Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verification of Information Flow Security for Python Programs

Master Thesis

Severin Meier

September 14, 2018

Advisors: Prof. Dr. Peter Müller, Marco Eilers

Department of Computer Science, ETH Zürich

Abstract

Ensuring that no confidential data is leaked is a major concern in the software industry. One way to guarantee this is to formally prove that a program has secure information flow. This property can be expressed as noninterference, which requires relating two executions of the program in order to prove it, making it a hyperproperty. Modular product programs provide a method to prove hyperproperties using off-the-shelf verifiers, however, their original definition is only for a small language. In this thesis, we extend the definition of modular product programs to make it suitable for encoding a wide range of object-oriented languages, as well as verification constructs such as predicates and pure functions. We present encodings to prove the absence of termination channels and a way to ensure that concurrent programs cannot leak any secrets. In order to prove information flow security of Python programs, we extend the specification language of Nagini, a Python verifier, and provide an implementation of the encoding based on Nagini and the Viper verification infrastructure. The resulting implementation can encode the same subset of Python programs that Nagini supports as modular product programs, and the transformation does not impact completeness for almost all examples. Our evaluation shows that it allows us to verify secure information flow properties with a reasonably small overhead compared to standard Nagini verification.

Acknowledgements

I would like to thank my supervisor Marco Eilers for his invaluable inputs and fruitful discussions throughout the project, as well as his useful feedback. I would also like to thank Prof. Dr. Peter Müller and his group for giving me the opportunity to work on this fascinating topic. Last but not least, I want to thank my family for their endless support.

Contents

Chapter 1

Introduction

Writing correct software is challenging, especially with the high degree of complexity that modern programs typically display. With software governing more and more parts of our lives, it is a serious concern that confidential data might be leaked. Unfortunately, it happens fairly frequently that software contains weaknesses, which can be exploited to gain access to this secret data. It is therefore desirable to have mathematically rigorous proofs that a program does not inadvertently reveal any of its secrets.

Information flow security, the program property expressing that no information about secret data is revealed, addresses this issue. We can express it as a noninterference property: If the program is executed multiple times with the same public inputs, but potentially different secret values, the public output should always be the same. If this is the case we know that the secret values did not affect the public outputs and an observer cannot draw any conclusions about the secrets. Since one has to relate multiple executions of the program to show this, noninterference is called a hyperproperty.

Figure 1.1 shows a motivating example of a program that does not exhibit secure information flow, as it can leak secret information. The method `sum_foo` takes a list of integers as an input and sums up the values obtained by calling a method `foo` on each element in the list. If that method raises an exception, a message is printed to notify the user. This is a problem in a situation where the values in the list are secret, and whether or not `foo` raises an exception depends on its input. In that case, an observer can learn information about the values from whether or not the message was printed.

Deductive program verification is the discipline of verifying programs with respect to a specification supplied by the user. This can be automated in verifiers that try to construct a formal proof that the program fulfills the specification or raise an error if it does not. Such verifiers have developed into a state where one can prove a wide variety of program properties effi-

```python
def sum_foo(l: List[int]) -> int:
    f_sum = 0
    try:
        for element in l:
            f_sum += foo(element)
    except SomeException:
        print("Error: foo raised exception")
    finally:
        return f_sum
```

Figure 1.1: Example program which might leak secrets (Python).

ciently, but hyperproperties remain a challenge, as most tools only reason about single executions.

Barthe, D'Argenio and Rezk [5] proposed self-composition as a way to be able to use these tools for the verification of hyperproperties, such as secure information flow. The idea is to reduce the hyperproperty in the source program to a normal trace property of a newly generated program. A single execution of this generated program models multiple executions of the source program. The same principle is applied in modular product programs [12], which, unlike self-composition, also allow for modular verification of the generated program.

The goal of this thesis is modular verification of secure information flow for Python programs using Nagini [11], a Python verifier based on the Viper verification infrastructure [18]. We present a framework for providing Python programs with relational specifications to prove secure information flow, as well as the absence of termination channels and possibilistic noninterference of concurrent programs. Reusing as much of the existing infrastructure as possible, we provide an implementation to encode the Python programs and their specifications as modular product programs.

Modular product programs are only defined for a small language originally, which does not allow us to encode the range of programs we aim to verify. The example in Figure 1.1 uses some more sophisticated language constructs, such as try/catch/finally blocks, which are used frequently in real world programs and should therefore be supported. To that effect, we extend the definition of modular product programs by common language constructs, such that it is suitable for encoding a wide range of object-oriented programming languages. We also add support for predicates and pure functions, which are verification constructs used for expressing specifications, to the modular product program encoding.

The contributions of this thesis are the following:

- We extend the definition of modular product programs to allow for the verification of object-oriented programming languages.

- We define a modular product program encoding for predicates and pure functions.
- We implement the modular product program encoding in the Viper framework, in such a way that it remains independent of the source language.
- We complement the Nagini specification language with contracts expressing secure information flow properties, including a convenient way to specify defaults.
- We extend Nagini such that it can encode Python programs into modular product programs and verify secure information flow.
- We adapt the methodology for proving absence of termination channels presented in the MPP paper to work with our implementation, and we provide a way to prove possibilistic noninterference for concurrent Python programs.
- We evaluate our implementation with respect to expressiveness and completeness, as well as performance.

Chapter 2 provides the necessary background knowledge needed in this thesis. In Chapter 3 we describe the extensions we made to modular product programs. How we make use of them to prove secure information flow security, as well as absence of termination channels and possibilistic noninterference, is covered in Chapter 4. In Chapter 5 we document the implementation in Nagini and Viper, which we evaluate in Chapter 6. We conclude in Chapter 7.

Chapter 2

Background

This chapter covers the required background knowledge of topics needed in this thesis. We begin with some general notes about hyperproperties, including some related work. Next we discuss noninterference and specifically Information Flow Security, as the example of a hyperproperty we are going to focus on in this thesis. We then give a summary of modular product programs, the basis for our extensions covered in Chapter 3. We also provide a short overview of Viper and Nagini, the tools on which the implementation of this project is based.

2.1 Hyperproperties

A *hyperproperty* is a program property which relates multiple executions. An example of a 2-hyperproperty, a property relating two executions, would be determinism. A program is deterministic, if running it twice with the same inputs always gives the same output. A 3-hyperproperty is, for example, transitivity, because proving it requires three executions to relate. In general a k-hyperproperty is a program property relating k executions. Hyperproperties can be expressed in specifications which relate multiple executions, which is why they are called *relational specifications*, as opposed to *unary specifications*, which only concern a single execution (e.g., functional specifications).

Proving hyperproperties of a program is challenging, since usually program verifiers consider a single execution of a program, and aim to prove functional specifications. One way to reason about hyperproperties is using relational program logics [6, 21, 23], but they are hard to automate and require dedicated tools for verification.

Another way is self-composition [5], where one creates a new composite program, which simulates running the source program multiple times. Fig-

```
method m(x: Int) returns (res: Int)
   relational x1 == x2 ==> m(x1) == m(x2)
{
   res := x + 1
}

method wrapper(x1: Int, x2: Int)
/*self-composition of m, generated automatically*/
{
   var tmp1 := x1 + 1 // inlined m(x1)
   var tmp2 := x2 + 1 // inlined m(x2)
   assert x1 == x2 ==> tmp1 == tmp2
}
```

Figure 2.1: Example of relational specification and self-composition. Method `m` gets a relational specification, `wrapper` shows the self-composition of `m`.

ure 2.1 shows an example how to prove determinism of a method `m`. We relate two calls to `m` with the same input, i.e., we add the relational specification `x1 == x2` ⇒ `m(x1) == m(x2)` to the specifications of `m`. To verify this, we create a new `wrapper` method, where all the calls in the specification are inlined, so in our example we have the body of `m` twice, first with input `x1` substituted for the argument, then with input `x2`. We can then use the results of the inlined calls to assert the relational specification.

The advantage of this approach is that a standard program verifier can be used to run the verification. The downside on the other hand is that since all the code of the method is duplicated, verification becomes non-modular, as there is no one place where a specification of a call that `m` makes could be applied. If, for example, `m` calls another method `g`, which itself is deterministic, it is not clear how to convey this information to the inlined `m` in the wrapper, except to again inline the full body of `g`, because at the point in the inlining of `m` where we call `g`, there are no two executions the specification of `g` could relate. A way of making it modular would be to require exact functional specifications, but in many cases this is not desirable, as we want to allow for abstraction between code and specification. An example of a tool which implements self-composition for a real-world programming language is the Frama-C plugin RPP [7].

A third approach to verifying hyperproperties are product programs [3, 4]. They use a similar idea of modelling multiple executions of the source program in a new composite program, now called the *product*. In particular, we will focus on modular product programs (or MPPs) [12], which we will cover in more detail in Section 2.3. MPPs provide a modular way to verify programs with respect to hyperproperties, without requiring dedicated tools; instead, they allow for using off-the-shelf tools for functional verifica-

```
method main(val: Int, secret: Int) returns (res: Int)
   requires low(val)
   ensures low(res) // must fail: res depends on secret
{
   if (secret > 0) {
      res := val + 1
   } else {
      res := val
   }
}
```

Figure 2.2: Example program that leaks secret information. A secret input affects the result.

tion same as self-composition.

2.2 Information Flow Security

Information flow security (also secure information flow or SIF) is a program property expressing that a program does not leak any secret information, such as passwords or encryption keys. We can express this as a *noninterference* property as follows: If we execute the program multiple times using the same public inputs, but different secret values, the public outputs should be the same each time. If that is the case we know that no secret information can influence the public outputs, and therefore there is no leak. Since noninterference relates two executions, it is a 2-hyperproperty.

For verification of secure information flow, we require specifications which classify data into public and secret. We call public information *low*, as opposed to secret information being *high*. With this we can formulate SIF as follows: Let $\mathtt{x}^{(i)}$ be the value of variable `x` in execution i, Low_{in} the set of all low input variables and Low_{out} the set of low outputs. Now we have secure information flow if $(\forall \mathtt{x} \in Low_{in}. \mathtt{x}^{(1)} = \mathtt{x}^{(2)}) \Rightarrow (\forall \mathtt{y} \in Low_{out}. \mathtt{y}^{(1)} = \mathtt{y}^{(2)})$. In other words, this means that we have to prove that given the low inputs are equal in both executions, the low outputs are equal in both executions. Then we know that no high inputs had an effect on the low outputs, which means that no information about them could have leaked. We will assume that everything that is not explicitly specified to be low is high.

Figure 2.2 shows a small example program on which we try to verify secure information flow. The method `main` takes two arguments, a public `val` and a `secret`. We try to prove that the result `res` is low, which has to fail, since it will differ depending on the value of secret being positive or not. An observer, who would know the value `val` because it is public, could see from the result of the method (also public) whether or not `secret > 0`, which is a leak of secret information. We will show in the next section how to verify this code using modular product programs.

2.3 Modular Product Programs

In this section we provide a summary of modular product programs (or MPPs) as presented originally [12]. As mentioned before, the general idea of MPPs is to model multiple executions of the source program in a single composite program. MPPs do this in such a way that every loop and function call in the source program corresponds to one loop or call in the product. This is desirable, as loops and calls come with their own specifications, so when they are not duplicated there is exactly one point in the product at which their relational specifications can be applied, making verification modular and removing the limitations of self-composition mentioned in Section 2.1

MPPs are defined such that they can relate k executions of a program for arbitrary k, such that they enable us to prove k-safety hyperproperties. An MPP creates k copies of all the variables of the source program, and adds k *activation variables*. An activation variable is a Boolean variable representing whether or not a specific execution is active. For example, an assignment statement in the source program is transformed to k conditional assignments in the product, each one expressing that if execution i is active, the assignment on the i-th version of the variables is executed. A conditional statement is encoded as creating new versions of the activation variables, such that each branch gets its own activation variable to be used in its encoding. The value of the new variable is the value of the current activation variable conjoined with the condition of the corresponding branch.

Figure 2.3 shows the MPP for $k = 2$ of the example in Figure 2.2. All inputs and outputs are duplicated, to have one version per modelled execution, and we added two activation variables `p1` and `p2`. The low specifications are translated as mentioned above, the two versions of the low expression have to be equal. Note that the relational specifications (such as `low`) are conditional on both executions being active, as otherwise it would not make sense to compare the values. As mentioned above, the `if` conditional is encoded as new versions for the activation variables, for example, execution 1 is active inside the `then` block (`p1_t`), iff it was active before reaching the `if` statement (`p1`) and the condition is true (`secret1 > 0`).

We can now use a standard functional code verifier to try and verify this code, and, as expected, it will report an error because it cannot prove the postcondition. If we instead made the `secret` input public (by specifying it as low), we could verify the example, since we would know both executions

```
method main(p1: Bool, p2: Bool,
            val1: Int, val2: Int,
            secret1: Int, secret2: Int)
   returns (res1: Int, res2: Int)
   requires p1 && p1 ==> val1 == val2
   ensures p1 && p1 ==> res1 == res2 // fails
{
   p1_t := p1 && secret1 > 0
   p2_t := p2 && secret2 > 0
   p1_e := p1 && !(secret1 > 0)
   p2_e := p2 && !(secret2 > 0)
   if (p1_t) { res1 := val1 + 1 }
   if (p2_t) { res2 := val2 + 1 }
   if (p1_e) { res1 := val1 }
   if (p2_e) { res2 := val2 }
}
```

Figure 2.3: Modular product program of the example in Figure 2.2.

always take the same branch of the conditional, and therefore the results can be proven to be equal in both executions.

For the formal definition of the encoding, which we will extend in Chapter 3, including loops and method calls, refer to the MPP paper [12].

2.4 Viper and Nagini

This section gives a quick overview of the Viper verification infrastructure [18], as well as Nagini [11], a Python frontend for Viper.

Viper is a verification infrastructure based on modular deductive verification and permission-based reasoning. The modular verification is based on specifications such as pre- and postconditions for methods and loop invariants. To reason about heap memory, Viper makes use of Implicit Dynamic Frames [4], i.e., each location in memory is associated with a permission. A method needs some positive permission amount to be able to read a memory location, or a full permission to modify it.

Viper defines its own intermediate verification language, called the Viper language. Frontends can encode higher-level languages into the Viper language, and Viper will use its backends for the verification. For the actual verification Viper provides two backends, one based on verification condition generation and one based on symbolic execution. The former encodes the Viper program into Boogie [15] for verification.

Viper supports verification constructs, such as predicates [19], an abstraction over assertions, and pure functions, which can be used in specifications.

A predicate can be exchanged for its body by using `fold` and `unfold` statements. Predicates can be defined recursively and need to be self-framing, meaning that only heap accesses are allowed where the assertion carries the permissions (e.g., `acc(x.f) ==> x.f == 0` is fine, but `x.f == 0` alone is not). They provide an intuitive way of expressing abstraction, e.g., for modelling abstract data types and objects, and can be transferred between methods via pre- and postconditions. Pure functions in Viper can have pre- and postconditions and a body which has to be an expression. This means they can, for example, contain calls to other functions or recursively call themselves, but cannot contain loops. A pure function can therefore not modify the state and is deterministic. If a function depends on the heap it needs to get all necessary permissions in the precondition, and if the permissions are wrapped in predicates, it needs to use an `unfolding` expression to access the predicates body.

Nagini is a frontend for encoding Python programs to Viper. It requires static type annotations on all method signatures, and enforces typing rules before doing an encoding. It defines a specification language for writing contracts in Python, which are used to verify functional correctness.

To manage Python's dynamically bound calls, Nagini enforces behavioral subtyping [17], meaning that an overriding method can only have weaker preconditions and stronger postconditions. It enforces this by introducing a new method for each overriding method, which has the specification of the overridden method, and in its body calls the overriding method. If this passes the verification we can be sure that behavioral subtyping is adhered to. A dynamically bound call is then translated to a statically bound call to the method of the static type of the receiver.

Chapter 3

Modular Product Program Extensions

In this chapter we discuss the extensions we made to the original definition of modular product programs [12]. The reason why we have to extend MPP's is that they were originally only defined for a very small language. This language supports method calls, if statements, assignments and while loops. Since our goal is to apply them to programs written in a subset of Python programs, these will not suffice. Note, however, that this chapter is not specific to Python programs, but a more general definition of the MPP encoding, which should be suitable for many common object-oriented languages. The challenge in extending the MPP definition are statements that influence the program's control flow. This is non-trivial and needs special handling in the MPP. In particular the extensions we made concern the following features:

Return statements The challenge in encoding return statements is that they are jumps, which do not mix well with interleaved executions. We discuss how to encode the control flow in Section 3.1.

Loops The original definition of MPPs already defines an encoding for while loops, however, we will have to adapt it to support more complex control flow, i.e., return statements in the loop, `break` and `continue`, as well as raising of exceptions. We consider only `while` loops, since `for` loops can be rewritten as `while` loops quite easily. The encoding of loops and related control flow is defined in Section 3.2.

Exceptions We define an encoding for raising exceptions, as well as try/catch blocks in Section 3.3.

Dynamically bound calls Dynamic method binding introduces some challenges for the MPP, as a call can reference different implementations in different executions. We discuss how we handle these in Section 3.4.

Verification Constructs We add support for pure functions and predicates in Section 3.6.

For all of these extensions we adhere to two principles:

- We do not rely on any additional specifications from the user which would not make sense on the source program level. This means we only require the functional specifications and the relational ones, e.g., specifying which values are low. The user should *not* be asked for additional information, for example under which conditions we break out of a loop.
- The encoding is not required to produce an operationally equivalent program, rather is is enough that the product produces the same verification result.

In Section 3.5 we show how we deal with fields of objects, which is not an extension but rather making explicit how the original implementation of MPPs handles these.

Even though we are mostly interested in secure information flow, a 2-relational property, we define the encoding for an arbitrary number of executions, so that k-relational properties can be verified. In the subsequent chapters we will only consider the special case of $k = 2$.

There are some adjustments we made to the original encoding which do not merit their own section in this chapter. These adjustments mostly address the new way we encode control flow. The complete extended MPP encoding can be found in Appendix A.

3.1 Return Statements

The obvious way to encode a `return` statement is to assign the result variable, followed by a `goto` to the very end of the method body. Unfortunately, in an MPP this cannot work, as we have two interleaved executions which are simulated in one, such that if one simulated execution jumps, the simulation skips the other execution as well. This means we could not express a situation where only one of the simulated executions returns. For this reason, we will not use any `goto`'s in the encoding.

Another idea would be that after assigning to the result variable, we assert the postcondition, followed by an **`assume`** `false`. The idea here is that after the return statement the postcondition has to hold, and with the **`assume`** `false` we ensure that the trace which returned is ignored afterwards. The problem with this is that it will not work for relational postconditions where the control flow is not low. For example consider the code in Figure 3.1.

We expect this example to verify since we always get the same result, but in the encoding of line 4 we cannot prove that the result values are equal in both executions. In the product we could have that, e.g., $\texttt{secret}^{(1)}$ is true, but not $\texttt{secret}^{(2)}$, in which case $\texttt{res}^{(2)}$ is not yet defined at that point. Using

```
method test(secret: Bool) returns (res: Int)
   ensures low(res)
{
   if (secret) {
      return 1
   } else {
      return 1
   }
}
```

Figure 3.1: Return example.

this approach would in this case report an error, which means we would introduce an incompleteness in the encoding. We only have all the necessary information after considering all possible ways the method can return a result. Another reason why this approach does not work is that a `return` can happen inside a `try/finally` statement, such that we have to consider what happens in the `finally` block before asserting the postcondition.

