Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enabling Ghost Code in the Nagini Verifier

Bachelor's Thesis

Marco Principe

February 2026

Supervised by Prof. Dr. P. Müller, Dr. M. Eilers
Department of Computer Science, ETH Zürich

Abstract

In program verification, ghost code describes additional data and computations that exist purely for facilitating specifications and providing hints to verifiers. As such, ghost code can be removed from a program without visibly altering its behavior. Nagini, an automated, modular verifier for statically-typed, concurrent Python 3 programs, possessed no wide scale notion of ghost code nor an option to add ghost code by the user.

This thesis proposes a concept and an implementation for ghost code support within Nagini. This takes the form of a type system which determines whether any given element is ghost through user input or inference rules. Depending on the language fragment, we enforce constraints to guarantee the ghost code does not interfere with the regular program. Furthermore, we present a tool which removes the ghost code from a Nagini program to receive equivalent executable Python code.

We show that our implementation correctly supports most of the Python subset that Nagini supports and allows embedded ghost elements in previous Nagini code examples to be clearly marked.

Contents

Chapter 1

Introduction

Python is a widely popular programming language, particularly for its ease of use and expansive library covering a wide range of fields. It is oft deployed in automation and recently has grown a thriving ecosystem in data science.

In stark contrast, there are few tools for formal verification that support it. There are static type checkers, for instance mypy [1], though they do not go beyond type safety. It is for this reason that Nagini is being developed.

1.1 Nagini

Nagini [2] is an automated, modular verifier for statically-typed, concurrent Python 3 programs, built as a front-end to the Viper [3] verification infrastructure.

Nagini works on the principle of Hoare Logic [4]: Assertions, referred to as pre- and postconditions, define what properties must hold on entry or exit of a function respectively. To define these assertions, Nagini provides a library of contracts functions, which the user applies to their code. Nagini will then proceed by running mypy on this code and translating the code to Viper's intermediate verification language (IVL). Finally, Nagini runs Viper, which uses the Z3 SMT solver [5], to discharge proof obligations of these properties.

Let us look at an example. The function presented in Figure 1.1 takes two positive integers `a` and `b` and confirms whether `a` is divisible by `b`.

In comparison, the verifiable version of `is_divisible` with Nagini might look like Figure 1.2. `Requires` and `Ensures` define pre- and postconditions respectively. Thus, the function assumes `a` and `b` to be positive but then guarantees that the return value, specified by `Result`, is `True` if and only if the remainder of `a` divided by `b` is zero. The biconditional is given by the two `Implies` calls. Furthermore, it should be noted that the two `Ensures`

```
1 def is_divisible(a: int, b: int) -> bool:
2
3     if a == b:
4         return True
5     elif a < b:
6         return False
7     else:
8         r = is_divisible(a - b, b)
9         return r
```

Figure 1.1: Example function that tests whether `a` is divisible by `b`.

clauses are treated cumulatively, meaning it would be equivalent if we had combined them to one `Ensures` call with `and`. Lastly, `Decreases` describes that the value of `a` is decreased at every step, which is used for termination guarantees.

```
1  from nagini_contracts.contracts import *
2
3  def is_divisible(a: int, b: int) -> bool:
4      Requires(0 < a and 0 < b)
5      Ensures(Implies(a % b == 0, Result()))
6      Ensures(Implies(Result(), a % b == 0))
7      Decreases(a)
8
9      if a == b:
10         return True
11     elif a < b:
12         return False
13     else:
14         r = is_divisible(a - b, b)
15         return r
```

Figure 1.2: Previous `is_divisible` function with Nagini program annotations.

Many verifiers designed for particular programming languages write their specifications in special comments. This guarantees that the program will automatically ignore them on execution. However, they also need to implement a lexer and a parser to correctly read these comments because these usually do not appear in a program's AST. By comparison, Nagini writes its specification as "normal" code and simply uses the Python AST. This means, in return, that Nagini's specification is bound by Python's grammar. Furthermore, all Nagini elements must be removed before the verified code can be correctly executed, as many do not have proper runtime semantics.

1.2 Ghost Code

According to J. -C. Filliâtre et al., *Ghost code*, as opposed to *regular code*, does not alter the state of the system during execution, but rather serves to facilitate the specification that is needed for a static verifier. Therefore, it can be safely removed from the program without affecting its behavior [6].

Ghost code can store values and make computations which can then be used in program annotations, i.e. pre- and postconditions, invariants, assertions, etc. However, it may not modify regular values, be read as part of regular computations or change the control flow of regular code, e.g. by raising an exception.

To illustrate this, let us revisit the annotated `is_divisible` function from before. Figure 1.3 shows the same verifiable function but implemented in Dafny [7], a verification-aware programming language.

```
1  method is_divisible(a: int, b: int)
2   returns (r: bool)
3   requires 0 < a && 0 < b
4   ensures r <==> (a % b == 0)
5   decreases a
6   {
7
8       if (a == b) {
9           return true;
10      } else if (a < b) {
11          return false;
12      } else {
13          var r := is_divisible(a - b, b);
14          return r;
15      }
16  }
```

Figure 1.3: Verifiable `is_divisible` function implemented in Dafny.

Imagine we want to extend the specification of this function. In this case, it could be helpful to know that given `a` is divisible by `b`, what the result of a division would be. Hence, we add ghost code to keep track as seen in Figure 1.4. The function now returns an additional ghost integer `f`. If `a` is not divisible by `b`, we arbitrarily return `-1`. Otherwise, we set `f` to be `1` in the base case and recursively increment it at every step.

Fundamentally, any regular fragment of a programming language can be ghost, such as variables, functions or statements. Therefore, we say that any element's *ghost status* describes whether it is ghost. Furthermore, we

```
1  method is_divisible(a: int, b: int)
2   returns (r: bool, ghost f: int)
3   requires 0 < a && 0 < b
4   ensures r <==> (a % b == 0)
5   ensures r ==> a == b * f
6   decreases a
7   {
8
9       if (a == b) {
10          return true, 1;
11      } else if (a < b) {
12          return false, -1;
13      } else {
14          var r, f := is_divisible(a - b, b);
15          if (r) {
16              f := f + 1;
17          }
18          return r, f;
19      }
20  }
```

Figure 1.4: `is_divisible` function with added ghost integer return `f`, which is the result of a division of `a` by `b` when the remainder would be zero.

abstract elements as being stored either in the regular state or in the *ghost state*, depending on their ghost status.

