backend, based on verification condition generation.

Example	Silicon [s]	Carbon [s]	Verdict
Figure3_01_secure_array.java	3.08	7.19	pass
Figure3_02.java	1.47	1.83	fail
Figure3_02_low_inv.java	1.64	1.87	fail
Figure3_02_low_inv_declassified.java	1.45	1.89	pass
Figure3_02_no_getter.java	1.32	1.67	pass
Figure3_03.java	1.05	1.39	fail
Figure3_03_declassified.java	1.07	1.39	pass
Figure3_04.java	1.09	1.37	fail
Figure3_04_declassified.java	1.07	1.38	pass
Figure3_05.java	1.10	1.40	fail
Figure3_05_array.java	2.50	2.32	fail
Figure3_05_array_declassified.java	1.86	2.29	pass
Figure3_05_declassified.java	1.09	1.38	pass
Figure3_06.java	1.21	1.54	fail
Figure3_06_declassified.java	1.21	1.54	pass

Table 5.1: Examples from Chapter 3 with their running time for both Viper backends and the verification result.

Example	Silicon [s]	Carbon [s]	Verdict
Figure3_07.java	1.65	1.79	fail
Figure3_07_array.java	2.59	2.54	fail
Figure3_07_array_declassified.java	2.00	2.45	pass
Figure3_07_declassified.java	1.41	1.73	pass
Figure3_08.java	1.64	1.79	fail
Figure3_08_array.java	2.96	2.71	fail
Figure3_08_array_declassified.java	2.08	2.63	pass
Figure3_08_declassified.java	1.45	1.77	pass
Figure3_08_nonmod.java	1.67	1.91	fail
Figure3_08_nonmod_array.java	3.15	2.96	fail
Figure3_08_nonmod_array_declassified.java	2.10	2.72	pass
Figure3_08_nonmod_declassified.java	1.46	1.86	pass
Figure3_09.java	1.21	1.62	fail
Figure3_09_declassified.java	1.21	1.61	pass
Figure3_10.java	1.26	1.61	fail
Figure3_10_declassified.java	1.26	1.65	pass
Figure3_11.java	1.53	2.15	fail
Figure3_11_array.java	3.77	18.45	fail
Figure3_11_array_declassified.java	2.23	8.10	pass
Figure3_11_declassified.java	1.53	2.18	pass
Figure3_12.java	1.45	1.82	fail
Figure3_12_declassified.java	1.41	1.83	pass
Figure3_13.java	1.77	2.38	fail
Figure3_13_array.java	9.56	28.52	fail
Figure3_13_array_declassified.java	2.69	9.62	pass
Figure3_13_declassified.java	1.83	2.27	pass
Figure3_14_secure_array.java	3.98	15.52	pass
Figure3_15.java	1.19	1.60	fail
Figure3_15_declassified.java	1.21	1.60	pass
Figure3_15_mod.java	1.36	1.80	fail
Figure3_15_mod_declassified.java	1.40	1.81	pass
Figure3_15_nonmod.java	1.42	1.99	fail
Figure3_15_nonmod_declassified.java	1.41	1.90	pass
Figure3_16_run2.java	1.35	1.74	fail

Table 5.1: Examples from Chapter 3 with their running time for both Viper backends and the verification result.

The examples are labeled with their corresponding figure from Chapter 3. Some examples have slight adaptations. These include converting ternary statements to if statements and adding permissions, or other verification

```
class A {

  void foo(int secret){
    A obj1 = new A();
    A obj2 = new A();
    A recvr = secret > 0
        ? obj1
        : obj2;
  }
}
```

```
class A {

    //requires 0 <= secret
    //   && secret < 2;
   void foo(int secret){
     A obj1 = new A();
     A obj2 = new A();
     A[] objs = new A[2];
     objs[0] = obj1;
     objs[1] = obj2;
     A recvr = obj[secret];
  }
}
```

Figure 5.1: A simple example for the array transformation on the examples.

constructs that have been omitted in Chapter 3 for the sake of readability. The examples from Figures 3.1 and 3.14 are the only secure examples. They are marked *_secure* in the table and correctly pass verification. All other examples include insecure information flows and correctly fail verification. For every failing example, we tested our implementation on the same program with an appropriate `declassify` statement to create a passing example. Moreover, we tested it on some other adapted versions of the examples. The adaptations can be inferred mostly from the name.

