ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verification of Practical Go Programs

Bachelor Thesis

Luca Halm

September 01, 2021

Advisors: Prof. Dr. Peter Müller, Felix Wolf, João C. Pereira

Department of Computer Science, ETH Zürich

Abstract

Gobra is a verifier based on the Viper verification framework, which statically verifies Go code. In this project, we use Gobra to verify part of the implementation of the SCION protocol.

SCION is a new internet architecture designed to provide secure routing and forwarding. We use Gobra to verify a part of SCION's border router, which is responsible for forwarding packets.

Verifying such an extensive project poses some challenges, such as scalability issues of the verifier and high annotation overhead. Furthermore, it exposes idioms and features of the programming language that may not be supported by the verifier.

In this work, we identify several shortcomings of Gobra and present a list of verification techniques that decrease verification time. Furthermore, we extend Gobra with three new features to decrease specification overhead and increase performance.

Contents

Chapter 1

Introduction

Go [1] is a compiled, concurrent, statically typed programming language developed by Google. It was designed to simplify the development of software for multiprocessor and network systems.

Even when tested, Go programs can suffer from various kinds of bugs that jeopardize memory safety and functional correctness. Software verification can help in reducing the risk of unexpected software faults by proving that the program behaves according to some formal specification.

Gobra [2] is a modular verifier for Go programs. It verifies Go code that is annotated with preconditions, which must hold before the method gets invoked, and postconditions, which must hold after the method execution. Gobra performs verification by encoding the annotated source code to a program in the Viper [3] intermediate verification language which gets processed by the Viper toolchain.

SCION [4] is a new internet architecture designed and developed at ETH Zürich. SCION was developed to provide secure routing and forwarding, alongside a number of other desirable properties. The reference implementation[1] of the SCION protocol is written in the Go programming language.

VerifiedSCION is a research project which aims to verify SCION from the high-level protocol design all the way down to the implementation. The focus is on verifying safety, functional correctness, and security of the implementation of the SCION routers.

In this project, we verified memory safety for a crucial part of the SCION border router using Gobra in the context of the VerifiedSCION project. For verification, we specified part of the SCION source code and used Gobra to prove that the code adheres to the specification. During our work on verification, we also identified some shortcomings of Gobra and implemented

[1]https://github.com/scionproto/scion

three new features to decrease specification overhead and verification runtime. The first feature introduces a new `preserves` keyword, which is used to specify that an assertion is both a precondition and a postcondition. The second feature, the `outline` keyword, allows a sequence of code to automatically be outlined by Gobra. Finally, the third feature introduces a new processing step that splits the produced Viper program into multiple smaller ones.

In Sec. 2, we present some concepts and tools that are important for this thesis. Sec. 3 introduces the relevant parts of the SCION codebase and describes how we approached the verification process. Sec. 4 then presents the three features we added to Gobra. In Sec. 5, we present three examples from our work. Finally, Sec. 6 discusses our results and presents statistics.

Chapter 2

Preliminaries

2.1 SCION

SCION [4] is a new internet architecture that aims at overcoming the routing problems of the BGP protocol, while increasing the resilience against DDoS attacks. In today's internet, each router decides how to forward packets based on a table of IP prefixes it maintains. Therefore, the internet service providers (ISPs), which control the prefix tables of their routers, have full control over which route their clients' data takes.

Unlike current solutions, SCION packets specify not only the intended destination of the packet but the full path of the packet through the network. A path consists of a list of hops that the packet has to traverse to get to the destination. Every such hop field contained in the packet header corresponds to an autonomous system (AS). Therefore, each client has full control over what path the packets should take. The header also contains info fields, which store metadata for each path segment, e.g. a timestamp indicating when the path segment expires.

Additionally, since the sender is responsible for picking the paths for its packets, there is no need for long inter-domain routing tables and complex longest IP prefix matching. This means forwarding packets is more efficient. The router simply has to look at the next hop in the path, which it finds in the header of the packet, and send it there.

The SCION architecture can be separated into a control plane and a data plane. The task of the control plane is to discover paths and to make them available to end hosts, essentially providing end-to-end paths through the network. The data plane uses these paths to forward packets to the desired destination.

This project will focus on verifying parts of the implementation of the border router, which implements the packet forwarding logic of the dataplane. In particular, the code that we verify reads an incoming packet, checks whether the header is valid, and extracts the next hop field from the header.

2.2 The Gobra Ecosystem

Gobra [2] is a modular verifier for Go programs and includes support for Go's built-in concurrency primitives, most prominently, channels and goroutines, but also mutexes and wait-groups. Gobra is a frontend for Viper and encodes annotated Go code to the Viper intermediate language, which is then verified by the Viper backend.

If verification terminates successfully, Gobra shows a corresponding message. Otherwise, if Viper returns a verification error, Gobra translates it back to the level of Go code, such that the user knows on what line a problem has occurred.

Gobra's annotation consists of preconditions, postconditions, and loop invariants. Preconditions and postconditions must be written before the method definition and make up the method's contract. Loop invariants are added before loop statements.

Listing 2.1 shows an example of a Go function with annotations. The precondition specifies that tha parameter `x` needs to be greater or equal to zero when the method is called. The postcondition guarantees that the returned value is equal to `2 * x`. For verification to succeed, we add two loop invariants. These specify that `res` stays smaller or equal than `2 * x` and that `res` is equal to `2 * k`.

```
requires x >= 0
ensures result == 2 * x
func foo(x int) (result int) {
  k := 0
  res := 0
  invariant res == 2 * k
  invariant res <= 2 * x
  for k < x {
    res = res + 2
    k = k + 1
  }
  return res
}
```

Listing 2.1: Example of Go code with annotations.

Gobra uses modular verification. This means that each method is verified independently from other methods. Therefore, for each method, Gobra assumes the preconditions and tries to prove that the postconditions hold after execution of the method body. If the method calls other methods, Gobra simply assumes that their contracts hold.

2.2.1 Permission-Based Verification

Permission-based verification is used to verify memory safety. If a method tries to read or modify a memory location, it requires permission to that location. There are two types of permissions, read permission and write permission. Multiple methods can hold read permission to a memory location at the same time, as long as no other method holds write permission. If a method holds write permission, no other method can hold read or write permission to that location.