1.2.1 Soundness Properties

In order for ghost code to be sound, it must not interfere with the behavior of the program. We will break this down into a number of properties.

1. Ghost code may not change the regular state in any way, such as by overwriting the values of variables. Otherwise, the regular state could differ during execution once the ghost code is removed. Similarly, the regular code may not read or write to the ghost state, since the ghost state would not exist during execution. However, the ghost code may read the regular state, which is oft required for sensible specifications. In short, we refer to this property as *Separation*.
2. Ghost code must not change the control flow outside of itself. This includes, for example, raising exceptions that can escape into regular code. Otherwise, it could change what statements are executed and when. We will refer to this property as *Conformity*.
3. Any piece of ghost code must terminate. As one might expect, calling a

function that runs indefinitely would affect the program's behavior. We will refer to this property as *Termination*.

1.3 Contributions and Outline

Prior to this thesis, Nagini did know some ghost elements. For instance, there are abstract data types, which can store data to be used in later program annotations, or ghost data types, like `PSet`. However, it had no wide scale notion of ghost code, any way to mark auxiliary elements as ghost or a method to identify any singular element as ghost. Moreover, there was no automated process to remove the ghost code, which includes the user's program annotations, from the code once it has been verified.

In this thesis, we propose a concept and an implementation for ghost code support within Nagini. This allows users to write code as shown in Figure 1.5. Furthermore, we present a tool which extracts the regular code from a Nagini program. The code below can, thus, be restored to the same as in Figure 1.1.

```
1  from nagini_contracts.contracts import *
2  from typing import Tuple
3
4  def is_divisible(a:int, b:int) -> Tuple[bool,GInt]:
5      Requires(0 < a and 0 < b)
6      Ensures(Implies(a % b == 0, Result()))
7      Ensures(Implies(Result()[0], a % b == 0))
8      Ensures(Implies(Result()[0],
9                      a == b * Result()[1]))
10     Decreases(a)
11
12     if a == b:
13         return True, 1
14     elif a < b:
15         return False, -1
16     else:
17         f: GInt = 0
18         r, f = is_divisible(a - b, b)
19         gr: GBool = r
20         if gr:
21             f += 1
22         return r, f
```

Figure 1.5: `is_divisible` function with previous `f` ghost return value, but implemented in Nagini with our added ghost code support

In Chapter 2, we describe our concept of ghost code within Nagini. Chapter 3 details how we implemented this concept and what limitations we faced. In Chapter 4, we evaluate our implementation and compare changes to the syntax. Finally, Chapter 5 concludes with final remarks and by outlining possible future steps.

Chapter 2

Ghost Code in Nagini

When we began our work to extend Nagini with a framework that can support ghost code, we identified the following constraints:

- Our solution must support the subset of the Python language currently supported by Nagini. We are, however, free to choose what element from that subset can be ghost, as long as the result is sound and compatible with Nagini's design.
- All code must be accepted by the Python parser and by mypy's type checking. In other words, any solution is bound by Python's grammar.
- For the sake of backwards compatibility, we aim to minimize any changes to Nagini syntax.

For the rest of this chapter, we will elaborate on the principles, rules and details of our concept.

2.1 First Approaches with Ghost Types

Let us start with a seemingly simple question: How do we mark a variable as ghost? Some verifiers, such as Dafny, have a `ghost` keyword that can be employed to mark variables as ghost. However, Nagini is bound by Python's grammar and we can, thus, not introduce such a keyword. Nevertheless, when declaring a variable, it can be given a type annotation.

Since Nagini already runs mypy, our first approach was, thus, to create "*ghost types*" which encode the ghost status of an element. Soundness, or at least separation, could then be enforced by mypy through existing typing rules.

When we decide how such a ghost type should be designed, we still want a variable to behave broadly similar. This means it should still support addition for example, regardless if it stores a regular or a ghost integer. Ghost types should, therefore, be defined in relation to their base types.

One important part of separation is that the regular state must not read the ghost state. We will cover this in greater depth in Section 2.4, but this means that a ghost value cannot be assigned to a regular variable. On the other hand, typing rules dictate that any value can be assigned to a variable if the value's type is a subtype of the variable's type. Putting this all together, the ghost type must be a supertype of its base version.

This poses a problem though. While Python does support defining subclasses of built-in types, it does not, although for good reason, allow to do the same with superclasses.

Hence, we cannot enforce separation with ghost types inside the existing type system. Nevertheless, we might still be able to use mypy to collect ghost information and we can then build a separate system to check for soundness. So, for our second approach, we define ghost types to be subclasses of their respective base type.

This approach, however, is not without its issues either. For example, if we were to add two variables together and only one of them is ghost, we would want the result to be a ghost type because the regular state should not read ghost state. Typing rules, on the other hand, would say the result would be a regular type because addition over integers returns an integer and not any subtype. Similarly, assigning a regular value to a ghost variable would not be allowed.

While there are ways to broadly overwrite this behavior, such as defining a metaclass which redefines class functions to return ghost types, they are too complicated for mypy to understand statically and are tricky to get right.

In conclusion, employing mypy for collecting ghost information with ghost types is not reasonably viable. Instead, we built up a separate system which bypasses mypy entirely.

2.2 A Type System over Ghosts

If we cannot create ghost types that are super- or subtypes, what if we simply create ghost type aliases? Since they are type aliases, they are interpreted the same as their base types as far as mypy is concerned. Consequently, mypy enforces no constraints on ghost code, but this still leaves us with type aliases we can assign special meaning to. For simplicity, we will refer to these special type aliases as *ghost names*. We will go over how they are defined later in Subsection 2.3.1.

With this in mind, our approach was to build up a type system that assigns every element a ghost status derived from user input or inference rules. This type system does not interface with Python's standard type system, but enforces its own constraints on user code.

Therefore, we have complete control over what elements can be ghost and what rules they must follow. In turn, we must be able to handle each code fragment and enforce any constraint ourselves. With this in mind, let us define the rules within our type system more concretely.

Rule 2.1 *Fundamentally, any expression or statement from our Python subset can be ghost.*

Therefore, ghost state may include variables, parameters, fields, functions, classes and more. We will cover the exceptions to this rule more closely in Section 2.4.

