Static Verification of the SCION Router Implementation

Bachelor's Thesis

Sascha Forster

Supervised by Marco Eilers, Prof. Dr. Peter Müller
Department of Computer Science
ETH Zurich
Zurich, Switzerland

September 4, 2018

Abstract

The SCION project aims to improve routing in today's Internet. The SCION internet architecture consists of path servers, beacon servers, certificate servers, SIBRA servers and border routers. The border router is responsible for routing packets between different autonomous systems and has been implemented in Python. In this project, we verified memory safety, progress and I/O behaviour, for a part of the code, to guarantee that this implementation does not crash and that it correctly forwards packets. We used Nagini, a Python verifier based on the Viper verification framework, to statically verify the code.

We gathered various interesting statistics about the performance of the verifier while working on the project. Additionally, we measured the verification performance of the different backends that Nagini uses. Furthermore, we gained insight about what it means to use the Nagini frontend on a larger codebase instead of only small examples. We identified multiple code and specification patterns that impede verification performance and we describe how they can be avoided. One significant insight is that Nagini does not fulfil completeness when wildcard read permissions are used. Finally, we identified multiple bugs in Nagini, and as a result, most of them could be fixed.

Acknowledgements

I would like to thank my thesis supervisor Marco Eilers for his significant support during this project. Weekly meetings and regular discussions via E-Mail helped me overcome any difficulties that I encountered. I would also like to thank Prof. Dr. Peter Müller for the opportunity to work on this project.

Contents

1 Introduction

Routing is a central part of the Internet architecture as it allows information to be sent along its way from the source to the destination. There are, however, quite a few issues present in the architecture as it is used now. One big problem is the ability of malicious users to hijack traffic, which is flowing between two endpoints. This is exacerbated by the fact that a sender has no real control over which route their data takes.

The SCION [1] project aims to achieve significant improvements in this area. A major difference between SCION routing and traditional routing is the ability for the sender to choose the path that they want their data to follow. This is accomplished by saving the defined route in the data packet as a list of hop fields. Each hop corresponds to an autonomous system, e.g., an Internet service provider (ISP). SCION border routers can thus be stateless, as they do not need to keep a routing table and the routing information is encoded in the data packets.

As part of the SCION project an example implementation for a border router has been implemented in Python. The goal of the VerifiedSCION[1] project is to "verify the routing protocol SCION from the high-level design all the way down to the implementation" [2]. Verification can be performed to guarantee many different properties, but here we are interested in properties relevant to reliable operation of a router. As a first property, we chose memory safety, because we want to guarantee that the router does not crash due to a runtime error, e.g., an illegal access to heap memory. Then, we chose progress, such that we can ensure that the code does not get stuck in an infinite loop. Of course, the main method in a router should never terminate to ensure availability, but processing of a single packet should be guaranteed to terminate. The last relevant property is input/output (I/O) behaviour, because we want to show that the router correctly processes and forwards any valid packet it receives. Thus far, the router code has not been verified for these three properties.

In this project, we verified a part of the router implementation using a static verification tool. More specifically, we focused on the part of the code which checks if a received packet is well-formed, processes the packet accordingly and finally forwards it.

[1] http://www.pm.inf.ethz.ch/research/verifiedscion.html

1.1 Outline of Methodology

The verification of the code was performed in a static manner, meaning without executing the code itself. As the verification tool, we used the Nagini verifier [3], which is based on the Viper verification framework [4]. Nagini provides support for writing method contracts, which are used to verify the code. The router code is written in the Python programming language. We needed to add type annotations, because Nagini requires knowledge about types and Python is dynamically typed.

Since this is the first attempt to apply Nagini to a large, real-world codebase, an additional part of the project was to evaluate the performance and usability of Nagini. To gain statistics about performance, we measured the time needed to verify the methods. In the results section, we present statistics about the performance of Nagini. As a side task, while using the verifier, we identified bugs in Nagini.