Gobra offers fractional permissions. To specify write access to a memory location one can use `acc(x.f)`, which is shorthand for `acc(x.f, 1/1)`. To state read permission, any fraction that is less than one can be used, e.g. `acc(x.f, 1/10)`. Permission assertions are used in preconditions and postconditions to state that a method requires permission to a given memory location or that it returns permission to it.

The example in Listing 2.2 shows a function `foo` which takes a pointer to an `int` as an argument. The preconditions, on Line 1 and Line 2, specify 1/3rd permission to `x` and that the value of `x` is `3`. The method first calls `bar`, which is defined on Line 11. `bar` requires and returns 1/6th permission to `x`. Therefore, on Line 4, `foo` passes 1/6th permission to `bar` and keeps the other 1/6th. Since `foo` keeps a fraction of the permission, Gobra can verify that the value is not changed by `bar`. Because of that, the assertion on Line 5 verifies. `baz` on the other hand, which is called on Line 6, requires 1/3rd permission to `x`. Therefore, `foo` passes every permission to `baz`. Since `foo` does not keep any permission to x, Gobra cannot verify that x is not changed and the assertion on Line 7 fails to verify.

```
1  requires acc(x, 1/3)
2  requires *x == 3
3  func foo(x *int) {
4    bar(x)
5    assert *x == 3 // verifies
6    baz(x)
7    assert *x == 3 // fails
8  }

9  requires acc(x, 1/6)
10 ensures acc(x, 1/6)
11 func bar(x * int)

12 requires acc(x, 1/3)
13 ensures acc(x, 1/3)
14 func baz(x *int)
```

Listing 2.2: Example of fractional permissions.

2.2.2 Predicates

Predicates allow assertions to be parameterized and named. They can have any number of parameters and can be recursive. In Listing 2.3, a predicate `sliceAcc` is defined which parameterizes access permission to all elements in a slice of integers.

```
pred sliceAcc(slice []int) {
    forall i int :: 0 <= i && i < len(slice) ==> acc(slice[i])
}
```

Listing 2.3: A predicate that asserts write permission to all elements in a slice.

As seen in Listing 2.4, predicates can also be used as preconditions and postconditions. In Gobra, a predicate instance and its body are not the same. To use the assertion that is contained in the predicate, it first has to be unfolded. This replaces the predicate instance with the actual assertion. Analogously, when a predicate is folded the assertion is replaced by an instance of the predicate, which in our example means that the method passes the permissions to the elements in the slice to the predicate, thereby causing the assertion on Line 13 to fail.

```
1   requires sliceAcc(slice)
2   ensures sliceAcc(slice)
3   func foo(slice []int) {
4       // this assertion fails, predicate needs to be unfolded first
5       assert forall i int :: 0 <= i && i < len(slice) ==> acc(slice[i])

6       // to get permission to the slice elements we need to unfold
7       unfold sliceAcc(slice)

8       // now the assertion holds
9       assert forall i int :: 0 <= i && i < len(slice) ==> acc(slice[i])

10      // to satisfy the postcondition we must fold it again
11      fold sliceAcc(slice)

12      // now the assertion no longer holds
13      assert forall i int :: 0 <= i && i < len(slice) ==> acc(slice[i])
14  }
```

Listing 2.4: Example of a predicate being undolfed and folded.

2.3 Viper

Viper [3], which stands for Verification Infrastructure for Permission-based Reasoning, is a language and collection of tools developed at ETH Zurich. The Viper infrastructure provides a simple, sequential, object-based, imperative programming language and two back-ends, one based on symbolic execution and one based on verification condition generation.

Viper is used to verify partial correctness of program statements, which means that verification guarantees that the properties specified at a program state hold if that program state is reached. For example, postconditions of a method are guaranteed to hold if a call to that method terminates.

A Viper program consists of five types of top-level declarations, which we call members of said Viper program:

- Fields have a name and a type. Each object has all fields. A field of an object can only be accessed if the method holds the required permission.
- Methods have input and output parameters, preconditions, postconditions, and optionally a body. Methods can modify the program state.
- Functions have input parameters, one return value, preconditions, and postconditions. They can have a body containing a single expression, which can read but not modify the program state.
- Predicates have input parameters and a body. They are typically used to abstract over assertions.
- Domains have a name and a body containing a number of domain functions followed by a number of domain axioms. They can contain type parameters and are useful to introduce custom types.

Chapter 3

Methodology

3.1 Verification Goals

There are two kinds of properties that we have verified:

- Memory Safety, i.e., no data races, no illegal heap memory accesses, no indices that are out of bounds when accessing an array
- Functional Specification, i.e., methods produce the desired result

The main focus was on verifying memory safety. In some cases, functional specification is needed to allow verification of memory safety. For example, if an index used to access an element in an array is computed during runtime, functional specification is required that states that the calculated index is always in bounds of the array.

In order to fix the scope of the project, we selected a set of methods to verify and formulated a set of assumptions. Namely, we assumed that the SCION crossover feature is not used and that the incoming packets are well-formed and correctly parsed. In addition, we assumed that functionality provided by third-party libraries is correctly implemented.

3.2 Codebase

The SCION codebase[1] is structured into several packages and currently depends on over 50 third-party libraries. We focused on the implementation of the router, which takes on a crucial role in the dataplane of the SCION architecture.

We worked on the `scionPacketProcessor.process` method which invokes over ten other methods. These methods are responsible for parsing the

[1] https://github.com/scionproto/scion

path of the packet, validating the packet length, validating different values in the packet header, and modifying the header. In this thesis, we managed to verify the first three methods called from `process` and their dependencies.

The first called method is `scionPacketProcessor.parsePath`. It is responsible for decoding the current hop field and info field from the header, validating them, and constructing a packet with error information if a problem occurs. The second method, `validatePktLen`, checks if the packet payload has the expected length. `updateNonConsDirIngressSegID`, which is the third method called by `process`, updates the value of the info field and stores it in the packet header.

Fig. 3.1 shows the call graph of the `parsePath` method. It directly calls four methods. `GetCurrentHopField` and `GetCurrentInfoField` decode the hop field and info field from the packet header. Then, `validateHopExpiry` and `validateIngressID` use the decoded hop field and info field to validate the timestamp and ingress ID of the packet.

The call graph contains all methods that we specified and verified. In addition, we specified 22 stubs, which are method signatures annotated with the corresponding specification but without the body, for methods in the SCION codebase that were out of scope to verify.

The router implementation heavily relies on the gopackets library[2], which provides packet decoding functionality. In order to use the methods and data types from the library, we specified 8 interfaces and 26 method stubs.