Rule 2.2 *Every element has a ghost status. Under almost all circumstances, these must be binary and cannot be mixed, meaning any element is either regular or ghost but not both. The only exception to this rule is the return value of a function, whose regular and ghost elements the caller must separate when the call completes.*

In order to simplify fulfilling separation, we generally try to forbid combining any ghost and regular elements unless necessary to facilitate useful specifications. Since a function in Python can only return a singular value and we want to support additional ghost returns, we must allow mixed returns.

Moreover, when we handle mixed returns, we want to assure that the user will still be able to easily determine the ghost status of each return. Consequently, we define a further constraint.

Rule 2.3 *Mixed returns must have the structure of* `Tuple[R, G]`*, where R is the regular return and G is the ghost return. As normal, R and G can each be a tuple that holds multiple return types.*

2.3 Marking and Inferring Ghosts

In general, a verifier can choose whether the ghost status of any given fragment of code is inferred or supplied by the user. Having the user state what they want is important and oft makes it clearer in future reviews of the code which elements are ghost. However, there are cases where further input is redundant and should, thus, not be required.

Hence, we introduce the concept of a *ghost context*: If we know an element to be ghost, primarily because it has been marked as such, we consider any sub-element to be in a ghost context. For example, if we define a ghost function, any element that is part of that function, such as its parameters or the statements in its body, must be ghost.

For Nagini, we stick to the following rules.

Rule 2.4 *Variables, fields, functions and classes must be marked as ghost when they are declared, unless their declaration is in a ghost context. If these elements are unmarked, they are considered regular.*

Rule 2.5 *Expressions, statements and elements in a ghost context are inferred.*

2.3.1 Variables and Fields

As stated before, we use ghost names to mark variables as ghost. For fields, the same approach works as well.

```
1  i = 0                # Regular
2  j: int = 0           # Regular
3  gi: GInt = 0         # Ghost
4
5  GList = List[int]
6  MarkGhost(GList)     # Newly defined ghost name
7
8  gh_i: GList          # Ghost
9  t: List[int]         # Regular
10 gt: List[GInt]       # Ghost
```

Figure 2.1: Example of marking variables as ghost, where `GInt` is a ghost name for ghost integers we have built into Nagini's library.

It is worth mentioning that this means that any newly declared variable with no type annotation are understood to be regular.

For Python's built-in types, we add corresponding ghost names to Nagini's library, for instance `GInt` to declare a ghost integer. Moreover, a user can define their own ghost names, which are particularly applicable for user defined classes, with the `MarkGhost` function, as seen in Figure 2.1.

When we annotate a variable or field that holds a generic class, we do not need to define a new ghost name. Instead, any generic type that contains a ghost name is treated equivalently to a ghost name. Similarly, a generic ghost class is still treated as ghost even if its contained type is a normal type annotation.

However, this does not allow a user to mix ghost names with normal type annotations in a generic. For example, an annotation of `Union[GInt, str]` would be unclear whether the associated element should be ghost or not and is, hence, rejected.

One special case for variable definitions to underline here are loop variables, which are created in for loops and in comprehensions. The loop variable always contains an element of the iterable and, as a consequence, must

always have the same ghost status as the iterable. Therefore, loop variables are inferred and should not be marked ghost elsewhere.

2.3.2 Functions and Classes

Functions and classes are marked as ghost by using the `@Ghost` decorator we added.

```
1  @Ghost
2  def foo(i: int, j: int) -> int:
3      ...
4
5  @Ghost
6  class A:
7      def __init__(self) -> None:
8          ...
9
10 def bar(i: int, gs: GInt) -> GInt:
11     ...
```

Figure 2.2: Example in marking functions and classes as ghost.

Since they are within a ghost context, any element declared within a ghost function or class must similarly be ghost. For example, the `__init__` method in Figure 2.2 is ghost because it is belongs to a ghost class.

The ghost status of a function's return can be marked by using ghost names for the annotation in the same way as for variables and fields. For instance, the `bar` function would normally return `None`, but we have added a ghost integer return.

When it comes to adding ghost arguments to a function, we assume a user knows how many they wish to pass. Since functions in Python can have variadic arguments, which enables them to accept any number of arguments, we limit them.

Rule 2.6 *Variadic arguments must always be regular. Thus, ghost functions cannot have them.*

Similarly, mypy provides generic types, which are used in polymorphic methods. While we deem polymorphic ghost functions as potentially helpful, we do not see value for generic ghost arguments in regular functions.

Rule 2.7 *Generic types cannot be used in regular functions for ghost arguments. They are allowed, however, for regular arguments and, in ghost functions, for ghost arguments.*

Lastly, predicates, functions that argue over the permissions of an object, in Nagini are only allowed for specification. Therefore, functions with the `@Predicate` decorator are inferred to be ghost as well.

2.3.3 Expressions

Rule 2.8 *An expression is ghost if any of the following apply:*

- *It is a reference to a ghost element, such as a call to a ghost function.*
- *Any of its sub-expressions are ghost, excluding the arguments and the return value of a call.*
- *It is a lambda expression.*
- *It is a pure call and used as described in Subsection 2.4.4*

Otherwise, the expression is considered regular. Hence, any expression that reads from the ghost state is ghost, such as an addition with a ghost variable. Inversely, constants, for example, are never inherently ghost.

Lambda expressions are treated as ghost because Nagini only allows them in specific contract calls.

2.3.4 Statements

Rule 2.9 *A statement is ghost if any of the following apply:*

- *It is inside a ghost context.*
- *It is an assignment to only ghost variables, unless its right hand side is a singular impure regular call (c.f Subsection 2.4.4).*
- *It is a* `delete` *of only ghost objects, such as ghost variables.*
- *It is a control flow statement, i.e.* `if`*,* `for` *and* `while`*, whose iterable or condition is ghost.*
- *It contains only a ghost expression, such as a call to a ghost function.*

Any other statement is considered regular, though it may still contain ghost code. Consequently, there are statements that are always treated as regular, such as `continue`, `break`, `assert`, `with`, etc. Specifically for ghost assertions, though, Nagini provides an `Assert` function.

2.3.5 Nagini Built-ins

Nagini itself defines a wide range of functions for describing specifications. As such, most of Nagini's built-in functions are already marked as ghost.