Outline: In Section 2, some important concepts and tools, which are central to this report, are presented. If the reader is already familiar with those topics, it can be skipped. Section 3 describes how we approached this project and how we worked on it. Then, Section 4 continues by giving a detailed description of the verification of the code. With Section 5, we present the results of the project. In Section 6 we conclude this report and provide suggestions for future work.

1.2 Project Goals

The goals of the project were the following:

- Prove memory safety of the router by showing that no illegal memory accesses can happen.
- Prove absence of unintended infinite loops.
- Create an abstraction of the packet parsing mechanism of the router and declare a function that maps a SCION packet to the abstraction.
- Write an I/O specification [5] for the SCION router and verify that the implementation complies with it.

2 Preliminaries

This section serves to introduce the topics that are relevant to the project. If the reader is already familiar with these concepts, they can be skipped as they only contain general information.

2.1 SCION Routing

In traditional routing, a router keeps a table of prefixes according to which it forwards packets. This means that ISPs alone have the control over which routes clients' data takes. SCION [1] works differently, as it allows a client to choose their preferred route. To allow this, the route needs to be defined in every packet. It is stored as a sequence of different segments, which consist of a list of hop fields. Every hop field corresponds to an autonomous system (AS), e.g., an ISP. In a hop field, there is information about the ingress and egress interface of the AS. Essentially, these are the interfaces where the packet enters and leaves the AS.

SCION introduces a concept called isolation domains (ISD), which are groups of ASes. An ISD has a core, generally consisting of multiple ASes, called core ASes. ISDs are interconnected to provide Internet service to every AS and thus also all clients within an AS. The segments mentioned above are paths in an ISD or between multiple ISDs. There are three kinds of segments in SCION routing, up-, down- and core-segments. An up-segment is a list of hops from a non-core AS to the core. A down-segment is the opposite and a core segment is between core ASes. For this project, we focus on verifying parts of the code relevant to forwarding of a packet that only has one up-segment in its path.

2.2 Deductive Verification

Deductive verification is the concept of formally proving that a program adheres to a formal specification under all circumstances. A specification usually consists of contracts, which include preconditions, postconditions and loop invariants. These contracts are generally assertions and they describe the states that are allowed at a specific point in the program. Every method, which we want to verify, requires them, as they serve to specify the method behaviour. Preconditions need to hold true before the method is called in the program. Conversely, postconditions need to hold after the method is called.

To specify behaviour of loops in a method, we use loop invariants. Invariants have to be true before the loop executes, between every loop iteration and after the last iteration. If the loop terminates abnormally, e.g., via a break or return statement, the invariant does not need to hold, because the invariant only reasons about normal loop execution.

Evidently, the contracts must be side effect free, because they only serve to write a program specification. This is also necessary to keep the original program and its functionality intact.

In *modular* verification, each method is verified independently from the other methods. Still, there needs to be a specification, which allows for a successful verification, for all called methods. Modular verification allows for a much more efficient workflow, as it makes it easier to limit a verification to a subset of all methods of a codebase. However, a disadvantage is that all relevant information about a method's behaviour needs to be included in its specification.

More details about verification, which are specific to the verifier used for this project, follow in Section 2.6.

2.3 Viper

Viper[2] [4], which stands for Verification Infrastructure for Permission-based Reasoning, is a verification framework developed at ETH Zurich. Viper consists of an intermediate language, the Viper language, and two different backends, the symbolic execution backend and the verification condition generation backend. The latter encodes the code in Boogie[3] [6], which then generates a verification condition. In both toolchains, the output is then passed to the Z3[4] [7] SMT solver. By using Z3, Viper automates modular, deductive verification.

2.4 Nagini

Nagini[5] [3] is a frontend for Viper that encodes annotated Python code to the Viper language, so that it can then be verified using a Viper backend. Type

[2]https://bitbucket.org/viperproject
[3]https://github.com/boogie-org/boogie
[4]https://github.com/Z3Prover/z3
[5]https://github.com/marcoeilers/nagini

annotations are required by Nagini, as MyPy[6], a static type checker for Python, is run before the encoding. Furthermore, preconditions, postconditions and loop invariants need to be added to the original Python source code. In Nagini, we write pre- and postconditions before the method body and invariants before the loop body.