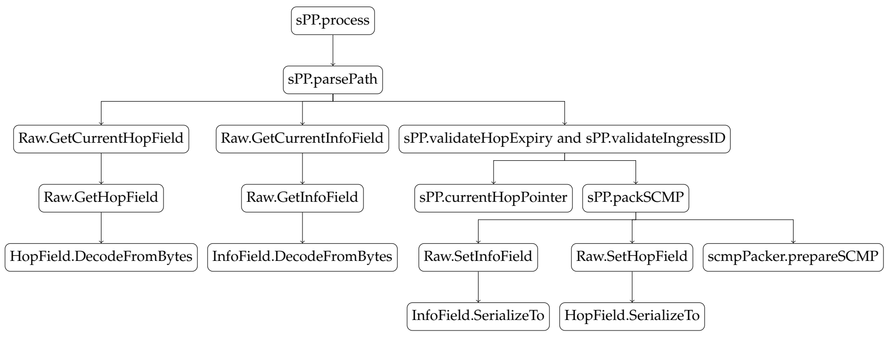

Figure 3.1: Call graph of `scionPacketProcessor.process`, where sPP stands for `scionPacketProcessor`

3.3 Verification Process

The goal of the verification is to prove memory safety of the `parsePath` method. We mostly focused on this task, as some issues with the verifier

[2]https://github.com/google/gopacket

arose, e.g. lack of support for standard library methods and missing support for global variables. One big problem was the performance of the verifier, this slowed down work since it could take a long time for the verifier to complete one verification run.

3.3.1 Bottom-Up Approach

We decided to work in a bottom-up manner because when verifying one method, it makes sense to know the preconditions of the methods that are called. Otherwise, many changes may have to be performed whenever the specification of a called method is added or edited. In addition, a method can only be considered verified when the contract of all called methods are fixed and they all successfully verify themselves. We, therefore, consider it most efficient to first verify all dependencies of a method before writing the contract for it.

One disadvantage of the bottom-up approach is that, when verifying a method, we might pick a postcondition that is too weak. This occurs if the caller requires more guarantees after the function call than the verified method returns in its postconditions. A simple example is a method requiring permission to some memory location but not returning it in the postconditions. Such a mistake is only discovered later when the caller is verified.

Nonetheless, we started verifying the leave nodes in our dependency graph. A central part of verifying memory safety is specifying which access permissions every method requires to perform its actions. In most cases, methods require read access to some of the arguments and write access to others. For arguments of type array, methods usually require access to a subset or all elements in that array. In those cases, access permissions can be specified using quantified permission, as shown in Listing 3.1. The method has two preconditions. The first one specifies write permission to a field. The second one uses a forall quantifier to state that the method requires write permission to elements in the array with an index greater or equal to 0 and less than the length of the array. This means the method requires access to all elements in the array. The two postconditions make sure that the permissions are given back to the caller.

```
requires acc(&s.BaseEmbedded.PathMeta)
requires forall i int :: 0 <= i && i < len(b) ==> acc(&b[i])
ensures acc(&s.BaseEmbedded.PathMeta)
ensures forall i int :: 0 <= i && i < len(b) ==> acc(&b[i])
func (s *Raw) SerializeTo(b []byte) error
```

Listing 3.1: Example of quantified permission

3.3.2 Handling Predicates

Predicates allow an assertion to be named and parameterized. This is a useful technique to (1) group multiple assertions, (2) decrease the number of preconditions and postconditions by replacing multiple assertions with one predicate, (3) formulate invariants that need to hold at multiple locations, and (4) decrease the verification overhead.

Even though we used predicates to handle invariants, they were most helpful in decreasing verification overhead and therefore improving verification runtime. Assertions that rely on quantified permissions, e.g. to formulate access permissions to all structs in an array, can significantly increase the verification burden. Especially in cases where quantified assertions are needed by a method only because a called method requires them, we observed that abstracting the quantified permissions via a predicate often lead to shorter verification times. We believe that this is because the verifier then performs fewer quantifier instantiations during verification. In that situation, we simply put the assertion into its own predicate and use the predicate instance instead. This gives us much more control over when the verifier is supposed to consider the actual assertion and when it is just supposed to pass the predicate around.

Predicates are also heavily used when it comes to specifying methods of an interface. In that case, we introduced predicates to abstract over the permissions required for the methods of that interface. Then, for each structure implementing that interface, we instantiated the predicate with the concrete assertions required by the method implementations.

3.3.3 Outlining

Large and complex methods can quickly cause long verification times. Especially if multiple predicates need to be unfolded or folded, there are many loops, or there are many array accesses. This did happen a few times during our work because the code was not written with verification in mind. To decrease complexity and runtime in such cases, we split up the method body into multiple methods, essentially splitting up the complex problem into smaller units, without changing the semantics of the program. By specifying preconditions and postconditions for each smaller unit we can manage the scope of complex assertions and avoid accumulation of assertions that are no longer needed, thereby decreasing the overall verification burden.

We used outlining when verifying the `scmpPacker.prepareSCMP` method, splitting it up into a total of seven methods, each with its preconditions and postconditions. This made both complexity and verification time more managable.

3.4 Library Specification

The router implementation heavily relies on the `gopacket` library[3]. To verify the code that makes use of this library, we added specification to all interfaces and methods that are used by the border router implementation. However, verifying the actual implementation of the library is out of the scope of the project, and as such we assume it is correct.

For example, Listing 3.2 shows the `SerializableLayer` interface defined in the `gopacket` library which we added to `gopacket/writer.gobra`. On Line 2, we declare an abstract predicate `Mem`. This predicate must be defined by all structures that implement the interface and should assert access to all memory locations it uses.

The interface contains two methods. From looking at implementations of the `LayerType` function, we know that it is pure and returns a global constant, therefore, the method does not require any memory permissions.

For the `SerializeTo` method, we do not know the exact implementations, so we added over-approximated specification. This is why, on Line 3 and Line 5, the method requires and returns full access to all memory locations specified in the `Mem` predicate. Furthermore, the method takes an argument `b` of type `SerializeBuffer`, which is also an interface defined by the library with a `Mem` predicate. On Line 4 and Line 6 we require and return permission to the memory of the `SerializeBuffer`.