Therefore, we introduce Boolean control variables to model the control flow. There are several reasons why a program execution could not reach a certain point in the program, namely because it already returned, it skips a section after a `break` or `continue` statement, or because an exception was raised and not yet caught. For each of these reasons we introduce a set of variables, one per execution, representing whether or not the corresponding execution skips a part of the program for this reason. We set them to `false` at the beginning of the method body, which gives us the new encoding of a procedure, shown in Figure 3.2.

The $\odot$ symbol represents sequential composition, $\mathtt{x}^{(i)}$ is the renamed version of variable `x` from the source program for execution i, where the new name must not clash with any other name in the product. $[\![s]\!]_k^{\mathring{p}}$ denotes the MPP of statement s for k executions, with $\mathring{p}$ being the activation variables. We use $\mathring{e}$ as an abbreviation of $e^{(1)}, \ldots, e^{(k)}$. $fresh(\mathtt{x}_1,\ \mathtt{x}_2,\ \ldots)$ denotes that the variable names $\mathtt{x}_1,\ \mathtt{x}_2,\ \ldots$ are fresh, meaning they have not been used before and do not occur in the program.

Note that unlike the original definition we only support methods with two return values, namely a `result` and an `error`. The `result` variable is used for the method's returned value, the `error` variable will hold any exceptions raised by the method.

We only execute statements of an execution if all of its control variables are `false`. This means that where in the original encoding we used the activation variables as guards for executing statements, we now use the activation variables conjoined with the negations of all its control variables.

```
$[\![$procedure m(x$_1$,...,x$_m$) returns (result, error){s}$]\!]_k$
  = procedure m(p$^{(1)}$,...,p$^{(k)}$,$args$) returns ($rets$){
        $\bigodot_{i=1}^{k}$ ret$^{(i)}$:= false;      // true iff returned
        $\bigodot_{i=1}^{k}$ break$^{(i)}$:= false;    // true iff after break
        $\bigodot_{i=1}^{k}$ cont$^{(i)}$:= false;     // true iff after continue
        $\bigodot_{i=1}^{k}$ except$^{(i)}$:= false;   // true iff uncaught exception
        $\bigodot_{i=1}^{k}$ error$^{(i)}$:= null;     // store exception
        $[\![s]\!]_k^{\mathring{p}}$
    }
    where
    $fresh(\mathring{\text{ret}}) \land fresh(\mathring{\text{break}}) \land fresh(\mathring{\text{cont}}) \land fresh(\mathring{\text{except}}) \land$
    $fresh(\mathring{\text{error}})$
    $args$ = x$_1^{(1)}$,...,x$_1^{(k)}$,...,x$_m^{(1)}$,...,x$_m^{(k)}$
    $rets$ = result$^{(1)}$,...,result$^{(k)}$,error$^{(1)}$,...,error$^{(k)}$
```

Figure 3.2: Procedure encoding.

```
$[\![$x:=e$]\!]_k^{\mathring{p}}$
  = $\bigodot_{i=1}^{k}$ if (p$^{(i)} \land \neg$ret$^{(i)} \land \neg$break$^{(i)} \land \neg$cont$^{(i)} \land \neg$except$^{(i)}$) then {
            x$^{(i)}$:=e$^{(i)}$
        }
$[\![$return z$]\!]_k^{\mathring{p}}$
  = $\bigodot_{i=1}^{k}$ if (p$^{(i)} \land \neg$ret$^{(i)} \land \neg$break$^{(i)} \land \neg$cont$^{(i)} \land \neg$except$^{(i)}$) then {
            result$^{(i)}$:=z$^{(i)}$;
            ret$^{(i)}$:=true
        }
```

Figure 3.3: Encoding of assignments and return statements.

The return statement is encoded as an assignment to the method's result variable, plus setting the ret flag to true. The new encoding of assignments and return statements is shown in Figure 3.3.

The introduction of control flow variables also has an impact on the encoding of specifications. In the MPP paper the encoding of specifications

```
i := 0
sum := 0
while (true)
   invariant sum <= x
{
   sum := sum + i
   if (sum > x) {
      return sum
   }
   i := i + 1
}
```

Figure 3.4: Loop with return statement.

is denoted as $\lfloor a \rfloor_k^{\mathring{p}}$, where a is the assertion to be encoded and $\mathring{p}$ represents all the activation variables. The activation variables are used to ensure that an assertion is only relevant when the corresponding execution is active. In our case what we want to express for assertions in the method body is that they are relevant when the execution is active *and* all the control flow flags are false. To make the distinction clearer we will denote the encoding of specifications as $\lfloor a \rfloor_k^{\mathring{act}}$, which is the same encoding but makes more explicit that *act* are not just the activation variables but can be any expression. In particular in pre- and postconditions, where the control flow flags are not defined, we will encode assertions with $act^{(i)} = \mathtt{p}^{(i)}$, but in the method body we will use $act^{(i)} = \mathtt{p}^{(i)} \wedge \neg\mathtt{ret}^{(i)} \wedge \neg\mathtt{break}^{(i)} \wedge \neg\mathtt{cont}^{(i)} \wedge \neg\mathtt{except}^{(i)}$.

3.2 Loops

We consider a verification methodology where a loop is handled such that outside the loop only the invariants are known, not the loop body. The loop body is only relevant for proving the invariants. Additionally, we assume that all the loop targets, the locations that get assigned to in the loop, get havoced. This means that after a loop we lose all information about modified locations, and we only learn the invariants and the negation of the loop condition.

The original MPP encoding defines how to encode `while` loops, however, there it is assumed that a loop can only be exited when the loop condition is false. We can no longer make this assumption, as the fact that we can have return statements, breaks, and exceptions means that we can in principle exit the loop at any point. As a consequence, we cannot assume that the loop condition is false immediately after the loop, and the loop invariant might be violated.

Figure 3.4 shows an example illustrating these problems. In a loop we increment the `sum` variable until it is larger than some value `x`. After the loop the loop condition will still be true, and the invariant will not hold. We still expect to be able to verify this code, because the invariant has to hold at the end of each iteration which is the case here, since when `sum` is greater than `x` we return and, therefore, do not reach the end of this iteration. Additionally, we expect to be able to prove the postcondition that `sum` is greater than `x`.

The desired behavior is difficult to encode using just the control flow variables, but fortunately, as mentioned at the beginning of the chapter, we do not need to get an equivalent program, rather it is enough to have one with equivalent verification result. The idea is that we treat the traces which exit the loop normally, meaning via the loop condition being false, separately from those traces which exit the loop via `return`, `break` or an exception. The former traces are covered in the while loop, with modified loop condition and invariants, as explained in the following paragraphs. The latter traces will be treated in what we call the *loop reconstruction*. The idea here is that after the loop we artificially reconstruct the verification state as it was before reaching the loop, then execute the loop body once to collect information for all traces. At the end of the reconstruction we remove all the traces which did actually exit the loop normally, using an `inhale` statement.

To inhale an assertion a means to assume that it holds, as well as gaining all the permissions of any heap access expressions in a. If a does not contain any heap accesses `inhale` a is the same as `assume` a. The counterpart to inhaling is exhaling an assertion a, which means to assert that a holds and all permissions a contains are held, followed by removing all these permissions. Exhaling a pure assertion is the same as an `assert`.

We conjoin the loop condition with the negation of the `ret`, `break` and `except` control variables, since these three represent a way we could exit the loop. This way the verifier does not know that the original loop condition is false after exiting the loop.

Since we have to allow the invariant to be violated in case that we exit the loop before the end of an iteration, we encode it such that it is conditional on the control flow flags not being set. As a consequence, we lose all information for traces which exit the loop other than via the condition being false (e.g., via returning, where one flag will be set). To learn what happened for these traces, we later add the loop reconstruction.

The complete loop encoding is shown in Figure 3.5. Before the loop we create a pair of variables called `bypass`, which encode whether an execution will not execute the loop. In that case we would lose all information about variables which are assigned inside the loop body as they get havoced. To keep that information we add invariants that those variables remain unchanged if the loop is bypassed. Note that this includes control variables which get assigned in the loop.

```
$[\![\texttt{while}\ (c)\ \texttt{invariant}\ inv\ \texttt{do}\ \{s\}]\!]_k^{\mathring{p}}$
  $=$  $\bigodot_{i=1}^{k} \mathsf{bypass}^{(i)} := \neg\mathsf{p}^{(i)} \lor \mathsf{ret}^{(i)} \lor \mathsf{break}^{(i)} \lor \mathsf{cont}^{(i)} \lor \mathsf{except}^{(i)};$
       $\bigodot_{i=1}^{k}(\texttt{if}\ (\mathsf{bypass}^{(i)})\ \texttt{then}\ \{\bigodot_{t\in Targets} \mathsf{tmp_t}^{(i)} := t\};$
       $\bigodot_{i=1}^{k} \{\mathsf{oldret}^{(i)} := \mathsf{ret}^{(i)};$   $\mathsf{oldbreak}^{(i)} := \mathsf{break}^{(i)};$
             $\mathsf{oldcont}^{(i)} := \mathsf{cont}^{(i)};$   $\mathsf{oldexcept}^{(i)} := \mathsf{except}^{(i)};\}$
       $\texttt{while}\ (\bigvee_{i=1}^{k}(\mathsf{p}^{(i)} \land \neg\mathsf{bypass}^{(i)} \land \neg\mathsf{ret}^{(i)} \land \neg\mathsf{break}^{(i)} \land \neg\mathsf{except}^{(i)} \land \mathsf{c}^{(i)}))$
        $\texttt{invariant}\ \lfloor inv \rfloor_k^{\mathsf{p}^{(i)} \land \neg\mathsf{ret}^{(i)} \land \neg\overset{\circ}{\mathsf{break}}{}^{(i)} \land \neg\mathsf{except}^{(i)}}$
        $\texttt{invariant}\ \bigwedge_{i=1}^{k}(\bigodot_{t\in Targets} \mathsf{bypass}^{(i)} \Rightarrow \mathsf{tmp_t}^{(i)} = t)$
        $\texttt{do}\ \{$
          $\bigodot_{i=1}^{k} \mathsf{cont}^{(i)} := \mathsf{false};$
          $\bigodot_{i=1}^{k} \mathsf{p_1}^{(i)} := \mathsf{p}^{(i)} \land \neg\mathsf{ret}^{(i)} \land \neg\mathsf{break}^{(i)} \land \neg\mathsf{except}^{(i)} \land \mathsf{c}^{(i)};$
          $[\![s]\!]_k^{\mathring{p}_1}$
          $\bigodot_{i=1}^{k} \texttt{inhale}\ \neg\mathsf{p}^{(i)} \lor (\neg\mathsf{ret}^{(i)} \land \neg\mathsf{break}^{(i)} \land \neg\mathsf{except}^{(i)})$
       $\}$
       $\texttt{if}\ (\bigvee_{i=1}^{k}(\neg\mathsf{bypass}^{(i)} \land (\mathsf{ret}^{(i)} \lor \mathsf{break}^{(i)} \lor \mathsf{except}^{(i)})))\ \texttt{then}\ \{$
          $\bigodot_{i=1}^{k} \{\mathsf{ret}^{(i)} := \mathsf{oldret}^{(i)};$   $\mathsf{break}^{(i)} := \mathsf{oldbreak}^{(i)};$
                $\mathsf{cont}^{(i)} := \mathsf{oldcont}^{(i)};$   $\mathsf{except}^{(i)} := \mathsf{oldexcept}^{(i)};\}$
          $\texttt{inhale}\ \bigwedge_{i=1}^{k}(\mathsf{p}^{(i)} \land \neg\mathsf{ret}^{(i)} \land \neg\mathsf{break}^{(i)} \land \neg\mathsf{except}^{(i)} \Rightarrow \mathsf{inv}^{(i)})$
          $\texttt{inhale}\ \bigwedge_{i=1}^{k}(\mathsf{p}^{(i)} \land \neg\mathsf{ret}^{(i)} \land \neg\mathsf{break}^{(i)} \land \neg\mathsf{except}^{(i)} \Rightarrow \mathsf{c}^{(i)})$
          $\bigodot_{i=1}^{k} \mathsf{cont}^{(i)} := \mathsf{false};$
          $\bigodot_{i=1}^{k} \mathsf{p_1}^{(i)} := \mathsf{p}^{(i)} \land \neg\mathsf{ret}^{(i)} \land \neg\mathsf{break}^{(i)} \land \neg\mathsf{except}^{(i)} \land \mathsf{c}^{(i)};$
          $[\![s]\!]_k^{\mathring{p}_1};$
          $\bigodot_{i=1}^{k} \texttt{inhale}\ \neg\mathsf{p_1}^{(i)} \lor \mathsf{ret}^{(i)} \lor \mathsf{break}^{(i)} \lor \mathsf{except}^{(i)}$
       $\}$
       $\bigodot_{i=1}^{k} \texttt{if}\ (\neg\mathsf{bypass}^{(i)})\ \texttt{then}\ \{\mathsf{break}^{(i)} := \mathsf{false}; \mathsf{cont}^{(i)} := \mathsf{false}\}$
       where
       $Targets$ are all variables assigned to in loop (including control flow flags),
       $fresh(\mathring{\mathsf{p}}_1) \land fresh(\mathring{\mathsf{bypass}}) \land \forall t \in Targets.fresh(\mathring{\mathsf{tmp}}_\mathsf{t})$
```

Figure 3.5: Encoding of loops.

We conjoin the loop condition with the negation of the `bypass` variable and all the control flow variables, except the one for `continue`. This one is special in the sense that it is the only one which can be set inside the loop without causing the execution to exit the loop. This is also why we set it to `false` at the beginning of the loop body.

After the loop we add the loop reconstruction. It is wrapped in a conditional statement, since we only have to do a reconstruction for those traces where the loop was not bypassed and which exited the loop other than via the loop condition being false. To get into the same verification state from before reaching the loop, we first set the control flow variables to the values they had there. If, for example, an execution returned before reaching the loop, then we set the `ret` flag of this execution to true here and thus, as expected, nothing is executed in the reconstruction. Alternatively, if an execution returned inside the loop body, we set the corresponding `ret` flag to false (we know the old value to be false, otherwise it could not have returned in the loop body), so that in the reconstruction the statements preceding the return are executed. Finally, if it did not return at all, we also set the `ret` flag to false and execute the loop body in the reconstruction as expected.

We then inhale all the loop invariants and add the loop body once more. After the body we inhale that some control flow flag is set, to remove all those traces which did exit the loop normally (we already have all the information for those from the loop invariants).

In the last line we reset the break and continue control variables, in case the loop was not bypassed. If it was bypassed, maybe because this is a nested loop and the outer one already hit a break or continue, we have to leave them as they are, since in this case this loop was not executed and thus we could not have changed the flags. If it was not bypassed on the other hand, we set them to false, because if they are true at this point, it means they were set in this loop (if they were true before we would have bypassed the loop) and the execution is no longer inactive after the loop. We do this for these two variables only here, because their effect is local to the loop, whereas, for example, a return statement will affect control flow on the method level.

The encoding of `break` and `continue` is now straightforward (see Figure 3.6). In both cases it is simply an assignment to the corresponding control flow variable.

3.3 Exception Handling

In this section we add an encoding for raising exceptions, as well as try/catch/finally blocks, two features which are absent in the original encoding, but very important for real programming languages.

```
⟦break⟧$_k^{\mathring{p}}$
  = $\bigodot_{i=1}^{k}$ if ($\mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)}$) then {
          $\mathtt{break}^{(i)}$:=true
      }
⟦continue⟧$_k^{\mathring{p}}$
  = $\bigodot_{i=1}^{k}$ if ($\mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)}$) then {
          $\mathtt{cont}^{(i)}$:=true
      }
```

Figure 3.6: Encoding of break and continue statements.

```
⟦raise e⟧$_k^{\mathring{p}}$
  =  $\bigodot_{i=1}^{k}$ if ($\mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)}$) then {
           $\mathtt{error}^{(i)}$:= new ();
           inhale typeof($\mathtt{error}^{(i)}$) = typeof(e)
           $\mathtt{except}^{(i)}$:=true
       }
```

Figure 3.7: Encoding of raise statements.

Figure 3.7 shows the encoding of a raise statement. It is very similar to the encoding of return statements, except that instead of assigning the result variable we assign to the error variable, and instead of setting the ret flag we set the except flag. Additionally we have to remember the type of the error variable to be the type of the exception raised. We do this by inhaling the type, using a helper function typeof that maps references to types. We can then use the same function in the catch blocks to match the error variable against caught exception types.

The try/catch statement encoding is shown in Figure 3.8. We support try/catch blocks with any number of except clauses, as well as an else and a finally block. The else block—as supported by Python—is executed iff there was no exception raised inside the try block.

Similar to the while loop we first define bypass variables, to express that the whole block is skipped by an execution. We then save the values of the control flow flags, which we will need later for the finally block. After the encoded try body, we store into new thisexcept variables whether or not

$$
\begin{aligned}
&[\![\textbf{try } \{s\} \textbf{ except } e_1 : \{s_1\} \ldots \textbf{ except } e_m : \{s_m\} \textbf{ else:}\{s_e\} \textbf{ finally:}\{s_f\}]\!]_k^{\mathring{p}} \\
= \; & \textstyle\bigodot_{i=1}^{k} \mathtt{bypass}^{(i)}{:=}\neg \mathtt{p}^{(i)} \vee \mathtt{ret}^{(i)} \vee \mathtt{break}^{(i)} \vee \mathtt{cont}^{(i)} \vee \mathtt{except}^{(i)}; \\
& \textstyle\bigodot_{i=1}^{k} \{\mathtt{oldret}^{(i)}{:=}\mathtt{ret}^{(i)}; \quad \mathtt{oldbreak}^{(i)}{:=}\mathtt{break}^{(i)}; \\
& \qquad \mathtt{oldcont}^{(i)}{:=}\mathtt{cont}^{(i)}; \quad \mathtt{oldexcept}^{(i)}{:=}\mathtt{except}^{(i)};\} \\
& [\![s]\!]_k^{\mathring{p}}; \\
& \textstyle\bigodot_{i=1}^{k} \mathtt{thisexcept_1}^{(i)}{:=}\mathtt{except}^{(i)} \wedge \neg \mathtt{bypass}^{(i)} \\
& \textstyle\bigodot_{i=1}^{k} \mathtt{p_1}^{(i)}{:=}\mathtt{p}^{(i)} \wedge \mathtt{thisexcept}^{(i)} \wedge \mathtt{issubtype}(\mathtt{typeof}(\mathtt{error}^{(i)}), e_1); \\
& \textstyle\bigodot_{i=1}^{k} \textbf{if } (\mathtt{p_1}^{(i)}) \textbf{ then } \{\mathtt{except}^{(i)}{:=}\mathtt{false}\}; \\
& [\![s_1]\!]_k^{\mathring{p}_1}; \\
& \ldots \\
& \textstyle\bigodot_{i=1}^{k} \mathtt{p}_m{}^{(i)}{:=}\mathtt{p}^{(i)} \wedge \mathtt{thisexcept}^{(i)} \wedge \mathtt{issubtype}(\mathtt{typeof}(\mathtt{error}^{(i)}), e_m); \\
& \textstyle\bigodot_{i=1}^{k} \textbf{if } (\mathtt{p}_m{}^{(i)}) \textbf{ then } \{\mathtt{except}^{(i)}{:=}\mathtt{false}\}; \\
& [\![s_m]\!]_k^{\mathring{p}_m} \\
& \textstyle\bigodot_{i=1}^{k} \mathtt{p}_{m+1}{}^{(i)}{:=}\mathtt{p}^{(i)} \wedge \neg \mathtt{thisexcept}^{(i)}; \\
& [\![s_e]\!]_k^{\mathring{p}_{m+1}} \\
& \textstyle\bigodot_{i=1}^{k} \textbf{if } (\mathtt{p}^{(i)}) \textbf{ then } \{ \\
& \qquad\qquad \mathtt{tmp_{ret}}^{(i)} := \mathtt{ret}^{(i)}; \qquad \mathtt{ret}^{(i)} := \mathtt{oldret}^{(i)}; \\
& \qquad\qquad \mathtt{tmp_{break}}^{(i)} := \mathtt{break}^{(i)}; \qquad \mathtt{break}^{(i)} := \mathtt{oldbreak}^{(i)}; \\
& \qquad\qquad \mathtt{tmp_{cont}}^{(i)} := \mathtt{cont}^{(i)}; \qquad \mathtt{cont}^{(i)} := \mathtt{oldcont}^{(i)}; \\
& \qquad\qquad \mathtt{tmp_{except}}^{(i)} := \mathtt{except}^{(i)}; \qquad \mathtt{except}^{(i)} := \mathtt{oldexcept}^{(i)}; \\
& \qquad \} \\
& [\![s_f]\!]_k^{\mathring{p}}; \\
& \textstyle\bigodot_{i=1}^{k} \textbf{if } (\mathtt{p}^{(i)}) \textbf{ then } \{ \\
& \qquad\qquad \mathtt{ret}^{(i)} := \mathtt{ret}^{(i)} \vee \mathtt{tmp_{ret}}^{(i)}; \\
& \qquad\qquad \mathtt{break}^{(i)} := \mathtt{break}^{(i)} \vee \mathtt{tmp_{break}}^{(i)}; \\
& \qquad\qquad \mathtt{cont}^{(i)} := \mathtt{cont}^{(i)} \vee \mathtt{tmp_{cont}}^{(i)}; \\
& \qquad\qquad \mathtt{except}^{(i)} := \mathtt{except}^{(i)} \vee \mathtt{tmp_{except}}^{(i)}; \\
& \qquad \} \\
& \text{where} \\
& \forall i \in [1, m+1].\mathit{fresh}(\mathtt{p}_i) \wedge \\
& \forall x \in \{\mathtt{by\mathring{p}ass}, \mathtt{ol\mathring{d}ret}, \mathtt{old\mathring{b}reak}, \mathtt{old\mathring{c}ont}, \mathtt{olde\mathring{x}cept} \\
& \quad \mathtt{t\mathring{m}p_{ret}}, \mathtt{t\mathring{m}p_{break}}, \mathtt{t\mathring{m}p_{cont}}, \mathtt{t\mathring{m}p_{except}}\}.\mathit{fresh}(x)
\end{aligned}
$$

Figure 3.8: Encoding of try/except/else/finally blocks.

there is an uncaught exception which happened inside this try block. For every exception handler *h* we then add a sequence of statements:

- We create a new activation variable $\mathsf{p}_h{}^{(i)}$, which expresses whether we execute this handler or not. This is the case iff execution *i* is active, there is an uncaught exception from this try block and the type of the error which was raised is a subtype of the error handler *h* accepts.