Two special cases are the `Unfolding` and `Reveal` contract functions. They are used solely for specification purposes, evaluating an expression while temporarily unfolding a predicate for Nagini's separation-style permissions [8] and revealing the implementation of a pure function respectively. However, they are conditionally allowed to be used within regular code. Specifically, they have a *key argument*, as we will call it. This is the given expression for `Unfolding` and the applied function for `Reveal`. If their key argument is regular, they are allowed to be used in regular code. Formally, they are treated as pure, regular functions, but are still to be removed after verification, the same as ghost code. In practice, their ghost status can be inferred to be the same as its key argument.

Similarly, Nagini defines `Thread` and `Lock` classes. These are for handling concurrency and their functions can hold additional ghost parameters, but they are, otherwise, meant to be placeholders for Python's built-in threading and locking tools.

2.4 Invalidity Rules and Soundness

We now know which elements of our code are ghost or not. However, we still need to ensure that our three properties hold for them. As such, we will dive into each property and define various constraints to guarantee soundness. Afterwards, we will cover calls separately because they can potentially violate any of the three properties.

2.4.1 Separation

Recall that separation states that ghost code must not alter the regular state. The ghost state may only read the regular state, while the regular state must not read or write to the ghost state. This property primarily concerns how elements may be assigned.

Rule 2.10 *For any sort of write, assigning an expression* `e` *to a target* `t` *is sound if all of the following apply:*

- *If* `e` *is ghost then* `t` *must be ghost.*
- *If this write happens in a ghost context,* `t` *must be ghost.*
- *If* `t` *is a subscript,* `t` *must be regular.*

Let us consider each of these cases in order. Firstly, since the ghost state may read the regular state, we have no limitations for assigning a regular `e`. However, regular state must not read the ghost state. Hence, the only case we must forbid is a ghost `e` and a regular `t`.

Secondly, if we are in a ghost context, the write must be in ghost code. Nevertheless, ghost code must not write to the regular state. In particular,

even if both `e` and `t` are regular, the write would have visibly altered the regular state. Otherwise, it would be possible to increment a global regular variable in a ghost function, which we could then read elsewhere.

The last point is similar by comparison. As is commonly known, Python has mutable objects, i.e. strings, lists and dictionaries. A variable can store a reference to such an object, but to alter the underlying object, a subscript like `l[1]` is used. While there is nothing wrong with storing a reference to a regular object in a ghost variable, altering the object itself would once again change the regular state. Furthermore, tracking if any given variable points to a regular or a ghost object is intensive at best and undecidable at worst. Therefore, we enforce the following restriction.

Rule 2.11 *Ghost code may not modify any objects.*

Although the objects may not be directly changed, new objects can still be created and variables can still have their references to those objects updated. Thus, if a user wants to update a ghost object, they can create a new object with all the same values except the desired changes and set the variable to point to the new object.

We now understand how values may be assigned. However, there are other cases that also impact separation but do not directly fall into the concept of reads and writes.

For one, there is class inheritance. Allowing a regular class to inherit from a ghost class would make it possible to pass down behavior that would not exist during execution. Inversely, a ghost class that explicitly inherits from a regular class could potentially call functions that alter the regular state.

Rule 2.12 *Classes may only inherit from classes with the same ghost status. The only exception is the* `object` *class, as it is the superclass of every class.*

On the other hand, a `delete` statement also affects the state. Therefore, a `delete` must not remove regular elements in a ghost context. Although not strictly necessary, we also forbid a singular `delete` statement to delete elements of differing ghost statuses for simplicity.

Lastly, we must be careful with the unpacking of elements. Since the unpacked element can be a call, whose return value may be a mixed tuple, a singular assignment statement could write to regular and ghost values simultaneously. Primarily, this involves confirming that Rule 2.10 is followed for each write. In addition, for simplicity, we only allow writing to mixed targets if every value is assigned to a target of the same ghost status. Thus, assigning only to ghost targets is still permitted. Similarly, we forbid the use of the packing operator `*` in an assignment when any involved element is ghost. We illustrate these points in Figure 2.3.

```
1  def foo() -> Tuple[Tuple[int, int], GInt]:
2      ...
3
4  i = j = 0       # Regular
5  gi: GInt = 0    # Ghost
6  gj: GInt = 0    # Ghost
7  gk: GInt = 0    # Ghost
8
9  (i, j), gi   = foo()        # OK
10 (i, gj), gi  = foo()        # Invalid
11 (gi, gj), gk = foo()        # OK
12
13 i, *t, j  = [0, 1, 2, 3]    # OK
14 gi, *t, j = [0, 1, 2, 3]    # Invalid
```

Figure 2.3: Unpacking with mixed targets must be straightforward.

In conclusion, we verify separation by monitoring state changing statements and class inheritance.

2.4.2 Conformity

Recall that conformity describes that ghost code must not change the control flow outside of its own scope. While this can be verified through a robust analysis, for example confirming that every raised exception is caught, we believe that many control flow altering statements are neither necessary nor all that helpful for ghost code.

Rule 2.13 *Ghost code must not raise any exceptions or contain any* `continue` *or* `break` *statements. Similarly, ghost* `return` *statements are only allowed if they are used in a ghost function.*

Thus, proving conformity is mostly trivial, as we just check ghost code for these statements. Nevertheless, we must ensure that ghost code raises no unintended exceptions, such as a `ZeroDivisionError`. Fortunately, Nagini already verifies this for all code.

2.4.3 Termination

In order to guarantee that termination is satisfied, we simply need to prove that every block of ghost code terminates. However, there is no need to prove that regular statements that contain ghost elements must terminate. After all, statements that fall under this category, e.g. assignments to regular and ghost variables, functions with ghost parameters, etc., will not run forever due to the addition of these ghost elements.

With this in mind, we will not go into further details regarding how to prove termination for singular statements or loops because this is an orthogonal issue to the question of sound ghost code and there are other resources [9] that cover it in greater depth.

2.4.4 Calls

Calls to functions must be handled carefully. Their return values are the only element which may combine regular and ghost elements into one mixed object. Furthermore, they may execute statements with arbitrary side-effects. As such, they could violate any of our three properties if not constrained.

As with assignments, we must check that the given arguments may be passed to the respective parameters as described in Section 2.4.1. More importantly, since calls can have mixed returns, we must ensure that unpacking those returns fulfills separation. To unpack a tuple, one can employ either an assignment or the unpacking operator `*`. Although both are sound, stars allow for more complicated expressions that are more difficult to verify.