The second argument, which is of type `SerializeOptions`, is a structure defined in the library that contains two boolean fields. Both fields are on the stack and not on the heap. Therefore we can access them without specifying memory permissions.

```
1 type SerializableLayer interface {
2     pred Mem()

3     requires Mem()
4     requires b.Mem()
5     ensures Mem()
6     ensures b.Mem()
7     SerializeTo(b SerializeBuffer, opts SerializeOptions) error

8     pure LayerType() LayerType
9 }
```

Listing 3.2: SerializableLayer interface defined in writer.gobra in the gopackets library

[3] https://github.com/google/gopacket

Chapter 4

Added Features

During our work on VerifiedSCION, we used Gobra to verify complicated and large pieces of code. As expected, we faced several minor and a few major issues. For example, we noticed that many methods have assertions that are used both as a precondition and as a postcondition, which lead us to introduce a new keyword to decrease specification overhead in cases where an assertion is supposed to hold before and after a method execution (Sec. 4.1).

One major problem was performance. To decrease verification runtime, we manually moved some pieces of code into new methods with their own specification, which we call outlining. To simplify outlining, we decided to introduce another keyword that can be used to tell Gobra to move a piece of code with a given contract to a new method (Sec. 4.2).

Additionally, we realised that the Viper programs created by Gobra can be very long. We added a feature that creates multiple smaller Viper programs for verification rather than one large program. These smaller programs can then be verified in parallel (Sec. 4.3).

4.1 Specification Keywords

Memory safety is a desired property for all programs. In Gobra, memory safety must be guaranteed before functional specification can be added. Verifying memory safety mostly consists of arguing about permissions to heap memory locations to prove the absence of data races, illegal heap memory accesses, array accesses that are out of bounds, etc. Most methods in the data plane require access to some memory locations, use them in the body, and finally give the permissions back to the caller. This is a common pattern, methods that modify the program state require access to heap locations and return the permissions after execution for further use. Therefore, most method contracts contain assertions specifying the same access permissions both as preconditions and as postconditions. This results in many duplicate lines and copy-pasting of assertions.

To avoid this, we introduced a new keyword that can be used in method specification to state that an assertion is both a precondition and a postcondition. The new keyword decreases specification overhead. It allows one assertion to be used to form a contract rather than a precondition and an identical postcondition, increasing readability and maintainability.

The design is straightforward: Gobra treats the new `preserves` keyword as syntactic sugar for a precondition and a postcondition with the same assertion, translating it into a precondition and a postcondition in the Viper program.

For methods that contain loops, most of those assertions also have to be included as invariants. We also considered a design that includes a `preserves_everywhere` keyword that extends the `preserves` keyword by also including the assertions as loop invariants for all loops in the method body, similar to `context_everywhere` in VerCors [5].

However, this introduces a tricky problem for nested loops if the assertion states read permission to a memory location. Since `preserves_everywhere` adds the same invariant to the inner loop as the outer loop, the outer loop has to give all permission away to the inner loop and can therefore not guarantee that the memory location was not modified. Therefore, we decided against this design.

A future design could allow the user to give names to assertions in the method specification. Then, the name of the assertion can be used in loop invariants to decrease the permission amount. This way, for each nested loop the permission amount can be modified without copy-pasting the assertion multiple times.

4.2 Outline

Large and complex methods can be tricky to verify. It can take a long time to verify a method which contains multiple loops, calls several other methods, uses many heap memory accesses, or requires folding and unfolding of predicates. In such a case, the verifier has to keep track of many memory locations, permissions, and assertions to create many complex reasoning steps for the verification to succeed.

As discussed in Sec. **??**, it can help to manually split these methods up into smaller ones. This can reduce the number of assertions and permissions the verifier has to keep track of, since each smaller method is verified independently.

The problem is that manual outlining requires the code to be rewritten as part of the verification process. This can introduce errors by accidentally changing the semantics of the program, which we want to avoid.

To simplify outlining of code, we introduce a new keyword that can be used to tell Gobra to automatically outline a sequence of statements into a function with a given contract. This allows code to be outlined without manual modification of the codebase, which eliminates the risk of accidental changes to the semantics of the program.

4.2.1 Design

We introduce a new `outline` keyword which is followed by a sequence of statements in parentheses. Gobra will move these statements into a new method. Like all methods, the created method needs a contract that specifies the required preconditions and the provided postconditions. The preconditions and postconditions for the new method can be written before the `outline` keyword.

```
func main() {
    x := 1
    y := 4

    requires y == 4
    ensures y == 7
    outline (
        y += 3
        z := x * 2
    )
    sum := x + y + z
}

```

Listing 4.1: Example usage of the outline feature

```
func main() {
    x := 1
    y := 4
    y, z = outlined(x, y)
    sum := x + y + z
}

requires y == 4
ensures y == 7
func outlined(x, y int)(int, int) {
    y += 3
    z := x * 2
    return y, z
}

```

Listing 4.2: Example after Gobra-to-Gobra transformation

This allows outlining of code without manually changing the program. Gobra automatically infers the arguments and return values of the new method from the statements in the outlined block. The current implementation has some limitations:

- Exceptional controlflow, namely `return`, `break`, `continue`, and `goto` statements in the outlined code are not supported
- The specification cannot contain `old` expressions
- Usage of `old` expressions in the outlined body is not supported
- Usage of the `&` operator to take the address of a variable is not allowed

The first limitation exists because a `return` statement in outlined code causes the created method to return rather than the method that contains the outline keyword, which is wrong behaviour. Additionally, Gobra infers which values to return from the outlined code and inserts `return` statements to do so. If the code was allowed to include `return` statements they would interfere with the ones added by Gobra.

A future implementation could support `return` statements by implementing a mechanism to inform the original method that the outlined code returned, e.g. by adding an additional return value that states whether the encountered `return` statement was added by Gobra or was already included in the original code.

The second limitation stems from the fact that the `old` expression causes heap dependent expressions to be evaluated in the initial state of the corresponding method. Variables are not heap-dependent, therefore the `old` expression has no effect on them. The third limitation is caused by the context difference of the method containing the outline block and the created method. An `old` expression in the outline block that should be evaluated in a state before the outline block does not work, because once the outlined code is moved to a method, the referenced context no longer exists.

Future designs could include a new keyword that acts like the `old` keyword on variables rather than heap-dependent expressions.

The fourth limitation is due to the way the feature is implemented. Currently, the variables defined outside the outlined code block and used inside are passed to the created method as arguments. These arguments are treated as exclusive by Gobra, meaning that it is not allowed to take the address of these values. This could also be supported by a future design, e.g. by always passing the pointer to functions and dereferencing whenever the value is used.