We explain general concepts in this and the following sections, but we will discuss more details about Nagini in relation to this project in Section 3.5.

An example of contracts in Nagini can be seen in Listing 1. The precondition specifies that argument `x` needs to be greater than 0 when the method is called. With the postcondition, we enforce that the returned value is equal to `2*x`. `Result()` is used to denote the returned value. To verify the postcondition, the verifier needs to know that `res` stays smaller or equal to `2*x` and to make the connection to `k`, we also need the condition `res == 2*k`.

```
1  def foo(x: int) -> int:
2      Requires(x >= 0)
3      Ensures(Result() == 2*x)
4      k = 0
5      res = 0
6      while k < x:
7          Invariant(res == 2*k)
8          Invariant(res <= 2*x)
9          res = res + 2
10         k = k + 1
11     return res
```

Listing 1: Example of Nagini contracts.

Nagini supports modular verification. Additionally, Nagini also has support for annotations to force a method to be contract-only, this means that the method does not get verified and its contracts are assumed to be correct. This is written as a `@ContractOnly` annotation. Furthermore, Nagini supports `Old` expressions, which are used to evaluate an expression in the state before the method was executed. They can be used in postconditions and in loop invariants.

[6]https://github.com/python/mypy

2.5 MyPy

Listing 2 shows an example of type annotations for MyPy. As can be seen in the code, every parameter that is not the receiver needs a type annotation. The example defines `x` to be of type `int` and `s` to be of type `str`, which is the string type in MyPy. Default values for parameters come after the type annotation. So, parameter `b` is of type `bool` and it has default value `True`. Additionally, a return type is also required, which in the example is `int`.

```
1 def foo(self, x: int, s: str, b: bool = True) -> int:
```

Listing 2: Example of MyPy type annotations in a method definition.

Some class constructors also need type annotations. In Listing 3, you can see that field `x`, which is initialised in the constructor with 0, does not need an annotation, as the type can be inferred by MyPy, which in this case is type `int`. If a field is initialised to `None`, we need a type annotation; otherwise MyPy will infer the `Any` type. This will not work, because Nagini requires every parameter and field to have a type which is not `Any`. An example of a type annotation in the constructor can be seen for the field `object`

```
1 def __init__(self) -> None:
2     self.x = 0
3     self.object = None #type: Optional[List[int]]
```

Listing 3: Example of MyPy type annotations in a class constructor.

2.6 Permission-based Verification

Permission-based verification allows us to verify memory safety. For this kind of verification we need a permission system. Two kinds of permissions are used, read permissions and write permissions. A permission is the right to access a memory location in the heap, which, most of the time, means the right to access a field of an object. Permissions can be held by a method. When a read permission is held, the method is only allowed to read the contents of the memory location. With a write permission, the method is additionally allowed to modify the data stored in the memory location. If a method does not hold any permission to a memory location, it is not allowed to read or write. Listing 4 shows an example of a read permission in Nagini, it states that we have 1/10th permission to access field `f` of object `x`. Read

permissions are denoted by a permission amount between zero and one in Nagini.

```
1 Acc(x.f, 1/10)
```

Listing 4: Example of a Nagini read permission.

A write permission is denoted by `Acc(x.f)`, which is short for `Acc(x.f, 1)`. This is considered a full permission to `x.f`, with a permission amount of exactly one.

Permissions are used to write a specification about memory safety, which means a specification where every method has enough permissions to perform its reads and writes. Nagini automatically enforces that only legal reads and writes of a memory location are allowed in a verified method. This means that for every location in the method body, where, e.g., a field is written to, the method also holds a sufficient permission. Furthermore, method calls are only allowed when all permissions defined in the precondition of the called method are fulfilled. An example of such a precondition is `Requires(Acc(x.f))`, which states that a write permission to `x.f` is required.