Rule 2.14 *The unpacking operator `*` must not be used to unpack calls with mixed returns.*

Nevertheless, we still need to show whether calling a given function in regular or ghost code is sound. To start with, we need a better understanding of pure and impure functions and how they interact with ghost code.

Pure functions are deterministic and do not have side-effects. In other words, they always return the same output if given the same input and they do not alter anything outside of their local state. Consequently, they always satisfy all three properties. In comparison, *impure functions* can be non-deterministic or have side-effects. When a function should be pure, the user must annotate it as such by using Nagini's pre-existing `@Pure` decorator.

When we call a function, we need to differentiate three cases:

- The function is ghost.
- The function is regular and pure.
- The function is regular and impure.

Ghost functions must fulfill our three properties already. Thus, it is safe to call them within ghost code. It is worth mentioning that ghost functions may not necessarily be pure. Although sound ghost code will most likely be pure, it does not need to be side-effect-free, as it may change the ghost state outside of the local scope.

Pure regular functions are safe to call in ghost code. In essence, if we call the function in ghost code, we can treat them as if they were a ghost function in

this instance. Similarly, we can treat the instance as ghost when it is given ghost arguments instead of regular arguments.

Rule 2.15 *When a pure function is called, we treat the call equivalently as if it were a ghost function if any of the following apply:*

- *This call appears in ghost code.*
- *We expect an argument to be regular but receive a ghost argument instead.*

Impure regular functions, however, can never be safely called in ghost code because we cannot confirm they will not interfere with regular code. We must, therefore, verify that impure regular functions are never called in any ghost statement or in any ghost context.

Thus, we conclude the following.

Rule 2.16 *Regular code may call regular functions. Ghost code may only call functions that are ghost themselves or are pure.*

Lastly, there exists a particular case where we cannot know for certain if the called functions will be pure. If a function has a parameter that is a union of two or more classes and we call a class function on that parameter, we cannot statically know which function is executed at runtime, assuming all classes implement the given function.

For our purposes, we check that all possible functions have the same signature in regards to which elements are ghost. If a singular possible function is impure, however, then we treat them all as impure.

Chapter 3

Implementation

Following the rules outlined in the previous chapter, we have extended Nagini to support ghost code and to check for the relevant properties. Furthermore, we have implemented an extractor that removes all ghost code from verified Nagini code.

3.1 Ghost checking

Nagini's previous verification process could be split into four broad phases, which are executed in order:

1. Type Check, where we run mypy to catch type errors and gather the types of every variable, field, etc.
2. Analysis, where we build data structures for later phases.
3. Translation, where the code is translated to Viper's IVL.
4. Verification, where the code is verified by Viper.

In the Analysis phase, Nagini runs over the Python AST and builds program nodes that abstract the elements of a program and tracks useful information. For example, for a function, it defines a `PythonMethod` object which stores what parameters it has, whether it is pure, if it is associated with a class and more. For our first change, we added an additional boolean field to the program nodes for variables, fields, functions and classes which tracks whether the given element is ghost.

In order to set those flags, we collect auxiliary information in the early phases. Firstly, during the Type Check phase, we capture any ghost names the user defined. Specifically, this entails looking out for `MarkGhost` calls, confirming the call is used correctly and storing the ghost name in the program node of the corresponding module. As such, it is possible for two modules to define

type aliases with the same name, but only one is a ghost name, without causing them to interfere with one another. Next, in the Analysis phase, we identify all elements the user has marked as ghost and set the flag on the respective program nodes.

At this point, we added another phase to the verification process: A Ghost Check phase in between the Analysis and the Translation. We elected for this to be its own phase to logically separate it from the other phases and to ease its inclusion into the Nagini pipeline. During this phase, we run over the Python AST once more and verify that the ghost code follows the rules we outlined in Chapter 2. The only thing we do not verify here is termination, which we will cover in Section 3.2

One unique challenge we faced, which only concerned the implementation, was the resolution of functions, particularly their calls, to Nagini's respective program node. Nagini possesses a mechanism to get them, but it does not succeed under all cases. For example, functions from Nagini's library have no program node as they are statically known and, otherwise, would require building the same structures during every run. Nevertheless, each such case had to be accounted for in our resolution process. Avoiding the resolution altogether was not an option either because the program node of a function stores whether it is ghost, which we must know for our analysis.

Whenever we evaluate the ghost status of an AST node in this phase, we also dynamically add two auxiliary boolean fields on each node: `is_ghost` and `contains_ghost`. As one might expect, `is_ghost` simply stores the result of the evaluation. Meanwhile, `contains_ghost` expresses if this element contains anything which must be removed before execution. This can be because the element itself is ghost, any of its sub-elements are ghost or it contains any of the special Nagini structures we previously introduced, such as `Unfolding`. Thus, `contains_ghost` will always be set if `is_ghost` is set. While these flags are used to make the ghost status of some elements widely available during the Ghost Check phase, their primary purpose is to simplify the processes described in the following sections.

3.2 Termination Analysis

We explained before that for ghost code to be sound, every block of ghost code must terminate. Since Nagini already had a mechanism for proving termination [9], we decided to base our analysis on that.

If a user wants to prove that a function or loop terminates, things would proceed as follows: The user would use the `MustTerminate` contract functions to describe in how many steps the element must terminate. Nagini would then, in the Translation phase, take that call and translate it to a termination obligation. Finally, Viper either returns a proof or rejects the program with

details on why it failed. Thus, termination is ensured for every element with a termination obligation through an existing encoding.

Consequently, if we want a specific section of code to terminate, we simply need to add an obligation into the Viper program that encodes that. To achieve this, we implemented a *terminating section* wrapper. This wrapper can be placed around any Viper code and defines a "permission" of a termination obligation, which we acquire before the code and release afterwards.

Finally, we can verify the final property of our ghost code. We extended Nagini's Translation phase so that whenever a ghost statement is translated, we add a terminating section around its translated code. Ghost statements are identified here by reading the previously added `is_ghost` flag of each AST node. Currently, this is mostly done for each statement separately for simplicity, though most statements that hold more statements, i.e. `if`, `while` and `for`, are put into a singular terminating section.