4.2.2 Encoding

The outline feature was implemented as a Gobra-to-Gobra-transformation. This means that the function is created before the code is translated into the Viper intermediate language. In the current design, the user does not state what parameters the outlined function takes or what it returns. That information is inferred from the body of the outlined function.

Listing 4.1 shows an example usage of the outline keyword. Listing 4.2 presents the same code after the Gobra-to-Gobra translation. The statements in the outline block are moved to a new method. The specification written before the outline keyword is moved to the new method. Gobra automatically identifies which variables must be passed to the function and which values must be returned. In Listing 4.2 on Line 4, the call to the new method is inserted by Gobra. The variables `x` and `y` are passed as arguments, `y` and `z` are returned. On Line 12 is the return statement, which is also inserted by Gobra.

To identify all variables that need to be passed to the outlined function as arguments, we first scan through the outlined code and find all variable usages. Of those variables, we filter out the variables that have been defined inside the outlined block. All variables defined before and used in the outlined code must be passed to the method as arguments. For example, in Listing 4.1, `x` and `y` must be passed to the function, because they are defined outside the outline block on Line 2 and Line 3, and used in the outlined code on Line 8 and Line 7, respectively.

Deciding what values to return requires more code analysis. All variables defined in the outlined code and used outside of the outlined block have to be returned. To find these, we search for statements that declare new variables in the outlined code, e.g. `z` in Listing 4.1 has to be returned, because it is defined in the outline block on Line 8. However, all variables that are defined before and modified in the outlined sequence must be returned as well. In our example in Listing 4.1, `y` is returned because it is modified in the outline block, on Line 7. We therefore identify for each variable that is passed as an argument if it is ever modified in the outlined code, and if so, return it.

Once we know all variables that need to be passed to the function and all variables that need to be returned, we can create the new method, add the specification provided before the outline keyword, move the code into the method and add a call to the method where the outline keyword was. Gobra does all of that before the code gets translated to the Viper intermediate language.

4.3 Chopping

Gobra translates Go code and specification to a program in the Viper intermediate language, which is then verified by the Viper infrastructure. Such a Viper program can be big, resulting in long verification times. For example, when verifying `dataplane.gobra`, which contains about 600 lines, the resulting Viper program is over 8000 lines long.

The Viper program contains various kinds of members, including methods, functions, predicates, fields, and domains. Naturally, the program contains many members, e.g. domains, that are used only by a subset of other members. Therefore, we expect that the verifier often has to keep track of assertions, permissions, etc, that are only sporadically used. We experimented to identify whether it is beneficial to output not one, but multiple Viper programs that each only contain the members that are used.

This means that for a simple Go program that contains a few methods and a main function that uses all the other methods, Gobra still creates one single Viper program, because the main function is the only one that is not called by any other function.

However, when verifying a package that defines methods that can be used by a client, Gobra outputs a Viper program for each method that is called from outside the package.

Another advantage of creating multiple Viper programs, instead of one, is that verification of the smaller programs can happen in parallel, which we expect to decrease verification time.

4.3.1 Algorithm

The process of splitting up one Viper program into multiple ones consists of four steps: (1) finding all dependencies in the Viper program and constructing the corresponding dependency graph, (2) identifying the strongly connected components in said graph and the dependencies between them, (3) finding all root components with no incoming edges and all components it depends on, and (4) creating a Viper program for each such root component and its dependencies.

To construct the dependency graph for the Viper program we analyze the whole program and identify all members. For each such member, we add a vertex to the dependency graph and store the relationship between the vertex and the member in a map. Once the graph contains a vertex for each member, we find the dependencies between them. See Listing 4.3 for the corresponding pseudocode.

For each method or function, we check if the specification contains calls to functions, references to predicates, or uses domain types. We also search for domains used in argument types and return types. Furthermore, we scan the body for calls to other methods or functions, and identify references to predicates. Similarly, we check for each predicate which functions and domains it depends on, for each field, if the type is a domain, and for each domain, if it depends on other domains. All these dependencies are added to the dependency graph by adding an edge between the corresponding vertices.

```
def create_dependency_graph(viper_program):
    dependency_graph = Graph()
    vertex_member_map = Map()
    // add a vertex for each member in the program
    for (member in viper_program.all_members):
        // create a vertex
        vertex = new Vertex(member.name)
        // store the corresponding member in a map
        vertex_member_map.add(vertex, member)
        // add it to the graph
        dependency_graph.add_vertex(vertex)

    // add an edge for each dependency in the program
    for (member in viper_program.members):
        for (usage in find_usages(member)):
            dependency_graph.add_edge(member.name, usage.name)
    return dependency_graph, vertex_member_map
```

Listing 4.3: Creation of the dependency graph.

At this point, the dependency graph can contain many cycles. All vertices that appear in a cycle depend on each other and should therefore end up in the same Viper program. Such subgraphs where all nodes can reach each other are called strongly connected components.

Therefore, we can apply Tarjan's strongly connected component algorithm [6] to find all strongly connected components, as seen in Listing 4.4. The algorithm then outputs a directed acyclic graph in which the vertices represent strongly connected components, which contain one or more vertices from the initial dependency graph, and the edges represent the dependencies between the components.

```
def find_strongly_connected_components(dependency_graph):
    // run tarjan's algorithm to get a directed acyclic graph
    // containing all strongly connected components
    // and the dependencies between them
    sccDAG = tarjan_scc_algorithm(dependency_graph)
    return sccDAG
```

Listing 4.4: Usage of Tarjan's algorithm to obtain a directed acyclic graph of stronly connected components.

We then find all components that have no incoming edges in the acyclic graph, as shown in Listing 4.5. Components with no incoming edges contain the members that are not used by any other member from a different component. These are the root components for which we generate one Viper program each.

```
def find_root_components(sccDAG):
    roots = List()
    // find all components with no incoming edges
    for (component in sccDAG):
        if (indegree(component) == 0):
            roots.add(component)
    return roots
```

Listing 4.5: Identification of root components.