When calling a method, the required permissions need to be passed to the callee. To pass a permission to a callee, the caller needs to have at least as high a permission amount as the callee requires. For example, if the callee requires 1/5th permission, the caller requires at least 1/5th permission. The caller passes the permission amount required to the callee. If there is a leftover permission, it is kept by the caller. This means that, if the caller has the exact same permission amount as the callee, it gives away all its permission to the corresponding heap location. From this follows, that the caller forgets any information about the location in heap memory.

In addition to normal permissions, Nagini also features quantified permissions. Listing 5 shows an example of a `Forall` expression used to assert a quantified permission.

```
1 Forall(self.list, lambda el: Acc(el.x, 1/10))
```

Listing 5: Example of a quantified permission using a `Forall` expression in Nagini.

By using such expressions, one can express a statically unknown amount of permissions, since the length and contents of a list are usually not known in a static context. This is a very useful concept and it is one of two ways

how one can express an amount of permissions that is unknown in a static context in Nagini. The other way is by using recursive predicates, but that concept is not relevant to this project.

The logic used by Viper for permission-based verification is called implicit dynamic frames [8].

2.7 Predicates & Pure Functions

A predicate is an abstraction over assertions. In order to make the code more readable, predicates can be used to group contracts together, since they are all assertions. An example of a predicate in Nagini can be seen in Listing 6. In Nagini, a predicate is declared as a function that takes any number of arguments and returns a boolean value. The body of a predicate must contain only a single return statement, that returns the contents of the predicate. Any assertion that is well-defined in the context of the predicate can be included in the contents.

```
1 @Predicate
2 def pred(x: MyClass, i: int) -> bool:
3     return Acc(x.f) and x.f == i
```

Listing 6: Example of a predicate in Nagini.

In the example, we have a permission to field `x.f` and an expression that says that the field is equivalent to the argument `i`. A predicate needs to be self-framing, this means that it needs to include sufficient permissions to all fields accessed within the predicate.

As with fields, we need permissions to use the predicates, e.g., `Acc(pred(x, 2), 1/2)`, which simply denotes a read permission to the predicate. A write permission can be written without the fraction or also simply as `pred(x, 2)`. If one has a permission to a predicate, but requires the contents of the predicate, i.e., the permission to `f` and the knowledge that `x.f == i`, once can *unfold* the predicate. This means that the predicate gets replaced with its contents. `Unfold(Acc(pred(x, 2), 1/5))` would give us 1/5th permission to `x.f` and `x.f == 2`. The opposite of unfolding is *folding*, which replaces the contents with the predicate itself. As with the other expressions using permissions, folding and unfolding can also be done with a full amount. For unfolding, the permission amount of the predicate and the permission amount of a permission assertion within get multiplied. Additionally, there is also an `Unfolding(`*`perm, expr`*`)` expression, where *perm* is a permission assertion

and *expr* is an expression. This unfolding expression evaluates *expr* in a context where *perm* is unfolded.

In Nagini, there are special predicates for lists and dicts, `list_pred` and `dict_pred`, which denote a permission to a list or a dict and all of its elements. They can be used like normal predicates, but they do not need to be unfolded.

Another important concept are pure functions. These are functions that have no side effects, meaning they can be used in contracts. In Nagini they are marked with a `@Pure` annotation, so that the verifier knows to check for purity. Pure functions are only allowed to call other pure functions and they are not allowed to include loops. Further, no heap allocations, e.g., creating new objects, are allowed in a pure function. Recursion is allowed as it only uses the stack.

2.8 Obligations

An obligation is a promise of a method to perform a certain action, that needs to be fulfilled at some point. To reason about progress, termination obligations [9] can be used. `MustTerminate(n)` is an example of a termination obligation in Nagini. If used in a method precondition, it states that the method promises to terminate within `n` steps. A method with `MustTerminate(n)` is only allowed to call pure functions and methods with `MustTerminate(k)`, where `k < n`. When a method contains a loop, an invariant with a `MustTerminate(i)` obligation, where `i` decreases with every loop iteration, but stays non-negative, is needed.