3.3 Extraction

Prior to this thesis, there was no tool to automatically remove existing ghost elements, which includes contracts and other program annotations. Thus, this had to be done by hand. Since we can identify every ghost element now, we have introduced an optional Extraction phase. If the user runs Nagini with the `--extraction` argument, this phase is executed after the Ghost Check phase and prints the resulting code to the terminal. It is worth mentioning that we could have placed this new phase after the Verification phase as well.

Our goal for the extraction was that everything from or for Nagini is removed so that the resulting program can be correctly executed without it.

During the Extraction phase, we run an extractor which takes as input the Python AST of the user's code and outputs a new AST with the ghost elements removed. To achieve this, the extractor goes over the AST and reads the `is_ghost` and the `contains_ghost` field of each statement node. If the statement is ghost, it can simply remove the statement wholesale. A regular statement that contains no ghost elements requires no changes and is directly replicated. The only cases that need to be analyzed further are regular statements that contain ghost elements.

For most of these statements, this boils down to checking each sub-element with the same methodology until only regular elements remain. For example, for a class definition, we run over its body and check for ghost elements. Nevertheless, there are a few edge cases to consider on regular statements that contain ghost elements:

- For assignments, all ghost targets need to be removed. However, if this leaves the statement with no targets left, it ceases to be an assignment. This can happen when we call an impure function and store its returns in ghost variables. Since the call may have side-effects, removing it entirely would be unsound. Hence, we leave the new statement as a singular call whose returns are thrown away.
- For statements that contain more statements, there are no constraints that they must contain a regular statement. As such, it is possible for the new body of a statement to be empty. In cases where Python would not allow this, for instance the body of an `if`, we add a `pass` statement.
- For calls, aside from removing ghost arguments, we need to handle the special Nagini calls of `Unfolding` and `Reveal`. As stated before, their ghost status depends on whether their key argument is ghost or not. Therefore, if we encounter them in a regular statement, we must remove them and put their key argument in their place.

Finally, we also remove any imports from Nagini's library as the resulting program should no longer need them. For those interested, examples of the extractor are shown in Chapter 4.

3.4 Limitations

There exist limitations to our implementation. This thesis was subject to strict time constraints, which necessitated some concessions.

Firstly, there are fragments that we want to support, but currently do not. This includes the following:

- Ghost fields and ghost global variables can often be accessed in a wider scope than local variables. Fields can even be added dynamically. While our current system should be readily extendable to support them, they cannot be defined as ghost at this point.
- Recall that variadic arguments are not allowed to be ghost. Consequently, we must confirm that any ghost argument passed in a call is not given to the variadic argument. Combined with the complexities of how arguments can and must be passed, positional, keyword or only one of the two, we currently do not allow functions with variadic arguments to be called.
- To reiterate, we do not allow unpacking operators to be used on calls. Nevertheless, this should only impact functions with mixed returns, yet it currently forbids the operator to be used on all calls, even when it would be sound.

- As established, ghost variables can refer to objects which may be regular or ghost. Furthermore, ghost code may create new objects and assign them to ghost variables. Thus, it could be useful for ghost code to be able to create an instance of a regular class and store it only in the ghost state. However, this requires a kind of ghost constructor, as the normal `__init__` function may not be safe to call. This is further complicated by the fact that Python allows a function to only have one constructor. As such, we would need to supply some replacement for ghost constructors.
- The `Thread` and `Lock` classes from Nagini should be replaced by their Python versions during the extraction. However, this is not done at this point. Hence, these classes remain and have to be replaced by hand. Moreover, the `Lock` class specifically is only partially supported for verification.
- Functions with the `@ContractOnly` decorator in Nagini are assumed to fulfill their specifications and are not further checked. These are, therefore, also skipped in the Ghost Checking phase. The Extraction phase does not yet skip them, so the extraction fails because the necessary flags, i.e. `is_ghost` and `contains_ghost`, do not exist.

Secondly, there is potentially unsound code that we do not catch. These are the following problems we are aware of:

- Class methods in Python have a "self" argument. For example, `split`, which splits a string into multiple substrings based on the given separator, can be called on a string variable with the dot notation, i.e. `s.split(',')`. The variable does not just denote where the function can be found, as would be normal, but also on which element to execute it on. This argument is currently not analyzed and we, therefore, will not notice the implications if we execute a pure regular function on a ghost variable. Similarly, if we chain calls together, e.g. `s.upper().split(',')`, we will only analyze the final call, i.e. `split` in this example.
- Python allows users to overwrite the behavior of built-in operations, for instance addition. Since the new functions are checked themselves, this poses no direct problems for sound ghost code. However, a user could use a ghost name to mark a variable that is intended to be regular as ghost. This could have unintended effects on regular code. Hence, we must extend our implementation to verify that such overwrites do not alter the signature of those functions.
- Imports of other files in the user's code is allowed. The ghost status of elements is properly carried over as well. However, the Ghost Check and Extraction phases are only executed on the main file that is

being verified. As such, it is possible to include invalid ghost code in our analysis if the user does not verify some code before importing it. Nevertheless, if this should happen unintentionally, the user would notice once they try to extract their import.

Chapter 4

Evaluation

Originally, one of our goals was to support the Python subset that is supported by Nagini. To evaluate this, we have taken Nagini's test suite [10], filled with translation and verification tests, and checked our implementation's performance. In total, we have run 555 tests and roughly 77.5% of those have succeeded. As such, we can reasonably say that most of Nagini's Python subset is supported by our implementation.

For the rest of this chapter, we will take a closer look at some of Nagini's functional verification examples and analyze the behavior of our implementation when running the verification process and the extractor on these practical examples. For those interested, the file name of each example is always given in the caption of the respective figure.

To begin, Figure 4.1 shows a standard implementation of the `quickSort` algorithm. The program annotations describe that we must have the permissions to read the input array, denoted by the `Acc` and `list_pred` contracts, and that the algorithm must terminate, given by the `MustTerminate` contract. Despite the new checks we have introduced, our implementation manages to successfully verify this code.

If we run the extraction on this code, we receive the code shown in Figure 4.2. As we can see, all contract functions are removed and we get an executable program. However, this example is relatively simple because it does not contain any ghost elements beyond the contract functions.

Thus, we consider for our next example. Due to its original size, we will only show a number of excerpts presented in Figures 4.3 to 4.5 for ease of viewing.