We give a quick overview of the array adaptation which we have applied to suitable examples (examples are marked *_array*). Recall the proof obligation (PO17) that disallows high branching. Therefore, examples that set a high receiver through a branch are disallowed. In order to still evaluate our implementation on proof rules that require a low receiver, such as (PO10), we used an array access instead. The transformation is depicted in Figure 5.1. Note that this transformation was also used for the passing examples from Figures 3.1 and 3.14. Both of them include a high branch in the original version and would fail if the array transformation was not applied.

Due to the limitation discussed in Section 4.4.1, our implementation is not able to detect insecure information flows originating from dynamically-bound calls, or from an adversary subclass of our verified classes. Therefore, we could not evaluate our implementation against such examples.

Overall, despite this limitation, the evaluation demonstrates that our implementation reliably detects insecure information flow in representative examples. The results consistently matched our expectations. Moreover, the reasonably fast verification times suggest that our verification technique is sufficiently efficient.

Chapter 6

Conclusion

In this thesis, we presented a novel technique for verifying information flow security for a subset of Java. Our technique uses permission-based reasoning and builds on previous work for verification in the presence of unverified code and verification of information flow security. Our adversary controls the unverified parts of our code base and has access to a dynamically changing part of our heap memory. With our design, we ensure that the adversary does not gain any new information about the high inputs. This includes explicit and implicit information flow, as well as covert side channels such as timing and termination.

We implemented our technique on top of the VerCors verification tool and its underlying infrastructure Viper. Furthermore, we evaluated our implementation against various examples. With that, we have shown that our technique achieves the desired security guarantees and is efficient.

Due to practical limitations concerning subclassing and runtime class tracking, the implementation could not cover the complete design. However, our implementation is sound in a setting where all classes are declared to be final. Our technique successfully provides precise information flow security guarantees in the presence of unverified code.

Future Work For future work, the implementation could be extended to support subclassing and runtime tracking, to encompass the complete design. Moreover, to allow for a more precise runtime class tracking, the design could be extended to support the `final` keyword. Furthermore, it would be interesting to consider how static class members differ from the instance members that our design covers. Finally, it would be interesting to test both the design and implementation on more complex real-world examples.

Bibliography

[1] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.

[2] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery.

[3] John Boyland. Checking Interference with Fractional Permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[4] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[5] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[6] Marco Eilers, Severin Meier, and Peter Müller. Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 718–741, Cham, 2021. Springer International Publishing.

[7] Marco Eilers and Peter Müller. Nagini: A Static Verifier for Python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[8] Marco Eilers, Peter Müller, and Samuel Hitz. Modular Product Programs. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 502–529, Cham, 2018. Springer International Publishing.

[9] Gerwin Ernst and Toby Murray. SecC Tool Description and Isabelle Theories for SecCSL. https://covern.org/secc, 2019.

[10] Gidon Ernst and Toby Murray. SecCSL: Security Concurrent Separation Logic. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 208–230. Springer, 2019.

[11] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods Syst. Des.*, 48(3):152–174, 2016.

[12] Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[13] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[14] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release, 2006. https://www.cs.cornell.edu/jif.

[15] Frederic Necker. Verification of object invariants in the presence of unverified code. Master's thesis, ETH Zurich, 2023.

[16] Oracle. Object (Java Platform, Standard Edition 8 API Documentation). https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--, 2014. Accessed: 2025-08-29.

[17] Oracle. SecurityManager (Java Platform, Standard Edition 8 API Documentation). https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html, 2014. Accessed: 2025-08-25.

[18] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL '05, page 247–258, New York, NY, USA, 2005. Association for Computing Machinery.

[19] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[20] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[21] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), May 2012.

[22] University of Twente, FMT Group. VerCors Tutorial. https://vercors.ewi.utwente.nl/wiki/. Accessed: 2025-09-06.

[23] Dennis M. Volpano and Geoffrey Smith. Probabilistic Noninterference in a Concurrent Language. *J. Comput. Secur.*, 7(1), 1999.

ETH

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 Information Flow Security 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):**
Kaletsch	Nicolas Gabriel

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)**
Bern, 14.09.2025	

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