Once the root components are known, we have to find the dependencies for each such component. This means that for each root, we must find all other components in the acyclic graph that it depends on. We use depth-first search starting from each of the roots to find all components that are reachable from it and therefore are dependencies of it. This leaves us with an array of root components together with their dependencies, which can be seen in Listing 4.6

```
def find_root_dependencies(sccDAG, roots):
    subtrees = List()
    // for each root, run depth first search
    // to find all reachable components
    for (root in roots):
        dependencies = find_reachable_DFS(root, sccDAG)
        // note that root is also contained in dependencies
        subtrees.add(dependencies)
    return subtrees
```

Listing 4.6: Identification of dependencies of each root component.

The last step is then to translate these roots and their dependencies into Viper programs, as in Listing 4.7. We iterate over the root components and the components it depends on to find all vertices of the initial dependency graph that need to be translated into Viper. Once all vertices are known, we use the map from the first step to look up the members that correspond to these vertices. Finally, we combine all these members into a Viper program.

```
def translate_to_Viper(subtrees, vertex_member_map):
    viper_programs = List()
    // iterate over all roots with their dependencies
    for (subtree in subtrees):
        viper_program = ViperProgram()
        // iterate over all components in that subtree
        for (component in subtree):
            // iterate over all vertices in that component
            for (vertex in component):
                // translate vertex back to member
                member = vertex_member_map.lookup(vertex)
                viper_program.add_member(member)
        viper_programs.add(viper_program)
    return viper_programs
```

Listing 4.7: Translation of subtrees to Viper programs.

When all four steps are implemented they can be called one after the other to complete the chopping algorithm, as shown in Listing 4.8

```
def chopper(viper_program):
    // dg stands for dependency_graph
    // vmm stands for vertex_member_map
    dg, vmm = create_dependency_graph(viper_program)
    sccDAG = find_strongly_connected_components(dg)
    roots = find_root_components(sccDAG)
    subtrees = find_root_dependencies(sccDAG, roots)
    programs = translate_to_Viper(subtrees, vmm)
    return programs
```

Listing 4.8: Full chopping algorithm.

4.3.2 Evaluation

One major goal of the chopping feature is to decrease verification time. In theory, since the different Viper programs that are created can be verified in parallel, verification runtime should significantly decrease with the new chopping feature. Unfortunately, after finishing our implementation, we realised that Silicon, the backend verifier that Gobra uses, does not support parallel execution. Therefore, we are forced to verify the Viper programs sequentially, which can introduce more overhead than the smaller program sizes can compensate. Until the implementation of Silicon is updated to support parallel execution, there is nothing we can do to reach the expected runtime decrease.

Table 4.1 shows the results of our benchmark tests. All tests were executed on a warmed-up JVM on a Microsoft Surface Pro 4 with a 2.2 GHz 4-Core Intel Core i7 CPU and 16 GB of RAM, running Windows 10 and OpenJDK 11. The test cases include well-known algorithms and datastructures. We also include test cases which contain specification errors and implementation errors. Each test case was executed five times, shown is the mean execution time in miliseconds. As expected, the version with the new chopping feature is still slower in some cases due to the lack of parallelism.

To get a rough estimate of how fast Gobra would be if parallel execution was supported, we verified `dataplane.gobra`, as specified in commit b762163[1], and collected the time spent on each Viper program to verify. The total time required to verify `dataplane.gobra` with the new feature was around 25 minutes, whereas verification without the chopping feature takes five minutes and thirty seconds. However, the longest verification time spent on a single Viper program created by the chopper is 94 seconds. Therefore, with a sufficient amount of cores, verification of `dataplane.gobra` takes only 94 seconds, instead of five and a half minutes, when verification of all Viper programs generated by the chopper is done in parallel.

[1]https://github.com/jcp19/VerifiedSCION/commit/b76216363fa5158d383bde218bad16e6a8c78ee5

Testcase	Chopped	Single	Difference
binary_search_tree	15942	21255	-5313
dense_sparse_matrix	23949	23075	874
dutchflag	3315	3049	266
heapsort	23424	27346	-3922
list_of_interfaces	3635	2246	1389
parallel_search_replace	85181	109221	-24040
parallel_sum	94227	140061	-45834
relaxed_prefix	12539	13182	-643
zune	2612	1655	957
impl_errors/binary_search_tree	17421	15037	2384
impl_errors/dense_sparse_matrix	22946	16845	6101
impl_errors/dutchflag	3211	2224	987
impl_errors/heapsort	24930	28985	-4055
impl_errors/list_of_interfaces	3704	1598	2106
impl_errors/parallel_search_replace	73821	69735	4086
impl_errors/parallel_sum	72640	74295	-1655
impl_errors/relaxed_prefix	4731	4314	417
impl_errors/zune	2151	1286	865
spec_errors/binary_search_tree	15720	17298	-1578
spec_errors/dense_sparse_matrix	25985	20876	5109
spec_errors/dutchflag	2583	1886	697
spec_errors/heapsort	24469	29099	-4630
spec_errors/list_of_interfaces	3584	1879	1705
spec_errors/parallel_search_replace	69275	95064	-25789
spec_errors/parallel_sum	86214	104541	-18327
spec_errors/relaxed_prefix	9352	10290	-938
spec_errors/zune	2612	1554	1058

Table 4.1: Results of the benchmark tests. Each testcase was executed five times both with and without the new feature. Shown are the mean execution times in miliseconds.

Chapter 5

Examples

In this section, we show and explain a few examples of specification that we encountered during our work. The first example shows specification of a method with postconditions that depend on the returned error value. In the second example, we show a method that only verifies with an assertion that we added. Finally, the third example shows an implementation proof.

5.1 Error-Dependent Guarantees

Listing 5.1 shows a method stub `ToDecoded` that is responsible for converting a `Raw` structure to a `Decoded` structure. We only specified the stub without verifying the body. Nonetheless, we know from the implementation that the method requires access to the slice of bytes stored in `s.Raw`, which we specify on Line 1 and Line 2. The interesting parts of this example are the two postconditions on Line 3 and Line 4, which guarantee access to the returned `decoded` value and access to its `Mem` predicate, if the returned error value is `nil`. This is a common pattern. Often when an error is returned, the method has not fulfilled its purpose, which in this case means that the `decoded` structure might not have been created.

```
1  preserves acc(&s.Raw)
2  preserves bytesAcc(s.Raw)
3  ensures err == nil ==> acc(decoded)
4  ensures err == nil ==> decoded.Mem()
5  func (s *Raw) ToDecoded() (decoded *Decoded, err error)
```

Listing 5.1: Method stub with guarantees depending on the error value.