Figure 4.3 contains the definition of a simple tree node data structure, where each node stores a key, its payload and references to its "neighboring" nodes. This class is then used in the code of Figure 4.4 to build a binary search

```
from typing import List
from nagini_contracts.contracts import *
from nagini_contracts.obligations import MustTerminate

def quickSort(arr: List[int]) -> List[int]:
    Requires(Acc(list_pred(arr), 2/3))
    Requires(MustTerminate(2 + len(arr)))
    Ensures(Acc(list_pred(arr), 2/3))
    Ensures(Implies(len(arr) > 1, list_pred(Result())))
    Ensures(Implies(len(arr) <= 1, Result() is arr))
    less = []  # type: List[int]
    pivotList = []  # type: List[int]
    more = []  # type: List[int]
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            Invariant(list_pred(less) and list_pred(pivotList)
                      and list_pred(more))
            Invariant(len(Previous(i)) == len(less) + len(more)
                      + len(pivotList))
            Invariant(Implies(len(Previous(i)) > 0,
                              len(pivotList) > 0))
            Invariant(Acc(list_pred(arr), 1/2) and len(arr) > 0
                      and arr[0] == pivot)
            Invariant(MustTerminate(len(arr) - len(Previous(i))))
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

Figure 4.1: `rosetta_qsort.py` — Example of a standard `quickSort` implementation, verified in Nagini.

```python
from typing import List

def quickSort(arr: List[int]) -> List[int]:
    less = []
    pivotList = []
    more = []
    if (len(arr) <= 1):
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if (i < pivot):
                less.append(i)
            elif (i > pivot):
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return ((less + pivotList) + more)
```

Figure 4.2: Returned code by extraction if given `rosetta_qsort.py`.

tree (BST). As a refresher, a BST is a tree structure where the nodes are sorted in such a way that any left children of the node have a smaller key and any right children have a greater key.

Our last excerpt in Figure 4.5 defines multiple functions which describe what correct behavior for a BST is. Notably, a non-empty BST must be a tree and it must be sorted. A tree must be the parent of each of its children. A node is sorted if every left child's key is smaller and every right child's is bigger than its own, which must be given for each node in a BST.

If we attempt to verify this code, our implementation fails. Since the `sorted` function uses the `Implies` contract for its return value, we know the return must be ghost. However, this code makes no distinction between ghost and regular elements. As such, from the perspective of our implementation, this function says it will return a regular boolean but gives a ghost boolean, wherefore it rejects the code. Thus, this code is rejected because it does not follow the new constraint we enforce. Moreover, the `_get` function takes a `perm` parameter, which we may notice is only employed in contracts to handle object permissions. These are clear instances where previous Nagini verified code defines ghost elements which are not yet marked as such.

Once we properly mark these elements as ghost, verifying the code succeeds. In addition, its regular code can be extracted; the results from the excerpts can be seen in Figure 4.6. As one might expect, the functions defined in Figure 4.5 have been discarded, leaving only the data structures. Furthermore,

```
class TreeNode:
    def __init__(self, key: int, val: str,
                 left:'TreeNode'=None, right:'TreeNode'=None,
                 parent:'TreeNode'=None) -> None:
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
        Ensures(Acc(self.key) and self.key is key and
                Acc(self.payload) and self.payload is val and
                Acc(self.leftChild) and self.leftChild is left and
                Acc(self.rightChild) and self.rightChild is right and
                Acc(self.parent) and self.parent is parent)

    @Pure
    def hasLeftChild(self) -> Optional['TreeNode']:
        Requires(Acc(self.leftChild))
        return self.leftChild

    @Pure
    def hasRightChild(self) -> Optional['TreeNode']:
        Requires(Acc(self.rightChild))
        return self.rightChild

    @Pure
    def isRoot(self) -> bool:
        Requires(tree(self))
        return Unfolding(tree(self), not self.parent)
```

Figure 4.3: First excerpt of `iap_bst.py`: An annotated implementation of a tree node data structure with some of its functions.

the `Unfolding` contract function which was used in the `isRoot` function has been replaced with just its key argument.

However, our implementation is not without its flaws. In the example `cav_example.py`, which presents a straightforward ticketing system, we can find a function with the `@ContractOnly` decorator, as shown in Figure 4.7. As explained in Section 3.4, these functions are ignored when checking the ghost code, but not during the extraction. Consequently, our verification process succeeds while the extractor fails, highlighting a case where our limitations impact us on solving a practical problem.

Curiously, our last two examples, specifically `parkinson_recell.py` and `test_student_enroll_preds.py`, throw an exception upon verifying them. After consideration, we have determined the cause to be a pre-existing Nagini bug. Specifically, the way termination obligations are translated can, under

```
class BinarySearchTree:

    def __init__(self) -> None:
        self.root = None  # type: Optional[TreeNode]
        self.size = 0
        Fold(bst(self))
        Ensures(bst(self))

    def get(self, key: int) -> Optional[str]:
        Requires(Acc(bst(self)))
        Ensures(Acc(bst(self)))
        Unfold(bst(self))
        if self.root:
            res = self._get(key, self.root, 2)
            Fold(bst(self))
            return res
        else:
            Fold(bst(self))
            return None

    def _get(self, key: int, currentNode: Optional[TreeNode],
             perm: int) -> Optional[str]:
        Requires(perm > 0)
        Requires(Implies(currentNode is not None,
                         Acc(tree(currentNode), 1/perm)))
        Ensures(Implies(currentNode is not None,
                        Acc(tree(currentNode), 1/perm)))
        if not currentNode:
            return None
        Unfold(Acc(tree(currentNode), 1/perm))
        if currentNode.key == key:
            res = currentNode.payload
        elif key < currentNode.key:
            res = self._get(key, currentNode.leftChild, perm * 2)
        else:
            res = self._get(key, currentNode.rightChild, perm * 2)
        Fold(Acc(tree(currentNode), 1/perm))
        return res

    def __getitem__(self, key: int) -> Optional[str]:
        Requires(Acc(bst(self)))
        Ensures(Acc(bst(self)))
        return self.get(key)
```

Figure 4.4: Second excerpt of `iap_bst.py`: An annotated implementation of a binary search tree.