  The reason we have to use a `thisexcept` variable, instead of just relying on the `except` flag, is that it is possible that a handler *i* raises an exception of a type which handler *j*, where $j > i$, would catch. In that case we do not want to execute handler *j*, because of the semantics of try/catch statements. For checking if the handler accepts the error type, we use an `issubtype` function.

- We set the `except` flag to false, because at that moment there no longer is an uncaught exception around. We have to do this before we execute the handler, because the encoding of the handler will again use the same flags, so if we leave it true the handler will not do anything. Also, this way we have the flag available for raising an exception inside the handler.

- We encode the handler using the activation variables $\mathsf{p_h}{}^{(i)}$.

After the handlers we add the else block. It is encoded using new activation variables $\mathsf{p_{m+1}}{}^{(i)}$, which express that the execution is active and there was no exception inside the try body.

The finally block is special, since it gets executed whenever the try/catch statement was not completely bypassed, no matter what happened in the try block, the handlers or the else block. If, however, the execution was inactive before the try/catch statement, such that we never entered it, the finally block will not be executed. We can encode this by setting the values of the control flow variables to what they were before the statement. After the finally block, we then disjoin the control flow flags with what they were before the finally and assign this disjunction to the flags. This way they will be set if, e.g., a return happened in the try block or a handler or the finally block.

3.4 Dynamically Bound Calls

In this section we discuss the issue of dealing with dynamically bound calls. As a fundamental feature of many object-oriented languages, we need to support this. We will require programs to adhere to behavioral subtyping, meaning that an overriding method can only have weaker preconditions and stronger postconditions than the overridden method. This property can be verified as follows: For every method `Sub::foo` which overrides a

```
class A:
   def foo(self) -> int:
      Requires(LowEvent())
      Ensures(Low(Result()))
      print("A")
      return 0

class B(A):
   def foo(self) -> int:
      """ Overrides A::foo. """
      Requires(LowEvent())
      Ensures(Low(Result()))
      print("B")
      return 1

def test(secret: bool) -> None:
   Requires(LowEvent())
   if secret:
      a = A()
   else:
      a = B()
   x = a.foo() # has to fail, print reveals 'secret'
   Assert(Low(x)) # has to fail, could be 0 or 1
```

Figure 3.9: Example of dynamically bound methods (Python with Nagini contracts).

method `Super::foo`, a new method is created with pre- and postconditions of `Super::foo` and in the body calls `Sub::foo`. If we can verify this new method we know that the preconditions of `Super::foo` imply the ones of `Sub::foo`, and the postconditions of `Sub::foo` imply the postconditions of `Super::foo`. At the call site, a dynamically bound call to the method `foo` is now encoded as a statically bound call to `Super::foo`. This is sound, because we showed the contracts of `Sub::foo` to be compatible.

Unfortunately, this is not enough to ensure secure information flow, as is shown in the example in Figure 3.9. We have a class `B` inheriting from class `A` and overriding the method `foo`. `foo` requires `lowEvent`, which expresses that a call to the method happens in either all or none of the executions. The standard encoding of `lowEvent` is that the values of the activation variables are equal across executions. Here it is needed for calling `print`, which will reveal information on public outputs and, therefore, must not happen in only one execution. In both classes `foo` has exactly the same specification, so behavioral subtyping is adhered to. In method `test` we then create an object such that the type depends on a secret and call `foo` on it. Clearly the verification of the call should fail, because from observing the output we learn the value of `secret` (depending on whether we see "A" or "B").

$$
\begin{aligned}
\lfloor \texttt{lowEvent} \rfloor_{2}^{\overset{\circ}{act}} &= act^{(1)} = act^{(2)} \\
\lfloor \texttt{lowEvent} \rfloor_{2,dyn}^{\overset{\circ}{act}} &= (act^{(1)} = act^{(2)}) \wedge \\
&\quad (act^{(1)} \wedge act^{(2)} \Rightarrow \texttt{typeof}(\texttt{self}^{(1)}) = \texttt{typeof}(\texttt{self}^{(2)})) \\
\lfloor \texttt{low(exp)} \rfloor_{2}^{\overset{\circ}{act}} &= act^{(1)} = act^{(2)} \Rightarrow \texttt{e}^{(1)} = \texttt{e}^{(2)} \\
\lfloor \texttt{low(e)} \rfloor_{2,dyn,post}^{\overset{\circ}{act}} & \\
= [act^{(1)} \wedge act^{(2)} &\Rightarrow (\texttt{typeof}(\texttt{self}^{(1)}) = \texttt{typeof}(\texttt{self}^{(2)}) \Rightarrow \texttt{e}^{(1)} = \texttt{e}^{(2)}), \\
act^{(1)} \wedge act^{(2)} &\Rightarrow \texttt{e}^{(1)} = \texttt{e}^{(2)}]
\end{aligned}
$$

Figure 3.10: Encoding of lowEvent and low both standard and in dynamically bound methods.

Additionally, in line 23 we should not be able to assert that the result is low, even though both implementations promise this. The value will again depend on the value of secret, being 0 or 1. We now discuss the measures we take to make the encoding of relational assertions sound again, on the examples of `lowEvent` and `low()`.

If we look at the precondition of `foo`, what we want to express is that either both executions make the call or none do. In this setting this not guaranteed with the aforementioned encoding of `lowEvent`, namely that both activation variables must be equal, since the actual method we call depends on the type of the receiver. What we do to solve this is change the encoding of `lowEvent` in methods which can be the target of dynamically bound calls to also require the types of `self` to be equal, see Figure 3.10. Now we can guarantee that in both executions the same method will be called, because the receiver types are equal, making the encoding of `lowEvent` sound.

The issue with low is similar. When a client calls `x.foo()` they will assume that the result is low, which again depends on the type of the receiver `x`. To make our implementation sound we change the encoding of `low()` in postconditions of dynamically bound methods as defined in Figure 3.10. We make use of an *InhaleExhale* expression, a construct where the first expression is inhaled, the second exhaled. In this case this means that the method which has the postcondition has to prove the second part, which is the standard encoding of `low()`, but the caller only inhales the first part, which says that the two expressions are equal in both executions under the condition that the type of the receiver variable is low.

Note that in both cases we are more restrictive than strictly necessary. Our encoding is sound, but there are correct programs which we will reject with this encoding. For example, if in the program in Figure 3.9 we add another method to class `A` which we do not override in `B`, then we would ask a client to show that receiver types are low, even though the same method will

```
method main(secret: Int) returns (res: Ref)
   ensures low(res)
   ensures acc(res.f) && low(res.f) // low(res.f) must fail
{
   res := new(f) // create reference with field f
   res.f := secret
}
```

Figure 3.11: Example of why fields are duplicated (Viper).

be called no matter what. This shows that the requirement of types being equal is incomplete, however, we argue that in practice this would hardly be a problem, as situations where types of variables depend on secrets are rare.

In order to be more precise we could add the information which version of a method an object calls to the encoding. Then we would change the requirement such that the method versions must be equal on both receivers. This would allow for more correct programs to be verified, but would still be incomplete. We could, for example, override a method with an equivalent one, in which case the code would be correct, but the called method would be different and we still raise an error. In order to support this we would have to prove relational specifications of two different implementations, which our version of MPPs cannot do.

3.5 Heap Memory

This section explains how we encode heap memory, in particular fields of objects. This is the same as the original MPP implementation handles them, but we state it explicitly here as it will be important in the next section.

In the encoding all fields get k copies, so that each execution gets its own version. The reason is that we want to be able to express that both references and fields of references are low. An illustrating example can be found in Figure 3.11, we consider the case for $k = 2$. The method `main` creates a new object and assigns its (secret) argument to the field f. Since we want to prove that the returned reference is low, we need to encode `new` in such a way that both executions get the same reference. We can then prove that the result is low, because the creation of the object does not depend on any secret data, but the value of the field will be secret, so the second postcondition must fail. If we only had one field f, in the encoding we would first assign `res`$^{(1)}$`.f := secret`$^{(1)}$ and then `res`$^{(2)}$`.f := secret`$^{(2)}$. Since we know `res` is low, we would override the same location with the second assignment, so both would be equal and we could erroneously prove the postcondition. Using copies of the fields solves this issue.

3.6 Verification Constructs

In this section we discuss the encoding of verification constructs as used in verifiers based on implicit dynamic frames. In particular we look at pure functions and predicates, as they are supported by the Viper framework and Nagini. Unlike what we looked at so far, these constructs are not common object-oriented language features, but rather tools to help the verification.

3.6.1 Pure Functions

Pure functions are free of side effects, i.e., they cannot change the state of the heap. A function can have pre- and postconditions, and a body which is an expression. Statements (e.g., loops) are not allowed, but functions may be recursive and call other pure functions. The purpose of pure functions in verification is to be used in specifications, which should not have side effects.

We made a design decision that functions can never have any relational specifications. The main reason is that the verifier can inspect a function body where it is called, as opposed to methods which are abstracted via pre- and postconditions. This is equivalent to having a precise functional specification, which makes relational specifications obsolete. If we know exactly what is happening in each execution, there is no need to specify how executions behave relative to one another. Additionally, encoding relational functions would be messy, since a function can only have one value. Thus, if we want to encode them similar to procedures where return values are duplicated, we would need to make the values tuples and encode each function evaluation as tuple accesses. Together with duplicating all arguments it would get rather convoluted, and we see no advantage of supporting relational functions.

Nonetheless, even with functions being unary, there is a need for an encoding, because functions can contain expressions depending on the heap, i.e., heap-dependent predicates (see next section), field accesses or calls to other heap-dependent functions.

As mentioned in Section 3.5, all fields get k copies in the product, and as a result of this we need to create copies of all those functions which in any way depend on the heap. The process is straightforward: the function version for the i-th execution references all the i-th versions of fields, heap-dependent functions and predicates. A function call is then encoded the same in all executions if the function was not duplicated, or uses the respective version if it was. The complete encoding can be found in Appendix A.

```
predicate P(x: Ref) {
   acc(x.f) && low(x.f)
}

method main(x: Ref, secret: Bool)
   requires P(x)
{
   if (secret) {
      unfold P(x)
      val = x.f
   }
}
```

Figure 3.12: Example of a predicate (Viper, simplified).

3.6.2 Predicates

A Viper predicate is an abstraction over an assertion and can be used in specifications. A predicate can be exchanged for its content by using an *unfold* statement, and when the assertion holds it can be exchanged for the predicate using a *fold* statement.

Contrary to functions, we want to be able to express relational assertions with predicates. Figure 3.12 shows a simple example of a predicate `P`, which expresses having access to a field `f` and that said field is low. It also shows a possible way of using `P` in the method `main`: We get access to `x.f` from the predicate and use it to read the value into the `val` variable. As with pure functions, one might try to encode them similarly to how we encode procedures, namely adding activation variables and duplicating everything. This would result in an encoding as shown in Figure 3.13. As we see, encoding the predicate is no problem, but when we try to use it we run into an issue. Since if-statements create new versions of activation variables, and we then use them in the respective branches, we try to unfold the predicate in line 11 with the new version of these variables. Even assuming we knew both `p1` and `p2` are true, we do not know about the values of `p1'` and `p2'`. Therefore, we cannot unfold `P` in line 11, as we might not have permission to an instance of the predicate with these arguments, and we do not get access to the field `f` for either execution.

From this example we can see that we want an encoding which allows us to fold and unfold predicates for each execution independently. Similar to functions, we will therefore duplicate predicates to make one version for each execution. These k predicates only contain the unary parts of the predicate, as they are only meaningful for one execution each. To express the relational part of the predicate we create a pure function of Boolean type in addition to the k predicates. This relational function requires access to all

```
predicate P(p1: Bool, p2: Bool, x1: Ref, x2: Ref) {
  (p1 ==> acc(x1.f1)) && (p2 ==> acc(x2.f2)) &&
  (p1 && p2 ==> x1.f1 == x2.f2)
}

method main(p1: Bool, p2: Bool,
            x1: Ref, x2: Ref, secret1: Bool, secret2: Bool)
  requires P(p1, p2, x1, x2)
{
  p1' := p1 && secret1
  p2' := p2 && secret2
  unfold P(p1', p2', x1, x2) // fails: insufficient
   permission
  if (p1') {val1 := x1.f1}
  if (p2') {val2 := x2.f2}
}
```

Figure 3.13: Possible encoding of Figure 3.12 (Viper, simplified).

versions of the predicate in its precondition, so that it can unfold them to get all required permissions for the relational expressions. A predicate access, as it would occur in a pre- or postcondition, is then encoded as the conjunction of the predicate accesses and the relational function. The resulting encoding is shown in Figure 3.14

Note that we again need a special encoding for the case that the predicate access is in the postcondition of a method to which calls are dynamically bound. This is for the same reasons discussed in Section 3.4, as a predicate can contain relational assertions. The measures we take to make the encoding sound are analogous, except that here we have to consider the general case for arbitrary k, whereas in Section 3.4 we only considered the case of $k = 2$ for the information flow assertions.

In the case that the predicate we encode is unary, meaning that it does not contain relational assertions in the body and any other predicates it references are unary as well, we do not have to create the relational function. This is because its body would either just be `true`, in which case it always holds trivially, or it contains function calls to the relational function of the referenced predicates. In the latter case, we know that since all reachable predicates are unary (by our assumption), the other function calls will never contain meaningful assertions. Therefore, we only generate the relational function where necessary, and where we do not generate it, the conjuncts in the encoding (Figure 3.14) which contain it are not included, and in folding and unfolding the additional checks explained below are not generated.

We can now define encodings for `unfold` and `fold` statements. Since we separated the predicate, we can unfold or fold the predicates for each

$$
\begin{aligned}
&[\![\texttt{predicate P(x}_1\texttt{,...,x}_m\texttt{)}\ \{a\}]\!]_k \\
&\quad = \bigodot_{i=1}^{k} \texttt{predicate P}^{(i)}\texttt{(x}_1{}^{(i)}\texttt{,...,x}_m{}^{(i)}\texttt{)}\ \{a^{(i)}\}; \\
&\qquad \texttt{function P}^{(rel)}\ \texttt{(x}_1{}^{(1)}\texttt{,...,x}_1{}^{(k)}\texttt{,...,x}_m{}^{(1)}\texttt{,...,x}_m{}^{(k)}\texttt{) : Bool} \\
&\qquad\ \texttt{requires}\ \textstyle\bigwedge_{i=1}^{k} \texttt{P}^{(i)}\texttt{(x}_1{}^{(i)}\texttt{,...,x}_m{}^{(i)}\texttt{)} \\
&\qquad \{ \\
&\qquad\quad \texttt{unfolding}\ \textstyle\bigwedge_{i=1}^{k} \texttt{P}^{(i)}\texttt{(x}_1{}^{(i)}\texttt{,...,x}_m{}^{(i)}\texttt{)}\ \texttt{in}\ \{a^{rel}\} \\
&\qquad \} \\
&\qquad \text{where} \\
&\qquad a^{(i)} \quad = \quad a \text{ without the rel expressions, using field versions } i \\
&\qquad a^{rel} \quad = \quad \text{relational expressions from } a\text{, as well as } \texttt{P}^{(rel)} \\
&\qquad\qquad\qquad\quad \text{if } \texttt{P} \text{ is recursive} \\
&[\![\texttt{P(x}_1\texttt{,...,x}_m\texttt{)}]\!]_k^{\mathring{p}} \\
&\quad = \textstyle\bigwedge_{i=1}^{k}(act^{(i)} \Rightarrow \texttt{P}^{(i)}\texttt{(x}_1{}^{(i)}\texttt{,...,x}_m{}^{(i)}\texttt{)})\wedge \\
&\qquad (\textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow \texttt{P}^{(rel)}\texttt{(x}_1{}^{(1)}\texttt{,...,x}_1{}^{(k)}\texttt{,...,x}_m{}^{(1)}\texttt{,...,x}_m{}^{(k)}\texttt{)}) \\
&\qquad \text{where} \\
&\qquad act^{(i)} = \begin{cases} \texttt{p}^{(i)} \wedge \neg\texttt{ret}^{(i)} \wedge \neg\texttt{break}^{(i)} \wedge \\ \qquad \neg\texttt{cont}^{(i)} \wedge \neg\texttt{except}^{(i)}, \text{ in method body} \\ \texttt{p}^{(i)}, \text{ otherwise} \end{cases} \\
&[\![\texttt{P(x}_1\texttt{,...,x}_m\texttt{)}]\!]_{k,post}^{\mathring{p}} \\
&\quad = \textstyle\bigwedge_{i=1}^{k}(act^{(i)} \Rightarrow \texttt{P}^{(i)}\texttt{(x}_1{}^{(i)}\texttt{,...,x}_m{}^{(i)}\texttt{)})\wedge \\
&\qquad [\,\textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow (\textstyle\bigwedge_{i=1}^{k-1}(\texttt{typeof(self}^{(i)}\texttt{)} = \texttt{typeof(self}^{(i+1)}\texttt{)})) \Rightarrow \\
&\qquad\qquad\quad \texttt{P}^{(rel)}\texttt{(x}_1{}^{(1)}\texttt{,...,x}_m{}^{(1)}\texttt{,x}_1{}^{(2)}\texttt{,...,x}_m{}^{(2)}\texttt{)}), \\
&\qquad\quad (\textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow \texttt{P}^{(rel)}\texttt{(x}_1{}^{(1)}\texttt{,...,x}_1{}^{(k)}\texttt{,...,x}_m{}^{(1)}\texttt{,...,x}_m{}^{(k)}\texttt{)})] \\
&\qquad \text{where} \\
&\qquad act^{(i)} = \begin{cases} \texttt{p}^{(i)} \wedge \neg\texttt{ret}^{(i)} \wedge \neg\texttt{break}^{(i)} \wedge \\ \qquad \neg\texttt{cont}^{(i)} \wedge \neg\texttt{except}^{(i)}, \text{ in method body} \\ \texttt{p}^{(i)}, \text{ otherwise} \end{cases}
\end{aligned}
$$