5.2 Required Proof Aannotation

Listing 5.2 shows the relevant parts of the `GetInfoField` method, which decodes the info field specified by the index passed as an argument. On Line 8, `idx` is used to compute the index of the byte slice where the info field data starts. Then, on Line 10, a subslice is created with the bytes needed for decoding on Line 14. To guarantee that the calculated indices are in bounds of the slice, we added functional specification on Line 2 and Line 3. Line 4 specifies permission to all elements in the `s.Raw` byte slice.

`info.DecodeFromBytes`, called on Line 14, requires access to all elements in the slice passed as an argument, `subslice` in this case. Interestingly, without the assertion on Lines 11-13, the call does not verify, because Gobra cannot prove permission to all elements in the `subslice` slice.

We added the assertion on Lines 11-13 to help Gobra verify that all elements in `subslice` are also contained in `s.Raw`, to which access is guaranteed by unfolding `bytesAcc(s.Raw)`, and that the method has permissions to them. With that assertion the call to `info.DecodeFromBytes` verifies.

```
1   preserves acc(s)
2   preserves idx >= 0
3   preserves (MetaLen + idx*path.InfoLen)+path.InfoLen < len(s.Raw)
4   preserves bytesAcc(s.Raw)
5   ensures err == nil ==> infoField != nil && acc(infoField)
6   func (s *Raw) GetInfoField(idx int) (inf *InfoField, err error) {
7       unfold bytesAcc(s.Raw)
8       infOffset := MetaLen + idx*path.InfoLen
9       info := &path.InfoField{}
10      subslice := (s.Raw)[infOffset : infOffset+path.InfoLen]
11      assert forall i int :: 0 <= i && i <= len(subslice) ==>
12          &(s.Raw)[i + infOffset] == &subslice[i]
13          && acc(&subslice[i])
14      if err := info.DecodeFromBytes(subslice); err != nil {
15          fold bytesAcc(s.Raw)
16          return nil, err
17      }
18      fold bytesAcc(s.Raw)
19      return info, nil
20  }
```

Listing 5.2: Method that only verifies with an added assertion.

5.3 Implementation Proof

Listing 5.3 contain an implementation proof used by Gobra to verify that `Raw` implements the `Path` interface.

Lines 1-4 contains the definition of the `Raw` structure, which contains another structure of type `Base` and a slice of bytes. The `Path` interface is shown on Lines 5-13. We added an abstract `Mem` predicate and over-approximated specification to all methods.

Lines 14-18 define the `Mem` predicate for the `Raw` structure. It asserts permission to both fields and all elements in the bytes slice.

The implementation proof is shown on Lines 19-30. For `Raw` to implement `Path`, all preconditions of the interface methods must imply the preconditions of the implementations, and the postconditions of the implementations must imply the postconditions of the interface methods. This is why on Line 21 we unfold the `Mem` predicate required by the specification on Line 7, which then implies the preconditions of the `SerializeTo` implementation on Line 33. Then, by folding the `Mem` predicate on Line 23, the postconditions of the implementation imply the postconditions of the interface definition on Line 9. Therefore, the `SerializeTo` method of the `Raw` structure is a behavioural subtype of the `SerializeTo` method of the `Path` interface.

The reasoning for the `DecodeFromBytes` method is analogous. Therefore, Gobra can prove that `Raw` implements the `Path` interface.

```
 1  type Raw struct {
 2      BaseEmbedded Base
 3      Raw []byte
 4  }

 5  type Path interface {
 6      pred Mem()

 7      preserves Mem()
 8      preserves forall i int :: 0 <= i && i < len(b) ==> acc(&b[i])
 9      SerializeTo(b []byte) error

10      preserves Mem()
11      preserves forall i int :: 0 <= i && i < len(b) ==> acc(&b[i])
12      DecodeFromBytes(b []byte) error
13  }

14  pred (r *Raw) Mem() {
15      acc(&r.Raw) &&
16      acc(&r.BaseEmbedded) &&
17      bytesAcc(r.Raw)
18  }

19  (*Raw) implements slayers.Path {

20      (r *Raw) SerializeTo(b []byte) (err error) {
21          unfold r.Mem()
22          err = r.SerializeTo(b)
23          fold r.Mem()
24      }

25      (r *Raw) DecodeFromBytes(b []byte) (err error) {
26          unfold r.Mem()
27          err = r.DecodeFromBytes(b)
28          fold r.Mem()
29      }
30  }

31  preserves acc(&s.BaseEmbedded.PathMeta)
32  preserves forall i int :: 0 <= i && i < len(b) ==> acc(&b[i])
33  func (s *Raw) SerializeTo(b []byte) error

34  preserves acc(&s.Raw)
35  preserves acc(&s.BaseEmbedded)
36  preserves bytesAcc(s.Raw)
37  preserves forall i int :: 0 <= i && i < len(data) ==> acc(&data[i])
38  func (s *Raw) DecodeFromBytes(data []byte) error
```

Listing 5.3: Interesting part of the implementation proof for Raw and Path.

Chapter 6

Results

6.1 Verified Properties

During our work verifying the implementation of the SCION border router, we focused on verifying memory safety for the `process` method of the `scionPacketProcessor`. This requires all methods directly or indirectly called by the `parsePath` method to also be memory safe. Therefore, we started proving memory safety for methods at the bottom of the call graph and worked our way upwards. The router implementation also depends on methods in other packages of the SCION codebase. We added specifications without verifying the body for these methods, because they are not in the scope of this thesis. This allowed us to verify all methods in a bottom-up manner and finally prove memory safety for the `parsePath` method.

Two of the methods indirectly called by `parsePath` are responsible for extracting the current hop field and the current info field from the header of a SCION packet. The hop field and info field are contained in the header of the packet in a slice of bytes. To decode these values, these two methods calculate where the relevant data is encoded in the slice. Computation of these indices happens during runtime. Therefore, to verify memory safety we had to add functional specification to guarantee that these calculated indices are within the bounds of the slice.