```
@Predicate
def tree(n : TreeNode) -> bool:
    return (Acc(n.key) and Acc(n.payload) and Acc(n.leftChild)
            and Acc(n.rightChild) and Acc(n.parent) and
            Implies(n.leftChild is not None, tree(n.leftChild) and
                    getParent(n.leftChild) is n) and
            Implies(n.rightChild is not None, tree(n.rightChild) and
                    getParent(n.rightChild) is n))

@Pure
def sorted(n: TreeNode, upper: Optional[int],
           lower: Optional[int]) -> bool:
    Requires(tree(n))
    Decreases(tree(n))
    return (Unfolding(tree(n),
            Implies(upper is not None, n.key < upper) and
            Implies(lower is not None, n.key > lower) and
            Implies(n.leftChild is not None,
                    sorted(n.leftChild, n.key, lower)) and
            Implies(n.rightChild is not None,
                    sorted(n.rightChild, upper, n.key))))

@Predicate
def bst(t: BinarySearchTree) -> bool:
    return (Acc(t.root) and Acc(t.size) and
            Implies(t.root is not None,
                    tree(t.root) and sorted(t.root, None, None)))
```

Figure 4.5: Last excerpt of `iap_bst.py`: Functions used to define correct behavior of a BST.

the wrong circumstances, lead to it generating invalid Viper code. Therefore, since our implementation adds termination requirements for ghost code, examples that would otherwise succeed will now crash. The extractor does complete successfully, but we will not cover it in more detail here, as its results are comparable to previous examples.

In summary, our implementation supports most of Nagini's Python subset and can handle practical examples. Furthermore, previous Nagini examples contain ghost elements that had been baked into the regular code. With our implementation, we can make these explicit and automatically remove them.

```
class TreeNode():
    def __init__(self, key: int, val: str,
                 left: 'TreeNode'=None, right: 'TreeNode'=None,
                 parent: 'TreeNode'=None) -> None:
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self) -> Optional['TreeNode']:
        return self.leftChild

    def hasRightChild(self) -> Optional['TreeNode']:
        return self.rightChild

    def isRoot(self) -> bool:
        return (not self.parent)

class BinarySearchTree():
    def __init__(self) -> None:
        self.root = None
        self.size = 0

    def get(self, key: int) -> Optional[str]:
        if self.root:
            res = self._get(key, self.root)
            return res
        else:
            return None

    def _get(self, key: int,
             currentNode: Optional[TreeNode]) -> Optional[str]:
        if (not currentNode):
            return None
        if (currentNode.key == key):
            res = currentNode.payload
        elif (key < currentNode.key):
            res = self._get(key, currentNode.leftChild)
        else:
            res = self._get(key, currentNode.rightChild)
        return res

    def __getitem__(self, key: int) -> Optional[str]:
        return self.get(key)
```

Figure 4.6: Part of the extracted code of the previous `iap_bst.py` code.

```
@ContractOnly
def get_seats(id: int, num: int) -> List[Tuple[int ,int]]:
    Requires(num > 0)
    Requires(MustTerminate(1))
    Ensures(list_pred(Result()))
    Ensures(len(Result()) == num)
    Exsures(SoldoutException, True)
```

Figure 4.7: Contract only function taken from `cav_example.py`.

Chapter 5

Conclusion

In this thesis, we have made users capable of marking ghost elements and have provided a type system to ensure they do not interfere with the regular code. Furthermore, we have added a tool to remove identified ghost elements to get an equivalent executable program. These extensions cover most of Nagini's Python subset and allow users to mark ghost elements that previously were embedded into the regular code.

This work allows users of Nagini to state their intentions for verification more clearly, aids in confirming that regular and ghost elements remain separate and eases the effort of turning the verified result into practically usable code while maintaining its strong guarantees. Together, this creates progress towards building robust tools for verifying Python code.

Future work could extend our system by giving more support for ghost functions that only argue over the permissions of objects. For instance, one might want to write a predicate which shows that holding permissions over two disjunct halves of a list is equivalent to holding permissions over the entire list. Furthermore, we could strengthen the extractor's capabilities to enable the resulting code to be printed to a file and to allow multiple files to be extracted at once.

Bibliography

[1] J. Lehtosalo et al., *Mypy - Optional Static Typing for Python*, Accessed: 09-02-2026. [Online]. Available: https://mypy-lang.org.

[2] M. Eilers and P. Müller, "Nagini: A Static Verifier for Python," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., Cham: Springer International Publishing, 2018, pp. 596–603, ISBN: 978-3-319-96145-3. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_33.

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62, ISBN: 978-3-662-49122-5. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2.

[4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[5] Microsoft, *Z3 - an efficient SMT solver*, Accessed: 09-02-2026. [Online]. Available: https://github.com/Z3Prover/z3.

[6] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "The Spirit of Ghost Code," *Formal Methods in System Design*, vol. 48, no. 3, pp. 152–174, Jun. 2016, Extended version of https://hal.inria.fr/hal-00873187. DOI: 10.1007/s10703-016-0243-x. [Online]. Available: https://hal.science/hal-01396864.

[7] R. Leino, *Dafny*, Accessed: 20-02-2026. [Online]. Available: https://dafny.org/dafny/.

[8] M. Parkinson and G. Bierman, "Separation logic and abstraction," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05, Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 247–258, ISBN: 158113830X. DOI: 10.1145/1040305.1040326. [Online]. Available: https://doi.org/10.1145/1040305.1040326.

[9] V. Astrauskas, P. Müller, and M. Eilers, "Input-Output Verification in Viper," M.S. thesis, ETH Zürich, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:251556956.

[10] M. Eilers, *Nagini's Test Suite*, Accessed: 27-02-2026. [Online]. Available: https://github.com/marcoeilers/nagini/tree/master/tests.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden zwei Optionen ist **in Absprache mit der verantwortlichen Betreuungsperson** verbindlich auszuwählen:

☒ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.

☐ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe. Dabei habe ich nur die erlaubten Hilfsmittel verwendet, darunter sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson sowie Technologien der generativen künstlichen Intelligenz. Deren Einsatz und Kennzeichnung ist mit der Betreuungsperson abgesprochen.

Titel der Arbeit:

Enabling Ghost Code in the Nagini Verifier

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Principe	Marco Colin

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Freienstein, 27.02.2026	M. Principe

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] Für weitere Informationen konsultieren Sie bitte die Webseiten der ETH Zürich, bspw. https://ethz.ch/de/die-eth-zuerich/lehre/ai-in-education.html und https://library.ethz.ch/forschen-und-publizieren/Wissenschaftliches-Schreiben-an-der-ETH-Zuerich.html (Änderungen vorbehalten).