Figure 3.14: Encoding of predicate definitions and accesses.

$$
\begin{aligned}
&[\![\textbf{unfold}\ \mathsf{P}(\mathsf{x}_1,\ldots,\mathsf{x}_m)]\!]_k^{\mathring{p}}\\
&\quad= \textbf{assert}\ \ \textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow (\textstyle\bigwedge_{i=1}^{k}(\mathsf{perm}(\mathsf{P}^{(i)}(\mathsf{x}_1{}^{(i)},\ldots,\mathsf{x}_m{}^{(i)})) = \mathsf{write}) \Rightarrow\\
&\qquad\qquad\quad \mathsf{P}^{(rel)}(\mathsf{x}_1{}^{(1)},\ldots,\mathsf{x}_1{}^{(k)},\ldots,\mathsf{x}_m{}^{(1)},\ldots,\mathsf{x}_m{}^{(k)});\\
&\qquad \textstyle\bigodot_{i=1}^{k}\ \textbf{if}\ \ (act^{(i)})\ \ \textbf{then}\ \{\textbf{unfold}\ \mathsf{P}^{(i)}(\mathsf{x}_1{}^{(i)},\ldots,\mathsf{x}_m{}^{(i)})\}\\
&\qquad \text{where}\\
&\qquad act^{(i)} = \mathsf{p}^{(i)} \wedge \neg\mathsf{ret}^{(i)} \wedge \neg\mathsf{break}^{(i)} \wedge \neg\mathsf{cont}^{(i)} \wedge \neg\mathsf{except}^{(i)}\\
&[\![\textbf{fold}\ \mathsf{P}(\mathsf{x}_1,\ldots,\mathsf{x}_m)]\!]_k^{\mathring{p}}\\
&\quad= \textstyle\bigodot_{i=1}^{k}\ \textbf{if}\ \ (act^{(i)})\ \ \textbf{then}\ \{\textbf{fold}\ \mathsf{P}^{(i)}(\mathsf{x}_1{}^{(i)},\ldots,\mathsf{x}_m{}^{(i)})\}\\
&\qquad \textbf{assert}\ \ \textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow \mathsf{P}^{(rel)}(\mathsf{x}_1{}^{(1)},\ldots,\mathsf{x}_1{}^{(k)},\ldots,\mathsf{x}_m{}^{(1)},\ldots,\mathsf{x}_m{}^{(k)});\\
&\qquad \text{where}\\
&\qquad act^{(i)} = \mathsf{p}^{(i)} \wedge \neg\mathsf{ret}^{(i)} \wedge \neg\mathsf{break}^{(i)} \wedge \neg\mathsf{cont}^{(i)} \wedge \neg\mathsf{except}^{(i)}
\end{aligned}
$$

Figure 3.15: Encoding of Unfold and Fold statements.

execution under the condition that the execution is active, including not having any control flow flag set. Additionally, before we unfold we check that the relational parts of the predicate hold if both executions are active, by asserting that if we have permissions to all predicates the function value is true. We have to add the condition that we have permission here to make the expression consistent, as the relational function requires access to all predicates. In the case of `fold`, we assert the relational function after the two folds. This way we ensure that we have access to the predicates required in the function's precondition.

Note that in the case of `unfold` the assertion is not strictly necessary for soundness. If the relational parts of the predicates do not hold here, we will never assume them and, therefore, can never use them to prove any wrong assertions. The reason we still put the assertion there is because we want the `unfold` to fail when the relational parts do not hold. A user who writes `unfold P(. . . )` expects that afterwards all the contents of `P` are true, and if we do not put the assertion there, the verification might give an error later on because relational parts of the predicate do not hold, which can be difficult to debug.

Chapter 4

Secure Information Flow

In this chapter we discuss how we can use MPPs in verification of secure information flow in Nagini. We show how we extended the specification language of Nagini and how these specifications are encoded. We then look at obligations [8] and how we complement them to prove the absence of termination channels. Finally, we describe what additional checks are needed to guarantee possibilistic noninterference in a concurrent setting.

4.1 Specifications

To express the relational specifications we need in order to verify secure information flow, we need to extend the specification language of Nagini by adding new functions to Nagini's contract library. We now list all the new assertion expressions, statements and decorators we added and explain how to use them and how they are encoded.

4.1.1 Expressions

The following expressions can only be used in assertions, i.e., method specifications, loop invariants and in arguments of the `Assert` contract statement.

`LowEvent()`

`LowEvent()` expresses that a certain point in the program must be reached by either both executions or none of them.

The user is only allowed to write this in the precondition of a method, to express that whether or not a call to the function happens must not depend on a secret. This is, for example, needed if a method prints something to public outputs. The encoding is as discussed in Section 3.4, namely that the two activation variables must be equal, and in the case of dynamic binding

```
def bool_int(secret: bool) -> int:
    Ensures(f_low(Result()))
    if secret:
        return 1
    return True
```

```
def unchanged(secret: int, x: int) -> int:
    Ensures(Implies(Low(x), f_low(Result())))
    if secret == 0:
        return x + secret
    return x
```

Figure 4.1: Two examples illustrating the difference between Low and LowVal (Nagini). Both verify successfully with f_{low} = LowVal, but fail with f_{low} = Low.

we additionally require the types of the receiver objects to be equal (see Figure 3.10).

`Low(exp)`

With this we express that an expression `exp` is low. Note that not only variables can be specified to be low, but any expression, e.g., whether a value is even.

The encoding is again as shown in Section 3.4 and Figure 3.10. This specification makes use of references to compare the values across executions, meaning that, e.g., the encoding of `Low(v)`, for some variable `v` in the source program, compares the references stored in the two versions $\mathtt{v}^{(1)}$ and $\mathtt{v}^{(2)}$.

`LowVal(exp)`

This is the same as `Low`, except that we compare the values of the expression, not the references. In Python the two comparison methods are "`==`" for values and "`is`" for references. The distinction is important when we compare expressions between different executions in the MPP.

The difference is illustrated in two examples in Figure 4.1. Method `bool_int` returns 1 or `True` depending on a high input `secret`. In Python, those expressions are considered to be equal values, but the references are not the same ("`1 == True`" is true, but "`1 is True`" is not). Since the two executions we model in the MPP might take different paths depending on secret data, we cannot prove that the result is low when we compare the references of the two results. Method `unchanged` shows a similar issue: it will always return an `int`, but if a secret value is zero we add it to the result. Obviously, the value will remain unchanged either way, and in Python even the references will be equal because of the way `int`s are represented ("`x + 0 is x`" is true in Python), but because of an incompleteness in Nagini's encoding we may be unable to prove this reference equality. Therefore, we cannot prove the postcondition in both examples when we use `Low` for the contract f_{low}.

To solve the problem, we introduce the `LowVal` contract function, which makes the comparison based on Python's `__eq__` method. This method is defined on all Python objects (including primitives) and is used for the value comparison ("==" is syntactic sugar for calling the `__eq__` method). This gives us the following encoding:

$$\lfloor \texttt{LowVal(exp)} \rfloor_2^{\mathring{act}} = \texttt{T\_\_\_eq\_\_}(act^{(1)}, act^{(2)}, \texttt{exp}^{(1)}, \texttt{exp}^{(2)}),$$

where `T` is the type of `exp` and `T___eq__` is the encoding of `T`'s `__eq__` method. Substituting `LowVal` for f_{low} in Figure 4.1 allows us to verify both examples successfully.

In theory we can apply this encoding for any type `T`, where the user can define a custom `__eq__` method by overriding the one of the `object` class. However, because of technical reasons `LowVal` does not yet support arbitrary objects. The types for which we can use a value comparison are: `int`, `bool`, `float`, `string`, as well as tuples, sequences and sets, which have a custom equality function defined by Nagini which we can use to make the comparison in the product. For all other objects `LowVal` uses the `__eq__` method of Python's `object` class which compares references, and thus makes `LowVal` equivalent to `Low` in these cases.

Note that it is still useful to have the standard version of `Low` for the supported types, because we might want to use reference comparison across executions.

4.1.2 Declassify Statement

`Declassify(exp)` provides the ability to declassify an expression `exp`, as defined in the MPP paper.

Declassifying an expression means to make it low, if it was high to begin with. This is useful as often a program should be allowed to leak some amount of secret data, for example, to tell a user if a password was correct (without revealing the correct password). Without declassifying the result of a password check, SIF would not allow us to reveal it, as it depends on secret data.

Again the encoding is very similar to the definition in the original paper:

$$\lfloor \texttt{Declassify(exp)} \rfloor_2^{\mathring{act}} = \textbf{assume}\ act^{(1)} \wedge act^{(2)} \Rightarrow \texttt{exp}^{(1)} = \texttt{exp}^{(2)}.$$

The declassification translates into an `assume` statement, stating that `exp` is equal in both executions. The assumption is conditional on both executions being active and all control flow variables being false.

4.1.3 Method Decorators

We believe that in the majority of code in real-world code bases the data that is worked with is not secret. Still, in order to verify secure information flow because some parts of the code are security critical, using the contract functions shown so far, one would have to annotate the whole code base, specifying all low data as such, so that the MPP can be built and verified. This would lead to a large amount of boilerplate annotations, saying that all arguments and results of a method are low. For convenience, we therefore provide two method decorators, which automatically add SIF specifications where no secret data is involved.

`@AllLow` is used to express that all inputs and outputs of the method are low, and it never operates with any secret data.

In particular, if A is the set of all arguments of an all low method, H is the set of all heap-dependent expressions to which the method gets access in the precondition and R is the set of results (return value and raised exceptions), we add the preconditions

$$\texttt{LowEvent()} \text{ and } \forall \texttt{e} \in A \cup H.\ \texttt{Low(e)}.$$

The reason we add the `LowEvent()` is to be able to, for example, call print or other methods which reveal information. Additionally we add the postcondition

$$\forall \texttt{e} \in R \cup H.\ \texttt{Low(e)}.$$

For each loop in the method body, we add the loop invariant

$$\forall \texttt{t} \in T.\ \texttt{Low(t)},$$

where T is the set of targets in the loop.

Finally, we have to treat all predicates that the decorated method has access to as if they specified all variables and accessible heap locations to be low. This is because, for example, if a method requires access to some predicate `P`, we also have to require that `P` does not give access to secret values. For this purpose we generate a function for each predicate `P`, called $\texttt{P}^{(allLow)}$, that contains the assertion that all accessible variables and heap locations are low, as well as requiring the $\texttt{Q}^{(allLow)}$ function to be true for all predicates `Q` to which `P` has access. We then generate

$$\forall \texttt{P} \in PA_{pre}.\texttt{p}^{(1)} \wedge \texttt{p}^{(2)} \Rightarrow \texttt{P}^{(allLow)} \text{ and}$$
$$\forall \texttt{P} \in PA_{post}.\texttt{p}^{(1)} \wedge \texttt{p}^{(2)} \Rightarrow \texttt{P}^{(allLow)}$$

as additional pre- and postconditions respectively, where PA_{pre} are all predicate accesses in the preconditions and PA_{post} the ones in the postconditions. In the decorated method, we then encode predicate accesses, fold and unfold statements the same as defined for relational predicates, except using $\mathtt{P}^{(allLow)}$ instead of $\mathtt{P}^{(rel)}$ (see Figure 3.14 and Figure 3.15).

`@PreservesLow` is used to express that the method preserves lowness. This decorator is very similar to the `@AllLow` decorator, except that it does not require all inputs to be low, it just promises that if the inputs are low, the results will be, too. This is useful, for example, for an increment method. If we put an `@AllLow` decorator on the increment method, it could not be used to increment a secret value, because it would require the input to be low. Since we want to be able to use the same method for secret and public inputs, we added the `@PreservesLow` decorator.

If we again assume that A is the set of arguments of a method which is decorated with `@PreservesLow`, H the set of heap-dependent expressions the method has access to and PA_{pre} the set of predicate accesses in the precondition, we can express that all accessible state is low as follows:

$$\begin{aligned} allStateLow =& (\forall \mathtt{e} \in A \cup H.\ \mathtt{Old(Low(e))}) \wedge \\ & (\forall \mathtt{P} \in PA_{pre}.\mathtt{p}^{(1)} \wedge \mathtt{p}^{(2)} \Rightarrow \mathtt{Old(P}^{(allLow)}\mathtt{))}. \end{aligned}$$

`Old(exp)` is a contract function indicating that exp should be evaluated in the state before the execution of the method. With R being the set of results of the method and PA_{post} the set of predicate accesses in the postcondition, we add the postcondition

$$\begin{aligned} & allStateLow \Rightarrow \\ & \quad (\forall \mathtt{e} \in R \cup H.\ \mathtt{Low(e)}) \wedge (\forall \mathtt{P} \in PA_{post}.\mathtt{p}^{(1)} \wedge \mathtt{p}^{(2)} \Rightarrow \mathtt{P}^{(allLow)}) \end{aligned}$$

to the decorated method. The loop invariant we add to each loop is

$$allStateLow \Rightarrow (\forall \mathtt{t} \in T.\ \mathtt{Low(t)}),$$

where again T is the set of loop targets.

Note that here we do not add `LowEvent()` to the preconditions, as we want to allow the methods to be called depending on secrets. This is another reason for the distinction between `@AllLow` and `@PreservesLow`.

Similar to the `@AllLow` decorator, we have to change the encoding of predicate access, fold and unfold such that they use the predicates *allLow*-function. Again we use similar encodings to the ones shown in Figure 3.14 and Figure 3.15, with $\mathtt{P}^{(allLow)}$ instead of $\mathtt{P}^{(rel)}$, but here we make an additional change. Namely the function has to be true not only on the condition that both executions are active at this point, but also only when all state was low when the method was called. This means we use $\bigwedge_{i=1}^{k}(act^{(i)}) \wedge allStateLow$ on the left hand side of the implications instead of just $\bigwedge_{i=1}^{k}(act^{(i)})$.

In both decorators we have the option to exempt certain variables from being added to the low variables. For example, obligations (see next section) add more arguments to methods in order to carry information which is only needed for the verification of obligations. For these variables we cannot prove that they are low, but it is not necessary as they can not affect secure information flow. Therefore, we can set up the generation of additional specifications to ignore these variables, and be able to verify decorated methods with obligations active.

The decorators `@AllLow` and `@PreservesLow` could be further optimized in a way that we do not have to create the MPP of decorated methods, which adds complexity and therefore takes longer to verify. We did not implement this optimization but describe how it can be done and leave it for future work.

The way we described the decorators we encode all methods as an MPP, so consequently we verify that the decorators are justified, as the additional specifications could not be proven to hold otherwise. Note that it is important to check that the decorators are justified, as we do not want to trust the user to exempt methods completely from the SIF verification. When we optimize the encoding such that the decorated methods are not encoded as MPPs but only as single executions for the functional verification, we need to add some additional checks that they can only call methods which themselves do not leak secrets and if they access any global variables assert that they are low.

As a consequence of only selectively encoding methods into MPPs, we have to create stubs for the decorated methods, such that those methods we did encode could still call those methods, because they expect another signature with activation variables and duplicated arguments. These stubs can be body-less methods with the encoded method signature and specification. There is no need to add stubs for the encoded methods, as the decorated methods are not allowed to call undecorated ones.

4.2 Obligations and Termination Channels

One way a program can leak secret information is via *termination channels*, which means that whether or not a program terminates depends on a secret. Figure 4.2 shows two examples of programs with a termination channel. On the left there is a loop which will never terminate if the (high) input

```python
def loop(h: int) -> None:
    while h != 0:
        h = h - 1
```

```python
def recursive(h: int) -> None:
    if h == 0:
        return
    recursive(h - 1)
```

Figure 4.2: Examples of termination channels: A loop which terminates iff h ≥ 0 (left), a recursive function which terminates iff h ≥ 0 (right). In both examples h is high.

is negative, on the right a negative input will cause an infinite recursion. If an observer sees that the program does not terminate, they can learn information about the secret.

The MPP paper defines a method to verify absence of termination channels in loops. We will use the same approach and apply it to both the infinite loop and recursion scenario. We rely on the additional checks defined in the MPP paper to verify the relational aspects of termination channels, and for verification of termination itself we make use of obligations, which we briefly summarize in Section 4.2.1. In Section 4.2.2 we combine obligations with the termination channel verification.

4.2.1 Obligations

Obligations [8] are constructs which allow for verifying that a program eventually performs some action. In this section we present them as they are implemented in Nagini [1], since that is what we use in our own implementation. Obligations can be used to prove a variety of properties; for our purpose we will focus on the part which allows for proving termination, and we only provide a minimal explanation required to understand our steps to incorporate them in the MPP encoding.

To express that a method or loop terminates, Nagini provides a contract function `MustTerminate(measure)`. The `measure` argument is an integer expression describing an upper bound of the number of steps the method or loop is allowed to make before terminating. Steps means the maximum height of the call stack in the case of methods, or the number of iterations in a loop.

Figure 4.3 shows an example program with an obligation to terminate. In the precondition the method is required to terminate with `measure = 1`, which means that the method is not allowed to call any other methods. Because we want to show that the method terminates, we must give the loop a specification which guarantees that the loop will terminate. In the loop invariant we therefore specify that it terminates in `5 - i` steps. The measure in the loop invariant must decrease after each iteration, and never become negative.

```
def main() -> None:
    Requires(MustTerminate(1))
    i = 0
    while i < 5:
        Invariant(MustTerminate(5 - i))
        i += 1
```

Figure 4.3: Example program with obligation to terminate (Nagini).

The way obligations are expressed is via permissions to predicates, for example, having access permission to a `MustTerminate` predicate means there is an obligation to terminate. In the Viper encoding of the example this means a precondition is added, requiring access to said predicate. To prove that the method terminates an assertion is generated, which is conditional on an expression called the *guard*, and uses the *measure* integer expression provided by the user. The assertion checks that the measure is always non-negative and decreases over time. Besides this assertion there is what is called a *leak check*, which checks that no obligations are dropped without being fulfilled. This is done by asserting that the method has no remaining access permissions to any of the obligation predicates.

There are two places an obligation can be specified:

(a) In a method precondition. In this case the assertion that the measure decreases is placed in the precondition and uses an additional argument, called the `_caller_measures`, of the method representing measures which have to be passed by the caller. This ensures that if a method has an obligation to terminate, calling other methods is only allowed if the called method promises to terminate in fewer steps, and the callee will take over the obligation.

    In our example this means that in the precondition it is checked that a caller has no obligation to terminate in one step or less, because the `main` method could not fulfill that obligation. The guard in this case is `True`, as the obligation is not conditional.

    The leak check is placed in the postcondition.

(b) In a loop invariant. Here the assertion that the measure decreases is placed at the end of the loop body. In the example this means that at the beginning of the loop body the measure `5 - i` is stored, and at the end of the body it is asserted that `5 - i` evaluates to a smaller number than what was stored at the beginning.