The `GetInfoField` method shown in Listing 6.1 is one such method. The index is calculated on Line 8 and used on Line 10 to access the slice. On Lines 1-2, permissions to the slice of bytes and all elements in it are asserted as preconditions. These permissions are returned to the caller as postconditions on Lines 5-6. On Line 10, calculated indices are used to create a subslice. For this line to be memory safe, the lower index `infOffset`, which is calculated on Line 8 where both `MetaLen` and `path.InfoLen` are positive integer constants, must be greater or equal to zero. This is satisfied if the parameter `idx` is greater or equal to zero. Therefore, we add the precondition on Line 3 to guarantee this. Additionally, the upper index `infOffset+path.InfoLen` must be less than the length of the slice. To enforce this, we add the precondition on Line 4.

```
1  requires acc(&s.Raw)
2  requires forall i int :: 0 <= i && i < len(s.Raw) ==> acc(&(s.Raw)[i])
3  requires idx >= 0
4  requires MetaLen + idx*path.InfoLen+path.InfoLen < len(s.Raw)
5  ensures acc(&s.Raw)
6  ensures forall i int :: 0 <= i && i < len(s.Raw) ==> acc(&(s.Raw)[i])
7  func(s *Raw)GetInfoField(idx int) (infoField *InfoField, err error)
8      infOffset := MetaLen + idx*path.InfoLen
9      info := &InfoField{}
10     subslice := (s.Raw)[infOffset : infOffset+path.InfoLen]
11     if err := info.DecodeFromBytes(subslice); err != nil {
12         return nil, err
13     }
14     return info, nil
15 }
```

Listing 6.1: GetInfoField method with functional specification for index calculation.

6.2 Statistics

In total, we were able to successfully specify and verify memory safety for 27 methods, including the `parsePath` method. Furthermore, we specified 22 methods in different packages of the SCION project which get called by the data plane implementation but are out of scope to verify for this thesis. The router implementation also depends on the `gopacket` library[1], for which we had to add and specify eight interfaces and 26 method stubs.

Table 6.1 shows the number of methods verified, the number of method stubs specified, and the number of lines of specification added to each file.

6.3 Gobra

We used Gobra to verify memory safety of the `parsePath` method. The `parsePath` method plays an important role in the implementation of the SCION border router. It is the first method called from the `process` method, responsible for processing every scion packet that reaches the border router. Both these methods are implemented in `dataplane.gobra`. Therefore, we spent most of our time verifying the code in that file.

One major challenge during our work was verification performance. A few times when we started to verify a new method, Gobra did not terminate within two hours. We found that using predicates instead of quantified permission assertions and outlining for large methods are two helpful techniques to decrease verification runtime.

[1] https://github.com/google/gopacket

File	VM	SS	LoS
gopacket/base.gobra	0	4	20
gopacket/decoded.gobra	0	8	31
gopacket/layers/base.gobra	0	2	12
gopacket/layertype.gobra	0	2	6
gopacket/parser.gobra	0	2	14
gopacket/writer.gobra	0	8	22
lib/slayers/path/hopfield.gobra	2	0	10
lib/slayers/path/infofield.gobra	2	0	10
lib/slayers/path/scion/base.gobra	0	7	34
lib/slayers/path/scion/decoded.gobra	1	3	33
lib/slayers/path/scion/raw.gobra	7	4	129
lib/slayers/scion.gobra	7	7	106
lib/slayers/scmp.gobra	0	1	11
lib/slayers/scmp_msg.gobra	0	0	6
pkg/router/dataplane.gobra	8	0	348
Total	27	48	792

Table 6.1: Overview of the number of verified methods (VM), number of specified method stubs (SS), and number of lines of specification (LoS) per file.

In this section, we show how we used predicates and outlining to verify `dataplane.gobra`. By using these two techniques, verification time for `dataplane.gobra`, as specified in commit 8337018[2], was five and a half minutes. We show the effect of each technique by an example and by observing how verification time increases when the technique is not applied.

The first technique was using predicate instances instead of quantified permission assertions. For example, on line 1345 of `dataplane.gobra`, we use a predicate `scion.bytesAcc`, which can be seen in Listing 6.2, to state that the `updateSCIONLayer` method requires access to all elements in a slice of bytes. The predicate simply contains a forall quantifier specifying permission to all elements. If we replace the predicate instance with the actual assertion, verification no longer completes within two hours.

```
pred bytesAcc(data []byte) {
  forall i int :: 0 <= i && i < len(data) ==> acc(&data[i])
}
```

Listing 6.2: The predicate used to encapsulate quantified permission assertions.

[2]https://github.com/jcp19/VerifiedSCION/commit/8337018da441048f9bd64a94e745bde86ec76b14

We used the second technique when verifying the `prepareSCMP` method. The `prepareSCMP` method is over 70 lines long, calls more than 14 other methods, and requires unfolding and folding of two predicates. Before we applied outlining, verification of this method did not terminate within two hours. Since we did not know what part exactly caused the verification to take this long, we outlined seven blocks of code into their own methods, which we specified and verified in isolation.

However, even though outlining seemed to work in the beginning, we are still facing issues with the `prepareSCMPHelper5` method. We can verify the method and its contract if we remove the call to it from the `prepareSCMP` method body. Verification also succeeds if the call to `prepareSCMPHelper5` is added back and the body of it is commented out, without changing the specification. However, if both, the body of the `prepareSCMPHelper5` method and the call to it, are left in the code, verification no longer terminates within 2 hours. Unfortunately, we have no explanation for this odd behaviour.

Even though we managed to keep verification time low by using predicates and outlining, we still encountered a few statements that cause the verification to not terminate within two hours. Table 6.2 contains all such statements and where they occur. For these statements, neither using predicates nor outlining helped to keep verification runtime within two hours.

Statement causing long verification time	Line in dataplane.gobra
quote := make([]byte, quoteLen)	859
var scionLayer@ slayers.SCION	896
decoded := make([]gopacket.LayerType, 5)	909

Table 6.2: Statements that cause verification to not terminate within two hours.

Bibliography

[1] Google LLC. Go programming language. [Online]. Available: https://golang.org/

[2] ETH Zürich. Gobra. [Online]. Available: https://www.pm.inf.ethz.ch/research/gobra.html

[3] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. [Online]. Available: http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=MuellerSchwerhoffSummers16.pdf

[4] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat, *SCION: A Secure Internet Architecture*. Springer International Publishing AG, 2017.

[5] University Of Twente. The vercors verifier. [Online]. Available: https://vercors.ewi.utwente.nl/wiki#introduction

[6] R. Tarjan, "Depth first search and linear graph algorithms," *SIAM JOURNAL ON COMPUTING*, vol. 1, no. 2, 1972.

ETH

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 practical Go programs

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

Name(s):	**First name(s):**
Halm	Luca

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)**
01.09.2021	[illegible]

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