    The leak check is placed in the invariant, inside `InhaleExhale` expressions, such that they only get exhaled but never inhaled.

Since the obligation implementation makes use of `InhaleExhale` expressions in loop invariants, we need to make an adjustment to our loop encoding to be able to verify obligations. The issue is that when the exhaled part in a loop invariant is stronger than the inhaled part (which is the case here), it is possible that the invariant inhaled at the beginning of an iteration does not imply the invariant is preserved even with an empty loop body. Unfortunately, this is exactly the scenario we have in the MPP in the case that one execution exits the loop earlier than the other or if only one execution skips the loop entirely. Recall that in our MPP encoding a loop in the source maps to one loop in the product, which is executed as long as either of the executions is active, and the inactive one keeps on iterating without doing anything in the body. It follows that we cannot prove that the loop invariant is preserved.

The adjustment we made to solve this problem is to introduce new Boolean variables which we call `idle`, to represent that an execution is iterating but only because of the other execution, and is therefore not making any changes. We set both `idle` variables to false before the loop, and at the beginning of the loop we assign $\texttt{idle}^{(i)}{:=}\texttt{p}^{(i)} \wedge \neg\texttt{ret}^{(i)} \wedge \neg\texttt{break}^{(i)} \wedge \neg\texttt{cont}^{(i)} \wedge \neg\texttt{except}^{(i)} \wedge \neg\texttt{c}^{(i)}$. Now we can define the translation of `InhaleExhale` as

$$\lfloor [in, ex] \rfloor_k^{\mathring{act}} = [\lfloor in \rfloor_k^{\mathring{act}}, \lfloor \neg\texttt{idle} \Rightarrow ex \rfloor_k^{\mathring{act}}].$$

This translation ensures that we only need to show the exhale part of the invariant at the end of an iteration for executions which are not idling.

We can now encode obligations into the MPP as expected and use them for proving absence of termination channels.

4.2.2 Termination Channels

To prove the absence of termination channels we require a specification in the loop invariant or the method precondition, which tells us under which condition the loop or method will terminate, called the *termination condition* or e_c. Additionally we require a *ranking function* e_r, an integer expression with which we can express the obligation that after every loop iteration or call to a method, e_r must decrease but never become negative. To specify this we introduce the contract function `TerminatesSif(cond: bool, rank: int)`, with inputs `cond` for e_c and `rank` for e_r. The contract can appear in method preconditions, which allows for verification of the scenario with infinite recursion, or as a loop invariant, for the infinite loop case.

In both cases we create an obligation with guard e_c, to prove that if e_c evaluates to true the method or loop really terminates. Other than this we need three more assertions, namely:

1. $\texttt{low}(e_c)$. This ensures that whether or not the method terminates does not depend on secrets.

```
 0 method recursive(p1: Bool, p2: Bool,
 1                  h1: Int, h2: Int, /*obligation args*/)
 2   requires p1 && p2 ==> (h1 >= 0 == h2 >= 0) // assertion 1
 3   requires !(h1 >= 0) && !(h2 >= 0) ==>
 4      p1 == p2                                // assertion 2
 5   requires /* obligation measure check */
 6   ensures  /* obligation leak check */
 7   ensures old(p1 ==> h1 >= 0)
 8      && old(p2 ==> h2 >= 0)                  // assertion 3
 9 {
10   /* conditional return */
11   /* call recursive with new _caller_measures */
12 }
```

Figure 4.4: Encoding of `recursive` example from Figure 4.2, with the precondition `TerminatesSif(h >= 0, h + 1)` (Viper, simplified).

2. $\neg e_c \Rightarrow$ `lowEvent`. This assertion is to prove that whether or not an execution calls a method or reaches a loop which does not terminate, does not depend on high data.

3. A check ensuring that e_c is exact, meaning that $\neg e_c$ implies that the method or loop does not terminate.

Where we place these assertions differs between methods and loops, we will show each case on the examples from Figure 4.2.

Methods

When the `TerminatesSif` contract appears in a method precondition in the source program, we add the assertions 1 and 2 to the precondition of the product. The third assertion is checked in an additional postcondition in the product, expressing `old(`e_c`)`. This ensures that the end of the method is only reached if the termination condition, evaluated in the state before the method was executed, was true.

Figure 4.4 shows the simplified Viper encoding of the `recursive` method from Figure 4.2, which results when we add the precondition

```
Requires(TerminatesSif(h >= 0, h + 1)).
```

The measure needs to be `h + 1`, because it has to be positive even when we have `h == 0`. In lines 2–4 of the encoding we have the additional preconditions, the first one expressing that `h` is non-negative in either both or none of the executions, the second one that if `h` is negative then both executions must make the call and, therefore, not terminate. Note that it is not possible that only one execution has a negative `h` because of assertion 1. Lines 4 and

```
 0 method loop(p1: Bool, p2: Bool,
 1             h1: Int, h2: Int, /*obligation args*/)
 2 {
 3    assert p1 && p2 ==> (h1 >= 0 == h2 >= 0)    // assertion 1
 4    assert (!h1 >= 0 && !h2 >= 0) ==> p1 == p2 // assertion 2
 5    cond1 := h1 >= 0
 6    cond2 := h2 >= 0
 7    while(p1 && h1 != 0 || p2 && h2 != 0)
 8       invariant /*obligation leak check*/
 9       invariant p1 && !cond1 ==> h1 != 0      // assertion 3
10       invariant p2 && !cond2 ==> h2 != 0      // assertion 3
11    {
12       p1' := p1 && h1 != 0
13       p2' := p2 && h2 != 0
14       /* store obligation measures */
15       if (p1') {h1 := h1 - 1}
16       if (p2') {h2 := h2 - 1}
17       /* check obligation measures decreased */
18    }
19 }
```

Figure 4.5: MPP encoding of loop example from Figure 4.2, with loop invariant `TerminatesSif(h >= 0, h)` (Viper, simplified).

5 are pre- and postconditions added by the obligation. Line 6 is assertion 3, ensuring that any trace where h was originally negative does not reach the end, as for those traces we cannot prove the assertion. In the body there are no additional assertions.

With these specifications we can prove that the method terminates iff `h >= 0`. We can now guarantee that there is no termination channel, as callers have to prove that the termination condition is low and that both executions make the call if it does not hold.

Loops

When the contract is in a loop invariant, we add assertions 1 and 2 before the loop and ensure assertion 3 by showing that $\neg e_c$ implies the loop condition is always true and thus the loop does not terminate.

Figure 4.5 shows this on the `loop` example from Figure 4.2. Lines 3 and 4 contain assertions 1 and 2, in line 5 and 6 we store the value of the termination condition, because the value might change in the loop and we need the value from before the loop to check assertion 3. This happens in lines 9–10, ensuring that if e_c did not hold before the loop, the loop condition can never be false and thus the loop will not terminate.

```python
def m(val: int, wait: bool) -> None:
   if wait:
      i = 10_000
      while i > 0:
         i = i - 1
   print(val)

def main(secret: bool) -> None:
   t1 = Thread(target=m, args=(1, secret))
   t2 = Thread(target=m, args=(2, !secret))
   t1.start(m)
   t2.start(m)
```

Figure 4.6: Example satisfying possibilistic (but not probabilistic) noninterference (Python).

All these additions allow us to detect the termination channels in the examples from Figure 4.2.

4.3 Possibilistic Noninterference

Leino and Müller [16] propose a methodology for verification of advanced concurrency patterns. This methodology is implemented in Nagini, and in this section we will present how we can extend it to guarantee possibilistic noninterference in a concurrent setting, in particular we look at locks and forking and joining threads.

Possibilistic noninterference is a property which states that given two runs of a program with the same public inputs (and possibly different secret inputs), it is possible that the observed outputs are the same. In other words for any observed $\texttt{result}^{(1)}$ of execution 1, there exists a schedule for execution 2 such that $\texttt{result}^{(1)} = \texttt{result}^{(2)}$. In verifying possibilistic noninterference we assume that the scheduler is non-deterministic and unrestricted (i.e., it can choose to switch to any thread at any time). Figure 4.6 shows an example to illustrate this: Method m takes an integer argument `val` and a Boolean `wait`. If `wait` is true it executes a large amount of iterations and then prints `val`, otherwise it prints `val` immediately. In method `main` we `fork` two threads, both of which execute `m`, one with `val == 1` and the other with `val == 2` and the value of `wait` will be true in one thread and false in the other, but which is which depends on a Boolean `secret`. Now obviously when someone observes the output to be "12" or "21" they can be rather confident that they know the value of `secret`, because the thread where `wait` was true is much more likely to reach the print statement later than the other one. In the setting of possibilistic noninterference the example does not violate secure information flow however, since it is still possible that the scheduler has allowed all the iterations to happen before switching to

```python
class Cell:
   def __init__(self, val: int) -> None:
      self.value = val
      Ensures(Acc(self.value) and self.value == val)

class CellLock(Lock[Cell]):
   @Predicate
   def invariant(self) -> bool:
      return Acc(self.locked().value) and
             Low(self.locked().value)

def main(secret: bool) -> None:
   c = Cell(1)
   l = CellLock(c)
   l.acquire()
   c.value = 4
   if secret:
      l.release() # has to fail
      l.acquire()
   c.value = 5
   l.release()
```

Figure 4.7: Example program leaking a secret via locking (Nagini, simplified). If an observer ever sees the value 4 in the `Cell` object they know the value of `secret`.

the other thread, so the observer has no guarantee that their conclusion is correct.

An alternative property, which the example in Figure 4.6 does not have, is *probabilistic noninterference*, where one verifies that the probabilities of different outputs are the same given that all public inputs are equal. In order to verify this property one would have to make stronger assumptions about the scheduler and model the runtime, e.g. via step counting, which is why we focus on possibilistic noninterference.

4.3.1 Locks

In the methodology we consider, a lock is associated with a *lock invariant*. Without considering secure information flow, we encode acquiring a lock as inhaling the lock invariant, and releasing corresponds to exhaling the lock invariant. We want to support relational lock invariants, such that we can express, e.g., a lock invariant that gives access to some field and specifies this field to be low. The inhaling and exhaling of the lock invariant alone is not sound with respect to SIF, as we illustrate in Figure 4.7.

In the example we have a `Cell` object with a `value` field, and a `CellLock` object locking a `Cell` object. The lock invariant gives access to the field and specifies it to be low. In the main method we create a Cell and a lock and we acquire it. We set the value to 4, then depending on a secret we release the lock and acquire it again, to then set the value to 5 and finally release it. Now if an observer ever sees the value 4 in the cell they know that secret must be true, because otherwise no thread can ever have access before the value is set to 5.

To make the encoding sound we will ensure that whether or not a thread holds a lock does not depend on a secret. We can achieve this by adding two preconditions, lowEvent and low(self), to both the acquire and release methods. The first one guarantees that the action happens in both executions, which is enough to get the expected result in the example: the call to release in line 18 will fail, because it is not a lowEvent. The second precondition requires that the lock on which the acquiring or releasing happens is the same in both executions. This is necessary for situations where we have multiple locks and which one we lock or unlock depends on secret information. For example, when we have two locked Cells holding different values, releasing only one lock could reveal the secret.

4.3.2 fork/join

We consider a methodology in which forking a thread corresponds to exhaling the preconditions of the method the thread executes, whereas joining a thread means inhaling the postconditions of the method.

For fork we do not need any additional specifications, as everything the thread could require (e.g. lowEvent) must be in the precondition of the method the thread executes. However, we must consider that the method which a thread executes is not static and thus can depend on secrets. Consequently, we need to differentiate between executions when looking up which method's specifications to in- or exhale. In Nagini's existing encoding this lookup happens via a lookup function. To ensure this differentiation, we use two different versions of this function in the Viper encoding, even though it is not heap-dependent and thus we would not do this for the reasons discussed in Section 3.6.1.

Figure 4.8 shows an example where the method the thread executes depends on a secret, and since the methods print to the public output they require lowEvent. Note that in Nagini the start method takes as arguments a list of methods which could be executed (which one it actually is is determined in the creation of the Thread object). Obviously the example has to fail, as the output will immediately tell an observer the value of secret. The way the fork is currently encoded by Nagini (before the MPP encoding) is with one conditional statement per method the thread could run, e.g., **if** (getMethod(t) == print1) **then** {...} would be the first conditional, and in the then block of this conditional the corresponding method

```python
def print1() -> None:
    Requires(LowEvent())
    print(1)

def print2() -> None:
    Requires(LowEvent())
    print(2)

def main(secret: bool) -> None:
    if secret:
        t = Thread(print1)
    else:
        t = Thread(print2)
    t.start(print1, print2) # fork: has to fail
```

Figure 4.8: Example of a program where the method a thread executes depends on a secret and can not be forked safely (Nagini).

precondition is exhaled, in this case `print1`. When we create the MPP of this encoding this will create a new version of the activation variables for each of these `if` blocks, and since we do not know that both executions enter the same branch (the method depends on a secret and we duplicated the `getMethod` function), the exhale of the precondition `lowEvent` will fail, which is the behavior we expect.

The same argument holds analogously for `join`, except that here we inhale the postconditions instead of exhaling preconditions. The inhaling happens in the same way as the exhaling, namely in a series of conditionals where the thread's method is compared with each possibility. In the example in Figure 4.9, where we have two methods that are both promising that the value they assign to a field is low, but depending on a secret the thread's method will be different, we expect not to inhale that the value is low. This is actually the case, since the postconditions of `one` and `two` will be inhaled using the new activation variables, so what we inhale is that given both executions take the same branch, `c.val` is low. This means that, as expected, we cannot prove the assertion in line 19.

Therefore, the `fork/join` scenario does not require any additional checks in order to guarantee possibilistic noninterference.

```python
def one(c: Cell) -> None:
    Requires(Acc(c.val))
    Ensures(Low(c.val))
    c.val = 1

def two(c: Cell) -> None:
    Requires(Acc(c.val))
    Ensures(Low(c.val))
    c.val = 2

def main(secret: bool) -> None:
    c = Cell()
    if secret:
        t = Thread(one, args=(c,))
    else:
        t = Thread(two, args=(c,))
    t.start(one, two) # fork
    t.join(one, two)
    Assert(Low(c.val)) # has to fail
```

Figure 4.9: Example of a program where the method a thread executes depends on a secret (Nagini, simplified).

Chapter 5

Implementation

This chapter concerns the implementation of the MPP extensions and SIF specifications discussed in Chapters 3 and 4. We first give an overview of existing infrastructure that we used, then we discuss how we designed the implementation of the extended MPP. Finally, we discuss some optimizations we made to the encoding to improve verification performance.

5.1 Existing Infrastructure

The infrastructure we based our implementation on consists mainly of two parts. On one hand this is the existing implementation of the MPP transformation [12], which encodes Viper programs to Viper MPPs. On the other hand there is Nagini, which encodes Python programs into Viper.

As mentioned in Chapter 2, Viper defines an intermediate language and provides two backends, one based on verification condition generation and one based on symbolic execution. Viper is implemented in Scala, and it defines an abstract syntax tree (AST) for all the supported statements and expressions. The existing MPP transformation, also implemented in Scala, takes such an AST as input and transforms it into another Viper AST. The backends are also Scala programs, and they work directly with a Viper AST.

Nagini is implemented in Python, it takes a Python program file as input, parses and type checks it using *mypy*[1], and transforms it into a Viper AST. The creation of the Viper AST happens with the help of *jpype*[2], a library that allows Python programs to interact with a JVM, which is what Scala, and therefore Viper, runs on. This allows it to directly create Viper AST nodes as JVM objects. It then invokes the backends via jpype with the generated Viper AST as argument.

[1] `http://mypy-lang.org`
[2] `http://jpype.sourceforge.net`

5.2 Design

An important aim in designing the implementation was to be able to reuse as much of the existing infrastructure as possible. Another goal was to make the extended MPP transformation as general as possible, such that it could be reused by a frontend for another object-oriented language. If we did the MPP transformation directly in Nagini on the Python level, one would have to implement it again to do SIF verification of, e.g., Java programs. Our design decision is therefore to keep the extended MPP transformation on the level of the Viper AST, which makes it reusable for other frontends and it can be based on the original MPP implementation. It also allows us to reuse most of Nagini's Python to Viper encoding, which is quite involved and would require a lot of effort to rebuild for SIF.

The Viper language does not support some common language features that we want to support in Python programs. For example, `return` statements do not exist in Viper, which is why Nagini translates them into `goto`s. Since `return` requires a special encoding in the extended MPP, we want to have a special representation for it in the AST. For that reason we decided to extend the Viper AST.

The Viper AST is not built to be extensible and we do not want to add nodes directly in the Viper core, as our new nodes are not meant to extend the Viper language, but only as a way to represent an intermediate AST. Furthermore, if we added nodes directly to the AST, the Viper backends would have to be adapted to be able to cope with the new nodes, or at least to throw a meaningful error when they encounter one. This would have to be done every time someone wants to extend the AST, making this approach non-modular. What we do instead is to create two new Scala traits that extend the traits for statements and expressions, respectively. These new traits define an interface which all our extension nodes need to have, for example, they need to define a `subnodes` method, which can be used in the AST's infrastructure. This way we only have to adapt the infrastructure once, to allow it to handle the new traits. We can then use the same infrastructure of the original AST to, e.g., traverse or transform an AST containing new nodes. Since the AST which is given to the backends for verification must not contain any of the new nodes, the backends can remain unchanged and oblivious to the new AST nodes.

The MPP transformation rewrites all extension nodes to standard Viper AST nodes as described in Chapter 3. Here is the list of the AST statement nodes we add:

`Return(res, resVar)` `res` is the expression which is returned and `resVar` the result variable, such that we can assign the former to the latter in the encoding. We add the result variable as a field in the AST node, because a Viper method can have several return variables and we need the information from Nagini which one to assign the result to.

`Break(), Continue()` These require no subnodes, as we only need to know the location of these statements.

`Raise(assignment)` `assignment` is the assignment statement of the thrown error to the error variable.

`ExceptionHandler(errVar, exception, body)` This represents a catch block, with `errVar` being the error variable to be compared to `exception` in order to generate the condition under which the `body` of this handler should be executed.

`TryCatch(body, catchBlocks, elseBlock, finallyBlock)` This represent a try/catch statement. `body` represents the body of the try block, `catchBlocks` is a sequence of `ExceptionHandlers`. Both the `elseBlock` and `finallyBlock` are optional sequential compositions.

`Declassify(exp)` `exp` represents the expression to be declassified.

`InlinedCall(body)` This is a wrapper around an inlined method call. It is necessary because the inlined method needs its own set of control flow flags. If, for example, the method body which is inlined contains a return statement, we do not want to set the `ret` flag of the caller method. With this wrapper we can create new control flow variables to be used in the encoding of the inlined method's `body`.

The following list describes the expression nodes we add:

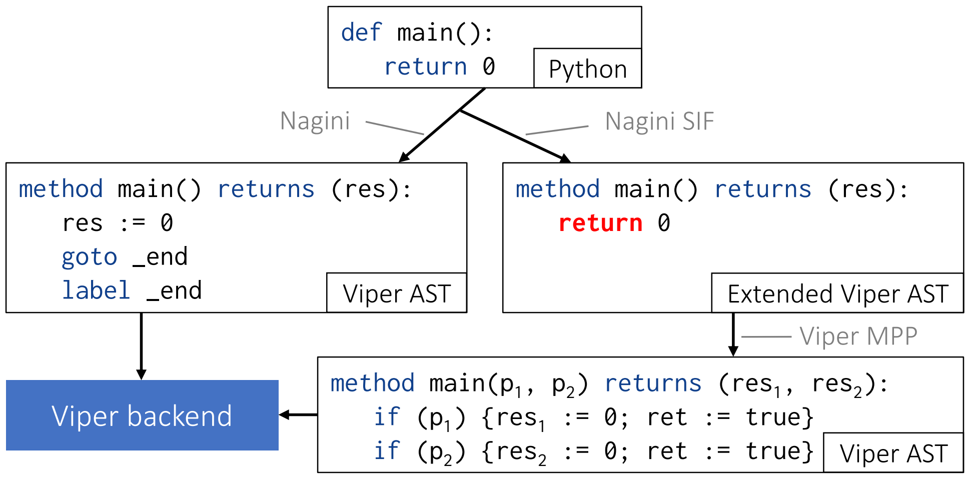
`Low(exp, cmp)` `exp` represents the expression which is specified to be low, and `cmp` is optional for a comparator to represent `LowVal`.

`LowEvent()` This represents the `lowEvent` assertion.

`Terminates(cond)` `cond` represents the termination condition. We do not require the termination measure, as that is handled by the obligation, but we need the condition to generate the additional assertions as described in Section 4.2.2.

With these new nodes we can implement the MPP transformation as described in Chapter 3, such that the resulting Viper program consists entirely of standard Viper AST nodes and can be verified using the backends. We can do this reusing large parts of the original MPP transformation, where we add support for the new nodes and adapt others to do the changed encoding. Our implementation is restricted to generating MPPs with $k = 2$.

To create the extended Viper AST from a Python program, we extend Nagini. Nagini translates Python programs in two phases: in the analyzing phase it collects information needed for the creation of the Viper AST nodes, which happens in the second phase. Nagini defines translator classes which handle different types of Python AST nodes, to traverse the Python AST and generate a Viper AST.



```
def main():
    return 0
```

```
method main() returns (res):
    res := 0
    goto _end
    label _end
```

```
method main() returns (res):
    return 0
```

```
method main(p1, p2) returns (res1, res2):
    if (p1) {res1 := 0; ret := true}
    if (p2) {res2 := 0; ret := true}
```

Figure 5.1: Design of the implementation on an example program.

We extend these translators in subclasses, such that we can largely use the same code but override the translation of some parts to create an extended Viper AST. For example, in the statement translator there is a method which translates a `return` statement into a `goto`. We override this in the new SIF statement translator to create a new Viper `return` node.

The design of the implementation is illustrated in Figure 5.1. It shows a small example program which returns 0 and how Nagini encodes it in a Viper AST that is sent to a Viper backend for verification. On the right side is shown what we do when running Nagini with the SIF option. The first step in the translation reuses much of the standard Nagini encoding, e.g., the creation of the method node with the arguments and result variables, but the return node is transformed into an extended Viper AST node (highlighted red in the figure). To get a Viper AST which we can hand to the backend, we do the MPP transformation which is entirely on the Viper level. The MPP is then verified using the unmodified backends.

5.3 Optimizations

In this section we discuss four optimizations we made in the implementation to improve verification performance. For the evaluation of their effectiveness we refer to Chapter 6.

5.3.1 Control Flow Optimizations

Many methods do not contain `return`, `break`, `continue` and `raise` statements all at once. It is therefore unnecessary to always create all the control variables and conjoin the activation variables with four others. Each variable we have makes the formula we hand to the underlying solver more complex, which is not only bad for performance, but makes the code much harder to read when looking at the MPP, e.g., for debugging purposes.

With this optimization, when encoding a method we first traverse the method's AST to find out which language features there are and only create the control variables which are needed for the method. In the encoding this means that we conjoin the activation variables with all those control flow flags which exist in the method.

The other control flow construct we do not have to generate every time is the loop reconstruction. The reconstruction is there to gather information in cases of control flow in the loop which involves the control flow flags. Thus, when a loop never modifies any of those flags we can omit it entirely.

5.3.2 Sequential Composition

When we encode, for example, the sequential composition of two assignment statements, what we get according to the MPP definition (for $k = 2$) are four consecutive conditionals, one for the first assignment in the first execution, then one for the first assignment in the second execution and so on. We could get the same result with just two conditionals, one per execution, with both assignments happening in the same conditional block.

This simplification is sound, because between two assignments the two executions can not influence each other, i.e., the assignment does not have any effect on the other execution. Therefore, it does not matter when we change the order such that we put the assignments of one execution first and the ones from the second execution after. In contrast, for example, a method call which takes two activation variables as arguments and can include the exhaling and inhaling of relational assertions does influence both executions, so the executions cross paths. This means that a call creates a barrier across which we must not reorder statements of different executions, i.e., everything which happens before the call in the source program, must happen before the call in the product for all executions.

In general, this optimization is looking for sequential compositions of statements which can be executed in sequence without affecting any relational aspects of the program, and compress them to use a single conditional, instead of creating one conditional for each statement. In particular the statements we can compress are assignments, `return`, `break` and `continue` statements as well as inhaling and exhaling unary assertions. For some of those statements, namely the ones which set control flow flags, we have to be aware that they introduce a barrier between themselves and the next statement. This means that, for example, we can optimize a return statement to be put into a conditional with previous statements, but any subsequent statement has to be after a new barrier, since the setting of the `ret` flag might have changed the condition under which that statement gets executed.

This optimization is not only great for readability of the product, it also decreases the number of paths a trace can take through the program. This is especially important for verifiers based on symbolic execution, as they pay a cost in verification time for every path through a program.

5.3.3 Activation Variables

In this optimization we make use of the following observation: Program traces in which both activation variables are false never execute anything and all assertions are trivially true. Thus, it is unnecessary to include those traces in our verification. We can avoid some verification effort by assuming at the very beginning of each method body that at least one of the activation variable is true.

5.3.4 Avoiding Duplicate Checks

Python does not require from the user to declare all the variables used, and the mypy type checker does not enforce variables to be defined before being accessed. This is why it is possible to write a program that type checks, but at runtime tries to read an undefined variable (e.g., only define a variable in the `then` part of an `if` statement and read it afterwards). To ensure that this is never the case, Nagini introduces two pure functions, `isDefined` and `checkDefined`. Wherever a variable is assigned to, Nagini generates an **`inhale`** `isDefined(id)` statement, where `id` is an identifier which is created uniquely for each variable. Then, wherever a variable `v` is accessed, `v` is replaced with `checkDefined(v, id)`, to ensure that the variable has been defined previously. `checkDefined` returns the first argument, so it does not affect the semantics of the expression in any way, but in its precondition it requires `isDefined(id)`.

These checks add some complexity to all assignments and variable reads, and we can convince ourselves that it is enough to have them in one execution only, as the possible paths through the program are the same for both, so if we have a potentially undefined variable access, we can find it in either execution. This optimization allows us to avoid needlessly duplicating these checks. The extended MPP transformation can be configured with a list of function calls and an alternative, such that when we encode a function call, in the first execution it is kept as is, but in the second execution we replace the call according to the selected alternative.

In the case of `isDefined` we replace it by `true`, the calls to `checkDefined` get replaced by their first argument.

Chapter 6

Evaluation

In this chapter we evaluate our implementation with regard to expressiveness and completeness, as well as performance. First, to evaluate the SIF verification we encode examples from the literature in Python and use them to test our implementation. Next we look at the subset of Python programs we can encode and verify. Then we evaluate the performance impact the MPP transformation has on verification. To do this we make use of Nagini's test suite, which we run without creating the MPP and compare the runtime to the one we get when verifying the MPP. We do this using different combinations of the optimizations on the MPP transformation discussed in Section 5.3 to assess their effect. All tests were run on a Lenovo Yoga 910-13IKB laptop with an Intel i7-7500U dual core CPU and 16 GB RAM, running Windows 10.

6.1 SIF Verification

To evaluate the expressiveness of the SIF verification, we encoded a number of examples from the literature in Python and run the verification. The examples are mostly the same as used to evaluate the original implementation [12], so we can also compare the performance to the results observed there. The examples we did not encode are the three that verify the absence of timing channels, which our tool does not support, as well as one which we cannot encode effectively in Nagini. This is because of technical reasons and some incompleteness in the verifier itself, which are unrelated to this thesis. We ran the SIF verification in Nagini with all optimizations discussed in Section 5.3 enabled and measured the performance. All the times are averaged over ten runs.

In Table 6.1 we list the translation and verification runtimes for both Viper backends. The translation times T_{trans} include both the transformation to an extended Viper AST and the transformation to the MPP. Here the

File	$T_{trans}[s]$	$T_{VCG}[s]$	$T_{SE}[s]$	$\frac{T_{VCG}}{T'_{VCG}}$	$\frac{T_{SE}}{T'_{SE}}$
banerjee [2]	1.06	8.05	5.68	1.07	0.98
constanzo [9]	0.87	5.35	5.84	1.06	1.35
darvas [10]	0.82	5.65	2.13	0.91	0.96
example [12]	0.74	6.00	8.02	1.10	1.96
example-decl [12]	0.92	7.18	6.22	1.09	0.98
example-term [12]	0.70	4.96	1.04	0.97	0.95
joana-1-tl [13]	0.71	4.85	1.28	0.89	0.69
joana-2-bl [13]	0.95	5.02	1.30	0.95	1.08
joana-2-t [13]	0.67	5.14	1.24	0.99	0.91
joana-3-bl [13]	0.82	5.92	2.48	0.92	1.78
joana-3-br [13]	0.90	6.24	3.06	1.12	2.13
joana-3-tl [13]	0.81	5.40	2.03	0.90	1.03
joana-3-tr [13]	1.22	5.83	2.67	0.91	1.73
joana-13-l [13]	0.68	5.02	1.30	0.97	0.97
kusters [14]	0.75	5.81	1.75	1.00	0.97
product [3]	0.87	29.68	37.40	0.81	0.87
smith [20]	0.84	7.59	12.44	1.13	1.08
terauchi1 [22]	1.22	5.00	1.12	1.04	0.94
terauchi3 [22]	0.69	5.09	1.31	0.94	0.98
terauchi4 [22]	0.69	5.27	2.02	0.99	0.81

Table 6.1: Evaluation of example programs: T_{trans} is the time for translation from Python to Viper including the MPP transformation, T_{VCG} shows the verification runtime for the backend based on verification condition generation, T_{SE} for the one based on symbolic execution. All times are given in seconds. T'_{VCG} is the verification time with all SIF specifications disabled for the VCG backend, T'_{SE} the same for the SE backend. We give the slowdown factors of the SIF verification compared to the same examples encoded as MPP without the SIF specifications.

largest part of the runtime is due to Nagini's encoding from Python to Viper, presumably because Nagini makes a large number of calls to the JVM, which are not very efficient. The MPP transformation on the other hand is only a single call to the JVM, and in all examples it contributes less than one tenth of a second. In most examples the complete translation takes less than one second, with a few exceptions slightly over one second. This means that the MPP transformation process does not have a big impact on the runtime, but we can already see that the translation often takes longer than the complete verification time in the original implementation, which were in many cases under a second.

The runtimes are given as T_{VCG} for the backend based on verification condition generation (VCG) and T_{SE} for the one based on symbolic execution (SE) respectively. We also measured the runtimes of the same examples with relational specifications disabled, to examine their performance impact. These times are denoted as T'_{VCG} and T'_{SE} respectively, we only give the slowdown factors for both executions. A factor greater than one means the addition of relational specifications slowed the verification down, whereas one below one means better performance with relational specifications.

We can see that the performance impact of relational specifications is fairly small in general, with most slowdown factors being close to one. In many instances they sped up the verification, presumably because the `lowEvent` precondition decreases the number of possible states to consider. The two examples with the highest slowdown are “example” and “joana-3-br”, which take about twice as long with SIF specifications. In the former there is a `low()` expression in a quantifier, which means the quantifier is trivial where we do not encode the expression, which we believe leads to the big difference. The latter example consists of a series of body-less loops annotated with termination conditions. Again this makes verification trivial without the SIF specifications, as there are no unary ones.

The runtimes of the verification show similar patterns as observed in in the MPP paper, albeit on a different scale. In general it is much slower than the original MPP encoding, which is to be expected, as Nagini generates a large amount of overhead in the translation from Python to Viper, which is necessary since Python is a much larger language than that supported by the original implementation.

We can see that most examples still verify in under 8 seconds, the exception being an example using unbounded heap data structures (lists) that verifies in 30 seconds and more. Between the backends there is no clear advantage for either one, on average the symbolic execution backend seems to be faster, but there are examples where verification condition generation verifies more quickly. VCG seems to be more consistent, with most examples taking around 5 seconds. We think the reason for VCG often being slower is that it has a larger constant overhead than the SE backend. The VCG backend encodes the Viper program into Boogie, and for each example it starts a .NET runtime. However, once this overhead is paid, it seems to scale better than the SE backend, which creates different path conditions working directly on the Viper AST and invoking the SMT solver for each one.

6.2 Encoding Python Programs

In this section we discuss the range of programs we can verify with our implementation and its limitations. For this purpose we made use of Nagini's test suites of which there currently are four: functional, I/O, obligations and our own SIF suite we used to test the implementation. Our implementation can encode all those tests into MPPs, and, except for three test cases, yield the same verification results.

The three exceptions are from the functional test suite, and we could not verify them in reasonable time. This means that we stopped them after running for longer than 30 minutes, without getting a result. They are examples that use a lot of quantifiers, which, we believe, caused *matching loops*. In Viper, quantifiers are associated with *trigger expressions*, such that when an expression in the program matches a trigger, the quantifier is instantiated and the corresponding expression is added to the verification state. A matching loop occurs when the triggering of one quantifier adds an expression which in turn triggers another quantifier instantiation and so on. We believe that the three examples in question already displayed this issue, but SMT solvers can often get around this by applying heuristics and instantiating the quantifiers which lead to a solution quickly. We think the MPP encoding has amplified the problem, because we duplicate expressions in the MPP, such that it is possible that expressions from both executions trigger the same quantifiers, instead of treating them separately per execution. This increases the chances of obtaining matching loops. We cannot, however, treat quantifiers separately in general, as they could contain relational expressions.

The consequence is that in MPPs it is even more important to choose triggers carefully. In earlier test runs we observed average slowdown factors over one hundred, with some extreme cases being several thousand times slower to verify as MPP than as a standard encoding. The much better results we present in this section we observed after Nagini's encoding of Python constructs was updated to use more efficient triggers. Note that many of the examples we can verify in reasonable time do contain quantifiers (often many, as Nagini uses them in encoding Python's built-in types), it is only these few cases where the problem is so severe.

In summary, we support the same Python subset as Nagini does, with the exceptions mentioned above, where we do not know the verification result.

6.3 MPP Verification Performance

To analyze the performance of our implementation and MPPs in general, we compare the verification times of MPPs with the functional verification times of the same programs. For this we need to encode programs with only unary functional specifications into MPPs, such that we can verify the same programs without encoding them as MPPs. The examples we use for this we take from Nagini's functional test set, which consists of 96 Python programs. As mentioned in the previous section we cannot efficiently verify three tests as MPPs, which is why the following analysis is based on 93 test cases.

	T_{avg} [s]	Avg. slowdown	Median slowdown	Control Flow	Sequential	Activation Variables	Duplicate Checks
VCG	8.96	1.54	1.26	×	×	×	×
SE	15.79	6.26	3.51				
VCG	8.27	1.44	1.20	✓	×	×	×
SE	10.15	3.85	2.24				
VCG	8.53	1.48	1.21	×	✓	×	×
SE	13.80	5.18	2.70				
VCG	8.70	1.52	1.21	×	×	✓	×
SE	12.93	5.28	2.97				
VCG	8.42	1.46	1.20	×	×	×	✓
SE	14.47	6.13	3.28				
VCG	8.29	1.44	1.18	✓	✓	✓	✓
SE	8.61	3.40	2.04				

Table 6.2: Verification times and slowdown factors of MPP programs compared to Nagini without MPP transformation for different combinations of optimizations. T_{avg} is the average verification time in seconds. The optimizations are the ones described in Section 5.3. Each optimization configuration was run using the Viper backend based on verification condition generation (VCG) and the one based on symbolic execution (SE).

The test setup is such that we first run two examples to give the JVM time to JIT compile, then for each of the 93 examples from Nagini's functional test set we perform the following steps:

1. Standard Nagini translation from Python to Viper programs.
2. Verification of the Viper program with the VCG backend ($\rightarrow T_{VCG}$) and the SE backend ($\rightarrow T_{SE}$).
3. Nagini translation from Python to Viper MPP.
4. Verification of the MPP with the VCG backend ($\rightarrow T_{VCG}^{MPP}$) and the SE backend ($\rightarrow T_{SE}^{MPP}$).

We averaged the measured times for each test file over five runs, and computed the average verification time of the MPPs ($\rightarrow T_{avg}$) and the average and median slowdown factors across all test files for each backend, where the slowdown factors are T_{VCG}^{MPP}/T_{VCG} and T_{SE}^{MPP}/T_{SE} respectively. We repeated this process for different configurations of the optimizations from Section 5.3, the results are listed in Table 6.2.

6.3.1 Performance Impact of MPPs in General

In this section we look at how the MPP encoding affects the verification performance in general. This means we will only consider the last two rows of Table 6.2 where all optimizations are enabled.

Since the MPP duplicates most parts of the source program, one might expect a slowdown in verification time of around two. When we look at the last two rows in Table 6.2 we can see that the verifier based on verification condition generation performs even better than this, with an average slowdown factor of under 1.5. We suspect the reasons for this are that the verification effort is not really doubled, as a certain overhead is only included once. We have mentioned in the last section how the VCG backend has a significant overhead to start the verification, which presumably contributes to the slowdown factor being so low. Also, since the additions to the code are duplications, the SMT solver can use the similarities for more efficient verification. This is possible because VCG generates one formula per method that is handed to the SMT solver, which can therefore see all parts at once and exploit the similarities.

The backend based on symbolic execution on the other hand shows a slowdown more than twice as big on average. Different from VCG, the SE backend creates an SMT formula for each path through the program and invokes the SMT solver on each one separately. This is why the solver cannot exploit the similarities from the duplications here. We believe that this and the fact that the overhead is lower to begin with lead to the bigger slowdown factor.

The average verification times are fairly similar, both backends being within 0.32 seconds of each other. This means that without the MPP encoding, SE performed better on average than VCG. That they perform the same here, we think, again shows that although VCG has a higher constant overhead it scales better with more complex programs.

Another observation we can make, is that the mean of slowdowns is significantly lower than the average. When we look at the distribution of slowdowns per example program (see Figure 6.1), we can see that the majority of examples display close to no slowdown, and only a few show large slowdowns. The figure shows the slowdown factors on the x axis, and the number of tests displaying a given slowdown on the y axis, collected into 20 bins of size 0.54. The figure includes one data point per test program per backend, so there is a total of 186 data points, one of which is not shown in the figure because it has a slowdown factor of almost 64, which would make the figure hard to read. This extreme slowdown was measured with the SE backend. One can see that with the VCG backend over fifty tests are less than 1.4 times slower, another 24 take less than twice the amount of

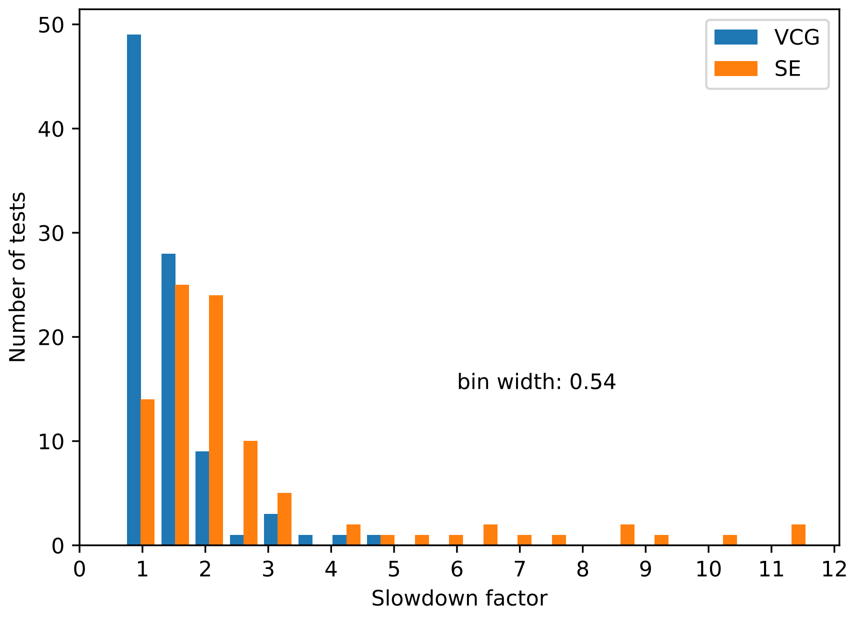


Figure 6.1: Histogram of slowdowns of MPP verification with all optimizations enabled compared to functional verification across 93 test cases in both backends (186 data points total). For better readability one example with slowdown factor 64 is omitted (SE).

time and it tapers off quickly. With the SE backend the same pattern can be observed, albeit less pronounced. Here there are a number of tests which slow down five times and more, with an extreme case of slowdown factor 64. This shows that the MPP encoding, with its additional execution paths, affects SE more strongly than VCG.

6.3.2 Performance Impact of Optimizations

We now analyze the impacts of the different optimizations on the verification performance, using the measurements in Table 6.2.

First of all, with the VCG backend we can observe that the runtimes barely change with different optimizations active. This is not entirely surprising, as all the optimizations generate an equivalent program, so the formula generated by the backend should be equivalent too. The only optimization which does improve performance of the VCG backend, by almost 8 percent, is the one addressing control flow. We believe that the biggest difference here comes from only generating the loop reconstruction where necessary, namely where a return, break or continue might happen. We think the removing of unnecessary control variables should not affect verification too much, as those variables we can remove are assigned to only once, namely at the beginning of the method where they are set to false. Therefore, they should not complicate the generated formula by much. The loop reconstruction on the other hand introduces a large amount of complexity, as the whole loop body is duplicated and the number of possible paths through the program is increased. This is even worse for nested loops, when both the inner and outer loop are reconstructed. In the reconstruction of the outer loop, where we include the loop body, we again have the inner loop together with its reconstruction. Therefore, the complexity grows exponentially with each nesting level, and the optimization where we only generate them where necessary can improve this greatly.

Looking at the numbers for the SE backend we can see that without any optimizations the average runtime is about 80 percent higher than with all optimizations active. The optimization where we create only the necessary control variables is the one with the biggest impact, bringing the average runtime down to about ten seconds from over 15.5 with no optimizations, an improvement of over 35 percent. Again we assume that the large improvement comes from removing the loop reconstruction where it is not necessary. The impact of this is even higher here than with VCG, as SE is much more sensitive to the number of paths through a program.

The removing of duplicate checks of defined variables has the smallest impact. Even though this affects many function calls, each one we remove means only a small improvement as the checks themselves do not affect the runtime by much in the first place.

Both of the other optimizations bring some improvement, in the case of compressing sequential compositions such that we can minimize the number of conditionals it is over 12 percent. Compared to the high impact of the control flow optimization one might expect more here, as this one should decrease the number of paths significantly. However, we believe that since the conditions of those statements we can compress are the same, the price of having the additional paths is not that high, leading to this respectable but not overwhelming improvement.

As for the last one, where we assume that at least one activation variable is true, we observe a performance improvement of over 18 percent. One might think this optimization halves the number of possibilities to consider, as there are four possible combinations of two Boolean values and fixing one leaves only two options. However, the possibility where both activation variables are false can be verified at very little cost, since nothing happens in the method body and all the specifications are trivially true. Therefore, the impact is not quite as high.

Overall combining all the optimization we measured a speedup of over 45 percent for the SE backend, which is quite significant. The effectiveness of the optimizations on VCG on the other hand is far less pronounced.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Modular product programs allow for modular verification of hyperproperties using off-the-shelf verification tools. In this thesis we extended the definition of MPPs by introducing encodings for return statements, break and continue statements and try/catch blocks, in order to make them suitable for encoding a wide range of object-oriented languages. We have presented a way to soundly verify relational programs containing dynamically bound calls, and added support for relational predicates and pure functions to MPPs.

We extended Nagini's specification language to enable expressing information flow security assertions, where in comparing variables of different executions we make a distinction between reference and value equality. For convenience, we added a way to specify whole methods as being low, while still verifying that those methods cannot leak secrets. This makes the tool easier to use and reduces the need for boilerplate specifications in larger code bases.

We adapted the methodology for proving absence of termination channels presented in the MPP paper to be used in Nagini. This involved integrating existing mechanisms of Nagini for proving termination, as well as the generation of additional assertions. We addressed verification of secure information flow for concurrent programs, which entails additional challenges as opposed to sequential programs, by adapting existing methodologies. In particular, we enable verifying possibilistic noninterference, the property that given two runs of a concurrent program using the same low inputs it is possible that we observe the same low outputs.

We have implemented the extended MPP transformation, for the special case of modeling $k = 2$ executions, in Viper, reusing much of the already existing infrastructure. We designed the implementation such that the MPP encoding can easily be reused for encoding other languages. To prove secure information flow of Python programs, we extended Nagini to make use of this transformation. The complete subset of Python that is supported by Nagini can be encoded into MPPs by our implementation.

To evaluate the implementation we encoded example programs from the literature in Python. We showed that with our extensions we can express most of them in Nagini, and get the expected verification result. We assessed the general performance impact of the MPP transformation by comparing the runtimes of tests from Nagini's test suite with and without the encoding. The results showed an acceptable overhead in the majority of test cases, with some exceptions we attributed to the triggering of quantifiers in the MPP. We proposed some optimizations for the MPP encoding and evaluated their performance impact, showing significant improvements for Viper's symbolic execution verifier.

7.2 Future Work

For future work there is a multitude of relational properties which could be proven using modular product programs. For one, there are more secure information flow properties, e.g., the absence of timing side channels, for which a methodology is already introduced in the MPP paper and could be adapted to work with Nagini. For another, there are more hyperproperties unrelated to secure information flow which can be shown using MPPs. For example, one could extend Nagini to prove reflexivity and transitivity of Python's `__eq__` method.

To prove, for example, transitivity, a 3-hyperproperty, our implementation could be adapted to support the more general MPP encoding for modelling $k > 2$ executions, to enable expressing k-hyperproperties.

Besides this, one could further optimize the MPP encoding. For example, currently in a nested loop the body of the inner loop is included a second time in the reconstruction of the outer loop. Since in the verification methodology we consider the loop body has no effect outside the loop (only the loop invariants and condition matter to the outside), it is not necessary to verify the loop body again. Another optimization, affecting the `@AllLow` and `@PreservesLow` decorators, we already described in Section 4.1.3.

Appendix A

Extended MPP Encoding

$[\![\textbf{procedure}\ \mathtt{m}(\mathtt{x}_1,\ldots,\mathtt{x}_\mathtt{m})\ \textbf{returns}\ (\mathtt{result},\ \mathtt{error})\{\mathtt{s}\}]\!]_k$

$\quad = \ \textbf{procedure}\ \mathtt{m}(\mathtt{p}^{(1)},\ldots,\mathtt{p}^{(k)},args)\ \textbf{returns}\ (rets)\{$

$\qquad \bigodot_{i=1}^{k} \mathtt{ret}^{(i)} := \mathtt{false};$ `// true iff returned`

$\qquad \bigodot_{i=1}^{k} \mathtt{break}^{(i)} := \mathtt{false};$ `// true iff after break`

$\qquad \bigodot_{i=1}^{k} \mathtt{cont}^{(i)} := \mathtt{false};$ `// true iff after continue`

$\qquad \bigodot_{i=1}^{k} \mathtt{except}^{(i)} := \mathtt{false};$ `// true iff uncaught exception`

$\qquad \bigodot_{i=1}^{k} \mathtt{error}^{(i)} := \mathtt{null};$ `// store exception`

$\qquad [\![s]\!]_k^{\mathring{p}}$

$\quad \}$

where

$fresh(\mathring{\mathtt{ret}}) \wedge fresh(\mathring{\mathtt{break}}) \wedge fresh(\mathring{\mathtt{cont}}) \wedge fresh(\mathring{\mathtt{except}}) \wedge$

$fresh(\mathring{\mathtt{error}})$

$args = \mathtt{x}_1{}^{(1)},\ldots,\mathtt{x}_1{}^{(k)},\ldots,\mathtt{x}_\mathtt{m}{}^{(1)},\ldots,\mathtt{x}_\mathtt{m}{}^{(k)}$

$rets = \mathtt{result}^{(1)},\ldots,\mathtt{result}^{(k)},\mathtt{error}^{(1)},\ldots,\mathtt{error}^{(k)}$

$[\![s_1;s_2]\!]_k^{\mathring{p}} \ = \ [\![s_1]\!]_k^{\mathring{p}};[\![s_2]\!]_k^{\mathring{p}}$

$[\![\textbf{skip}]\!]_k^{\mathring{p}} \ = \ \textbf{skip}$

$[\![\textbf{if}\ (\mathtt{e})\ \textbf{then}\ \{s_1\}\ \textbf{else}\ \{s_2\}]\!]_k^{\mathring{p}}$

$\quad = \ \bigodot_{i=1}^{k}(\mathtt{p_1}^{(i)} := \mathtt{p}^{(i)} \wedge \mathtt{e}^{(i)});$

$\qquad \bigodot_{i=1}^{k}(\mathtt{p_2}^{(i)} := \mathtt{p}^{(i)} \wedge \neg\mathtt{e}^{(i)});$

$\qquad [\![s_1]\!]_k^{\mathring{p}_1};[\![s_2]\!]_k^{\mathring{p}_2}$

where

$fresh(\mathring{\mathtt{p}}_1) \wedge fresh(\mathring{\mathtt{p}}_2)$

```
$[\![\mathtt{x}_1,\ldots,\mathtt{x}_n\mathtt{:=}\ \mathbf{call}\ \mathtt{m}(\mathtt{e}_1,\ldots,\mathtt{e}_m)]\!]^{\mathring{p}}_k$
  $=$  $\mathbf{if}\ (\bigvee_{i=1}^{k}\mathtt{p}^{(i)})\ \mathbf{then}\ \{$
           $\bigodot_{i=1}^{k}\mathbf{if}\ (\mathtt{p}^{(i)}\wedge\neg\mathtt{ret}^{(i)}\wedge\neg\mathtt{break}^{(i)}\wedge\neg\mathtt{cont}^{(i)}\wedge\neg\mathtt{except}^{(i)})\ \mathbf{then}\ \{$
                 $\bigodot_{j=1}^{m}(\mathtt{a_j}^{(i)}\mathtt{:=}\mathtt{e_j}^{(i)});$
               $\}$
           $ts\mathtt{:=}\ \mathbf{call}\ \mathtt{m}(\mathtt{p}^{(1)},\ldots,\mathtt{p}^{(k)},as);$
           $\bigodot_{i=1}^{k}\mathbf{if}\ (\mathtt{p}^{(i)}\wedge\neg\mathtt{ret}^{(i)}\wedge\neg\mathtt{break}^{(i)}\wedge\neg\mathtt{cont}^{(i)}\wedge\neg\mathtt{except}^{(i)})\ \mathbf{then}\ \{$
                 $\bigodot_{j=1}^{n}(\mathtt{x_j}^{(i)}\mathtt{:=}\mathtt{t_j}^{(i)});$
               $\}$
       $\}$
       where
       $fresh(\mathring{\mathtt{a}}_2,\ \ldots,\ \mathring{\mathtt{a}}_2)\wedge fresh(\mathring{\mathtt{t}}_2,\ \ldots,\ \mathring{\mathtt{t}}_2)$
       $as=[\mathtt{a_1}^{(1)},\ldots,\mathtt{a_1}^{(k)},\ldots,\mathtt{a_m}^{(1)},\ldots,\mathtt{a_m}^{(k)}]$
       $ts=[\mathtt{t_1}^{(1)},\ldots,\mathtt{t_1}^{(k)},\ldots,\mathtt{t_m}^{(1)},\ldots,\mathtt{t_m}^{(k)}]$
$[\![\mathbf{field}\ \mathtt{f}]\!]^{\mathring{p}}_k=\mathbf{field}\ \mathtt{f}^{(1)};\ldots;\mathbf{field}\ \mathtt{f}^{(k)}$
$[\![\mathtt{x:=}\ \mathbf{new}\ (\mathtt{f}_1,\ldots,\mathtt{f}_m)]\!]^{\mathring{p}}_k$
  $=$  $\mathtt{tmp}_x\mathtt{:=}\ \mathbf{new}\ (\mathtt{f_1}^{(1)},\ldots,\mathtt{f_1}^{(k)},\ldots,\mathtt{f_m}^{(1)},\ldots,\mathtt{f_m}^{(k)});$
       $\bigodot_{i=1}^{k}\mathbf{if}\ (\mathtt{p}^{(i)}\wedge\neg\mathtt{ret}^{(i)}\wedge\neg\mathtt{break}^{(i)}\wedge\neg\mathtt{cont}^{(i)}\wedge\neg\mathtt{except}^{(i)})\ \mathbf{then}\ \{$
             $\mathtt{x}^{(i)}\mathtt{:=}\mathtt{tmp}_x;$
           $\}$
       where
       $fresh(\mathtt{tmp_x})$
$[\![\mathtt{x:=e}]\!]^{\mathring{p}}_k$
  $=$  $\bigodot_{i=1}^{k}\mathbf{if}\ (\mathtt{p}^{(i)}\wedge\neg\mathtt{ret}^{(i)}\wedge\neg\mathtt{break}^{(i)}\wedge\neg\mathtt{cont}^{(i)}\wedge\neg\mathtt{except}^{(i)})\ \mathbf{then}\ \{$
             $\mathtt{x}^{(i)}\mathtt{:=}\mathtt{e}^{(i)}$
           $\}$
$[\![\mathbf{return}\ \mathtt{z}]\!]^{\mathring{p}}_k$
  $=$  $\bigodot_{i=1}^{k}\mathbf{if}\ (\mathtt{p}^{(i)}\wedge\neg\mathtt{ret}^{(i)}\wedge\neg\mathtt{break}^{(i)}\wedge\neg\mathtt{cont}^{(i)}\wedge\neg\mathtt{except}^{(i)})\ \mathbf{then}\ \{$
             $\mathtt{result}^{(i)}\mathtt{:=}\mathtt{z}^{(i)};$
             $\mathtt{ret}^{(i)}\mathtt{:=}\mathtt{true}$
           $\}$
```

$[\![\textbf{break}]\!]_k^{\mathring{p}}$

$= \bigodot_{i=1}^{k} \textbf{if}\ (\mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)})\ \textbf{then}\ \{$

$\quad\quad \mathtt{break}^{(i)}{:=}\mathtt{true}$

$\quad \}$

$[\![\textbf{continue}]\!]_k^{\mathring{p}}$

$= \bigodot_{i=1}^{k} \textbf{if}\ (\mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)})\ \textbf{then}\ \{$

$\quad\quad \mathtt{cont}^{(i)}{:=}\mathtt{true}$

$\quad \}$

$[\![\textbf{raise}\ \mathtt{e}]\!]_k^{\mathring{p}}$

$= \bigodot_{i=1}^{k} \textbf{if}\ (\mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)})\ \textbf{then}\ \{$

$\quad\quad \mathtt{error}^{(i)}{:=}\ \textbf{new}\ ();$

$\quad\quad \textbf{inhale}\ \mathtt{typeof}(\mathtt{error}^{(i)}) = \mathtt{typeof}(\mathtt{e})$

$\quad\quad \mathtt{except}^{(i)}{:=}\mathtt{true}$

$\quad \}$

$[\![\textbf{predicate}\ \mathtt{P}(\mathtt{x}_1,\ldots,\mathtt{x}_m)\ \{a\}]\!]_k$

$= \bigodot_{i=1}^{k} \textbf{predicate}\ \mathtt{P}^{(i)}(\mathtt{x}_1{}^{(i)},\ldots,\mathtt{x}_m{}^{(i)})\ \{a^{(i)}\};$

$\quad \textbf{function}\ \mathtt{P}^{(rel)}\ (\mathtt{x}_1{}^{(1)},\ldots,\mathtt{x}_1{}^{(k)},\ldots,\mathtt{x}_m{}^{(1)},\ldots,\mathtt{x}_m{}^{(k)}) : \mathtt{Bool}$

$\quad\ \textbf{requires}\ \bigwedge_{i=1}^{k} \mathtt{P}^{(i)}(\mathtt{x}_1{}^{(i)},\ldots,\mathtt{x}_m{}^{(i)})$

$\quad \{$

$\quad\quad \textbf{unfolding}\ \bigwedge_{i=1}^{k} \mathtt{P}^{(i)}(\mathtt{x}_1{}^{(i)},\ldots,\mathtt{x}_m{}^{(i)})\ \textbf{in}\ \{a^{rel}\}$

$\quad \}$

where

$a^{(i)} \quad = \quad a$ without the rel expressions, using field versions i

$a^{rel} \quad = \quad$ relational expressions from a, as well as $\mathtt{P}^{(rel)}$ if $\mathtt{P}$ is recursive

$[\![\mathtt{P}(\mathtt{x}_1,\ldots,\mathtt{x}_m)]\!]_k^{\mathring{p}}$

$= \bigwedge_{i=1}^{k}(act^{(i)} \Rightarrow \mathtt{P}^{(i)}(\mathtt{x}_1{}^{(i)},\ldots,\mathtt{x}_m{}^{(i)}))\land$

$\quad (\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow \mathtt{P}^{(rel)}(\mathtt{x}_1{}^{(1)},\ldots,\mathtt{x}_1{}^{(k)},\ldots,\mathtt{x}_m{}^{(1)},\ldots,\mathtt{x}_m{}^{(k)}))$

where

$$act^{(i)} = \begin{cases} \mathtt{p}^{(i)} \land \neg\mathtt{ret}^{(i)} \land \neg\mathtt{break}^{(i)} \land \\ \quad\quad \neg\mathtt{cont}^{(i)} \land \neg\mathtt{except}^{(i)}, \text{ in method body} \\ \mathtt{p}^{(i)}, \text{ otherwise} \end{cases}$$

$$
\begin{aligned}
&[\![\mathsf{P}(\mathsf{x}_1,\ldots,\mathsf{x}_m)]\!]^{\mathring{p}}_{k,post}\\
&\quad= \textstyle\bigwedge_{i=1}^{k}(act^{(i)} \Rightarrow \mathsf{P}^{(i)}(\mathsf{x_1}^{(i)},\ldots,\mathsf{x}_m{}^{(i)}))\wedge\\
&\qquad [\,\textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow (\bigwedge_{i=1}^{k-1}(\mathsf{typeof}(\mathsf{self}^{(i)}) = \mathsf{typeof}(\mathsf{self}^{(i+1)})) \Rightarrow\\
&\qquad\qquad \mathsf{P}^{(rel)}(\mathsf{x_1}^{(1)},\ldots,\mathsf{x}_m{}^{(1)},\mathsf{x_1}^{(2)},\ldots,\mathsf{x}_m{}^{(2)})),\\
&\qquad\ (\textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow \mathsf{P}^{(rel)}(\mathsf{x_1}^{(1)},\ldots,\mathsf{x_1}^{(k)},\ldots,\mathsf{x}_m{}^{(1)},\ldots,\mathsf{x}_m{}^{(k)}))]\\
&\qquad \text{where}\\
&\qquad act^{(i)} = \begin{cases} \mathsf{p}^{(i)} \wedge \neg\mathsf{ret}^{(i)} \wedge \neg\mathsf{break}^{(i)}\wedge \\ \qquad \neg\mathsf{cont}^{(i)} \wedge \neg\mathsf{except}^{(i)}, \text{ in method body}\\ \mathsf{p}^{(i)}, \text{ otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{unfold}\ \mathsf{P}(\mathsf{x}_1,\ldots,\mathsf{x}_m)]\!]^{\mathring{p}}_{k}\\
&\quad= \textbf{assert}\ \ \textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow (\bigwedge_{i=1}^{k}(\mathsf{perm}(\mathsf{P}^{(i)}(\mathsf{x_1}^{(i)},\ldots,\mathsf{x}_m{}^{(i)})) = \mathsf{write}) \Rightarrow\\
&\qquad\qquad \mathsf{P}^{(rel)}(\mathsf{x_1}^{(1)},\ldots,\mathsf{x_1}^{(k)},\ldots,\mathsf{x}_m{}^{(1)},\ldots,\mathsf{x}_m{}^{(k)});\\
&\qquad \textstyle\bigodot_{i=1}^{k} \textbf{if}\ (act^{(i)})\ \textbf{then}\ \{\textbf{unfold}\ \mathsf{P}^{(i)}(\mathsf{x_1}^{(i)},\ldots,\mathsf{x}_m{}^{(i)})\}\\
&\qquad \text{where}\\
&\qquad act^{(i)} = \mathsf{p}^{(i)} \wedge \neg\mathsf{ret}^{(i)} \wedge \neg\mathsf{break}^{(i)} \wedge \neg\mathsf{cont}^{(i)} \wedge \neg\mathsf{except}^{(i)}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{fold}\ \mathsf{P}(\mathsf{x}_1,\ldots,\mathsf{x}_m)]\!]^{\mathring{p}}_{k}\\
&\quad= \textstyle\bigodot_{i=1}^{k} \textbf{if}\ (act^{(i)})\ \textbf{then}\ \{\textbf{fold}\ \mathsf{P}^{(i)}(\mathsf{x_1}^{(i)},\ldots,\mathsf{x}_m{}^{(i)})\}\\
&\qquad \textbf{assert}\ \ \textstyle\bigwedge_{i=1}^{k}(act^{(i)}) \Rightarrow \mathsf{P}^{(rel)}(\mathsf{x_1}^{(1)},\ldots,\mathsf{x_1}^{(k)},\ldots,\mathsf{x}_m{}^{(1)},\ldots,\mathsf{x}_m{}^{(k)});\\
&\qquad \text{where}\\
&\qquad act^{(i)} = \mathsf{p}^{(i)} \wedge \neg\mathsf{ret}^{(i)} \wedge \neg\mathsf{break}^{(i)} \wedge \neg\mathsf{cont}^{(i)} \wedge \neg\mathsf{except}^{(i)}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{function}\ \mathsf{f}(\mathsf{x}_1,\ldots,\mathsf{x}_m) : T\ \textbf{requires}\ pre\ \textbf{ensures}\ post\ \{s\}]\!]\\
&\quad= \begin{cases} \left.\begin{array}{l} \bigodot_{i=1}^{k} \textbf{function}\ \mathsf{f}^{(i)}(\mathsf{x}_1,\ldots,\mathsf{x}_m) : T \\ \qquad \textbf{requires}\ \mathsf{pre}^{(i)}\ \textbf{ensures}\ \mathsf{post}^{(i)}\ \{s^{(i)}\}\end{array}\right\}, & \text{if } pre/post/s \text{ depend on the heap,}\\ \left.\begin{array}{l} \textbf{function}\ \mathsf{f}(\mathsf{x}_1,\ldots,\mathsf{x}_m) : T \\ \quad \textbf{requires}\ pre\ \textbf{ensures}\ post\ \{s\}\end{array}\right\}, & \text{otherwise.}\end{cases}
\end{aligned}
$$

$$
\llbracket \texttt{y:=f(x}_1\texttt{,}\ldots\texttt{,x}_m\texttt{)} \rrbracket_k^{\mathring{p}}
= \begin{cases}
\left.\begin{array}{l}
\bigodot_{i=1}^{k} \textbf{if } (\texttt{p}^{(i)} \land \neg\texttt{ret}^{(i)} \land \neg\texttt{break}^{(i)} \land \\
\qquad \neg\texttt{cont}^{(i)} \land \neg\texttt{except}^{(i)}) \textbf{ then } \{ \\
\qquad \texttt{y}^{(i)}\texttt{:=f}^{(i)}(\texttt{x}_1{}^{(i)},\ldots,\texttt{x}_\texttt{m}{}^{(i)}) \\
\quad \}
\end{array}\right\}, & \text{if } f \text{ depends on heap,} \\
\left.\begin{array}{l}
\bigodot_{i=1}^{k} \textbf{if } (\texttt{p}^{(i)} \land \neg\texttt{ret}^{(i)} \land \neg\texttt{break}^{(i)} \land \\
\qquad \neg\texttt{cont}^{(i)} \land \neg\texttt{except}^{(i)}) \textbf{ then } \{ \\
\qquad \texttt{y}^{(i)}\texttt{:=f}(\texttt{x}_1{}^{(i)},\ldots,\texttt{x}_\texttt{m}{}^{(i)}) \\
\quad \}
\end{array}\right\}, & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{l}
[\![\textbf{while } (c) \textbf{ invariant } inv \textbf{ do } \{s\}]\!]_k^{\mathring{p}} \\
= \quad \bigodot_{i=1}^{k} \texttt{bypass}^{(i)} := \neg \texttt{p}^{(i)} \lor \texttt{ret}^{(i)} \lor \texttt{break}^{(i)} \lor \texttt{cont}^{(i)} \lor \texttt{except}^{(i)}; \\
\quad \bigodot_{i=1}^{k} (\textbf{if } (\texttt{bypass}^{(i)}) \textbf{ then } \{\bigodot_{t \in Targets} \texttt{tmp}_\texttt{t}{}^{(i)} := t\}; \\
\quad \bigodot_{i=1}^{k} \{\texttt{oldret}^{(i)} := \texttt{ret}^{(i)}; \quad \texttt{oldbreak}^{(i)} := \texttt{break}^{(i)}; \\
\qquad\quad \texttt{oldcont}^{(i)} := \texttt{cont}^{(i)}; \quad \texttt{oldexcept}^{(i)} := \texttt{except}^{(i)}; \} \\
\quad \textbf{while } (\bigvee_{i=1}^{k} (\texttt{p}^{(i)} \land \neg \texttt{bypass}^{(i)} \land \neg \texttt{ret}^{(i)} \land \neg \texttt{break}^{(i)} \land \neg \texttt{except}^{(i)} \land \texttt{c}^{(i)})) \\
\qquad \textbf{invariant } \lfloor inv \rfloor_k^{\texttt{p}^{(i)} \land \neg \texttt{ret}^{(i)} \land \neg \mathring{\texttt{break}}^{(i)} \land \neg \texttt{except}^{(i)}} \\
\qquad \textbf{invariant } \bigwedge_{i=1}^{k} (\bigodot_{t \in Targets} \texttt{bypass}^{(i)} \Rightarrow \texttt{tmp}_\texttt{t}{}^{(i)} = t) \\
\qquad \textbf{do } \{ \\
\qquad\quad \bigodot_{i=1}^{k} \texttt{cont}^{(i)} := \texttt{false}; \\
\qquad\quad \bigodot_{i=1}^{k} \texttt{p}_1{}^{(i)} := \texttt{p}^{(i)} \land \neg \texttt{ret}^{(i)} \land \neg \texttt{break}^{(i)} \land \neg \texttt{except}^{(i)} \land \texttt{c}^{(i)}; \\
\qquad\quad [\![s]\!]_k^{\mathring{p}_1} \\
\qquad\quad \bigodot_{i=1}^{k} \textbf{inhale } \neg \texttt{p}^{(i)} \lor (\neg \texttt{ret}^{(i)} \land \neg \texttt{break}^{(i)} \land \neg \texttt{except}^{(i)}) \\
\quad \} \\
\quad \textbf{if } (\bigvee_{i=1}^{k} (\neg \texttt{bypass}^{(i)} \land (\texttt{ret}^{(i)} \lor \texttt{break}^{(i)} \lor \texttt{except}^{(i)}))) \textbf{ then } \{ \\
\qquad \bigodot_{i=1}^{k} \{\texttt{ret}^{(i)} := \texttt{oldret}^{(i)}; \quad \texttt{break}^{(i)} := \texttt{oldbreak}^{(i)}; \\
\qquad\qquad \texttt{cont}^{(i)} := \texttt{oldcont}^{(i)}; \quad \texttt{except}^{(i)} := \texttt{oldexcept}^{(i)}; \} \\
\qquad \textbf{inhale } \bigwedge_{i=1}^{k} (\texttt{p}^{(i)} \land \neg \texttt{ret}^{(i)} \land \neg \texttt{break}^{(i)} \land \neg \texttt{except}^{(i)} \Rightarrow \texttt{inv}^{(i)}) \\
\qquad \textbf{inhale } \bigwedge_{i=1}^{k} (\texttt{p}^{(i)} \land \neg \texttt{ret}^{(i)} \land \neg \texttt{break}^{(i)} \land \neg \texttt{except}^{(i)} \Rightarrow \texttt{c}^{(i)}) \\
\qquad \bigodot_{i=1}^{k} \texttt{cont}^{(i)} := \texttt{false}; \\
\qquad \bigodot_{i=1}^{k} \texttt{p}_1{}^{(i)} := \texttt{p}^{(i)} \land \neg \texttt{ret}^{(i)} \land \neg \texttt{break}^{(i)} \land \neg \texttt{except}^{(i)} \land \texttt{c}^{(i)}; \\
\qquad [\![s]\!]_k^{\mathring{p}_1}; \\
\qquad \bigodot_{i=1}^{k} \textbf{inhale } \neg \texttt{p}_1{}^{(i)} \lor \texttt{ret}^{(i)} \lor \texttt{break}^{(i)} \lor \texttt{except}^{(i)} \\
\quad \} \\
\quad \bigodot_{i=1}^{k} \textbf{if } (\neg \texttt{bypass}^{(i)}) \textbf{ then } \{\texttt{break}^{(i)} := \texttt{false}; \texttt{cont}^{(i)} := \texttt{false}\} \\
\quad \text{where} \\
\quad Targets \text{ are all variables assigned to in loop (including control flow flags),} \\
\quad fresh(\mathring{\texttt{p}}_1) \land fresh(\texttt{byp}\mathring{\texttt{a}}\texttt{ss}) \land \forall t \in Targets.fresh(\texttt{t}\mathring{\texttt{m}}\texttt{p}_\texttt{t})
\end{array}
$$

$$
\begin{aligned}
&[\![\textbf{try}\ \{s\}\ \textbf{except}\ e_1 : \{s_1\}\ldots\ \textbf{except}\ e_m : \{s_m\}\ \textbf{else:}\{s_e\}\ \textbf{finally:}\{s_f\}]\!]_k^{\mathring{p}}\\
=\ & \textstyle\bigodot_{i=1}^{k} \texttt{bypass}^{(i)}\texttt{:=}\neg\texttt{p}^{(i)} \lor \texttt{ret}^{(i)} \lor \texttt{break}^{(i)} \lor \texttt{cont}^{(i)} \lor \texttt{except}^{(i)};\\
& \textstyle\bigodot_{i=1}^{k} \{\texttt{oldret}^{(i)}\texttt{:=ret}^{(i)};\quad \texttt{oldbreak}^{(i)}\texttt{:=break}^{(i)};\\
& \qquad \texttt{oldcont}^{(i)}\texttt{:=cont}^{(i)};\quad \texttt{oldexcept}^{(i)}\texttt{:=except}^{(i)};\}\\
& [\![s]\!]_k^{\mathring{p}};\\
& \textstyle\bigodot_{i=1}^{k} \texttt{thisexcept}_1{}^{(i)}\texttt{:=except}^{(i)} \land \neg\texttt{bypass}^{(i)}\\
& \textstyle\bigodot_{i=1}^{k} \texttt{p}_1{}^{(i)}\texttt{:=p}^{(i)} \land \texttt{thisexcept}^{(i)} \land \texttt{issubtype(typeof(error}^{(i)}\texttt{)},e_1\texttt{)};\\
& \textstyle\bigodot_{i=1}^{k} \textbf{if}\ (\texttt{p}_1{}^{(i)})\ \textbf{then}\ \{\texttt{except}^{(i)}\texttt{:=false}\};\\
& [\![s_1]\!]_k^{\mathring{p}_1};\\
& \ldots\\
& \textstyle\bigodot_{i=1}^{k} \texttt{p}_m{}^{(i)}\texttt{:=p}^{(i)} \land \texttt{thisexcept}^{(i)} \land \texttt{issubtype(typeof(error}^{(i)}\texttt{)},e_m\texttt{)};\\
& \textstyle\bigodot_{i=1}^{k} \textbf{if}\ (\texttt{p}_m{}^{(i)})\ \textbf{then}\ \{\texttt{except}^{(i)}\texttt{:=false}\};\\
& [\![s_m]\!]_k^{\mathring{p}_m}\\
& \textstyle\bigodot_{i=1}^{k} \texttt{p}_{m+1}{}^{(i)}\texttt{:=p}^{(i)} \land \neg\texttt{thisexcept}^{(i)};\\
& [\![s_e]\!]_k^{\mathring{p}_{m+1}}\\
& \textstyle\bigodot_{i=1}^{k} \textbf{if}\ (\texttt{p}^{(i)})\ \textbf{then}\ \{\\
& \qquad\quad \texttt{tmp}_{\texttt{ret}}{}^{(i)} := \texttt{ret}^{(i)};\qquad \texttt{ret}^{(i)} := \texttt{oldret}^{(i)};\\
& \qquad\quad \texttt{tmp}_{\texttt{break}}{}^{(i)} := \texttt{break}^{(i)};\qquad \texttt{break}^{(i)} := \texttt{oldbreak}^{(i)};\\
& \qquad\quad \texttt{tmp}_{\texttt{cont}}{}^{(i)} := \texttt{cont}^{(i)};\qquad \texttt{cont}^{(i)} := \texttt{oldcont}^{(i)};\\
& \qquad\quad \texttt{tmp}_{\texttt{except}}{}^{(i)} := \texttt{except}^{(i)};\qquad \texttt{except}^{(i)} := \texttt{oldexcept}^{(i)};\\
& \quad\ \}\\
& [\![s_f]\!]_k^{\mathring{p}};\\
& \textstyle\bigodot_{i=1}^{k} \textbf{if}\ (\texttt{p}^{(i)})\ \textbf{then}\ \{\\
& \qquad\quad \texttt{ret}^{(i)} := \texttt{ret}^{(i)} \lor \texttt{tmp}_{\texttt{ret}}{}^{(i)};\\
& \qquad\quad \texttt{break}^{(i)} := \texttt{break}^{(i)} \lor \texttt{tmp}_{\texttt{break}}{}^{(i)};\\
& \qquad\quad \texttt{cont}^{(i)} := \texttt{cont}^{(i)} \lor \texttt{tmp}_{\texttt{cont}}{}^{(i)};\\
& \qquad\quad \texttt{except}^{(i)} := \texttt{except}^{(i)} \lor \texttt{tmp}_{\texttt{except}}{}^{(i)};\\
& \quad\ \}\\
& \text{where}\\
& \forall i \in [1, m+1].\mathit{fresh}(\texttt{p}_i)\land\\
& \forall x \in \{\mathring{\texttt{bypass}}, \mathring{\texttt{oldret}}, \mathring{\texttt{oldbreak}}, \mathring{\texttt{oldcont}}, \mathring{\texttt{oldexcept}}\\
& \quad \mathring{\texttt{tmp}}_{\texttt{ret}}, \mathring{\texttt{tmp}}_{\texttt{break}}, \mathring{\texttt{tmp}}_{\texttt{cont}}, \mathring{\texttt{tmp}}_{\texttt{except}}\}.\mathit{fresh}(x)
\end{aligned}
$$

$$
\begin{aligned}
\lfloor \texttt{lowEvent} \rfloor_{2}^{\mathring{act}} &= act^{(1)} = act^{(2)} \\
\lfloor \texttt{lowEvent} \rfloor_{2,dyn}^{\mathring{act}} &= (act^{(1)} = act^{(2)}) \land \\
&\quad (act^{(1)} \land act^{(2)} \Rightarrow \texttt{typeof}(\texttt{self}^{(1)}) = \texttt{typeof}(\texttt{self}^{(2)})) \\
\lfloor \texttt{low(exp)} \rfloor_{2}^{\mathring{act}} &= act^{(1)} = act^{(2)} \Rightarrow \mathsf{e}^{(1)} = \mathsf{e}^{(2)} \\
\lfloor \texttt{low(e)} \rfloor_{2,dyn,post}^{\mathring{act}} & \\
= [act^{(1)} \land act^{(2)} &\Rightarrow (\texttt{typeof}(\texttt{self}^{(1)}) = \texttt{typeof}(\texttt{self}^{(2)}) \Rightarrow \mathsf{e}^{(1)} = \mathsf{e}^{(2)}), \\
act^{(1)} \land act^{(2)} &\Rightarrow \mathsf{e}^{(1)} = \mathsf{e}^{(2)}]
\end{aligned}
$$

Bibliography

[1] V. Astrauskas. *Input-Output Verification in Viper*. PhD thesis, Master's thesis, Department of Computer Science, ETH Zürich, 2016.

[2] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Computer Security Foundations Workshop, IEEE(CSFW)*, page 253. IEEE, 2002.

[3] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *International Symposium on Formal Methods*, pages 200–214. Springer, 2011.

[4] G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *International Symposium on Logical Foundations of Computer Science*, pages 29–43. Springer, 2013.

[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004.

[7] L. Blatter, N. Kosmatov, P. Le Gall, and V. Prevosto. RPP: Automatic proof of relational properties by self-composition. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 391–397. Springer, 2017.

[8] P. Boström and P. Müller. Modular Verification of Finite Blocking in Non-terminating Programs. In John Tang Boyland, editor, *29th European*

Conference on Object-Oriented Programming (ECOOP 2015), volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 639–663, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[9] D. Costanzo and Z. Shao. A separation logic for enforcing declarative information flow control policies. In *International Conference on Principles of Security and Trust*, pages 179–198. Springer, 2014.

[10] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing*, pages 193–209. Springer, 2005.

[11] M. Eilers and P. Müller. Nagini: A static verifier for Python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification (CAV)*, LNCS, pages 596–603. Springer International Publishing, 2018.

[12] M. Eilers, P. Müller, and S. Hitz. Modular product programs. In A. Ahmed, editor, *European Symposium on Programming (ESOP)*, LNCS, pages 502–529. Springer International Publishing, 2018.

[13] D. Giffhorn and G. Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, 2015.

[14] R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. A hybrid approach for proving noninterference of java programs. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 305–319. IEEE, 2015.

[15] K. R. M. Leino. This is Boogie 2. *Manuscript KRML*, 178(131), 2008.

[16] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, 2009.

[17] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[18] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[19] M. Parkinson and G. Bierman. Separation logic and abstraction. In *ACM SIGPLAN Notices*, volume 40, pages 247–258. ACM, 2005.

[20] G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307. Springer, 2007.

[21] M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Notices*, volume 51, pages 57–69. ACM, 2016.

[22] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *International Static Analysis Symposium*, pages 352–367. Springer, 2005.

[23] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verification of Information Flow Security for Python Programs

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Meier	Severin

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
Zürich, 14.09.2018	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.