2.9 I/O Specifications

I/O specifications can be used to verify the desired I/O behaviour of a program. Generally, I/O behaviour is how the program interacts with the operating system. Examples of I/O operations are reading and writing to the console. For this project, an important I/O operation is the sending of some data via a network socket.

A simple way of specifying I/O behaviour is using a Petri net. This way of writing I/O specifications is proposed in [10]. Such a Petri net consists of places (nodes) and transitions between the places. Places are represented by circles and transitions are drawn as rectangles. A transition can represent an I/O operation that is performed. There is also a *split* operation, which simply duplicates its input. Sometimes, a *no-op* is also needed, it corresponds

to doing nothing. There are also tokens, which reside in a place and they can be moved between two places by a transition. A transition can be performed when there are tokens in all of its input places and it moves the tokens to its output places In Figure 1, an example of a Petri net specifying the I/O behaviour of a router can be seen. This Petri net specifies, that some data bytes are received and stored in *bs*, then a check is performed on `bs`. If the check is successful, the bytes get transformed by some function *bs'=f(bs)* and the result is passed to the send operation. If the check fails, the data is not further processed, this is represented by the *drop* operation in the figure.

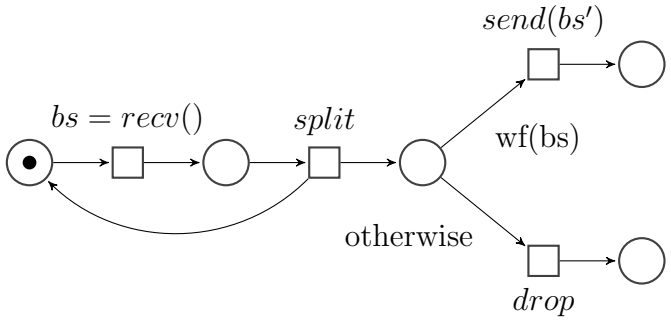

Figure 1: Example of a Petri net with a token in the left most place.

Nagini supports verifying code with respect to such an I/O specifications; the implementation is described in [5]. First an I/O operation has to declared in the code, so that it can later be used in method specifications Listing 7 shows an example of an I/O transition written in Nagini syntax. This could be the code for the transition *send* in Figure 1. It denotes a transition from place `t_pre` to `t_post` and it takes one argument `data`. `Terminates(True)` expresses that send is a non-blocking operation, whereas `Terminates(False)` would state that the operation is blocking and may not terminate. Since an I/O operation will be used in contracts, its return type needs to be `bool`.

```
@IOOperation
def send(
        t_pre: Place,
        data: bytes,
        t_post: Place = Result(),
        ) -> bool:
    Terminates(True)
```

Listing 7: Example of an I/O transition in Nagini.

I/O operations can be used like predicates in contracts, e.g., `send(t1, data, t2)` in a precondition states that the method will perform the send operation. To ensure that the token is eventually moved along in the Petri net, we use an obligation `token(t1, n)`, which states that a token is currently in place `t1` and it will be moved in at most `n` steps.

3 Methodology

3.1 Verification Goals

There are three properties that we verified:

- Memory Safety, i.e., no crashes due to runtime errors, like illegal heap memory accesses
- Progress, i.e., all methods make progress, meaning they will not get stuck in an infinite loop
- I/O Behaviour, i.e., the router correctly forwards a valid packet and discards invalid packets

In order to limit the scope of the project, we chose a specific path in the code to verify. Namely, we assumed that there is only one up-segment in the packet path. Furthermore, we assumed that SCION features like peering, SCION extensions and inter-domain communication are not used. Finally, we assumed that the router is not the recipient of the packet, i.e., the packet still needs to be forwarded.

3.2 The Codebase

The SCION Python router code, which can be found at [7], has two folders which are of interest to us. Namely, `infrastructure`, which contains the main files of the router and `lib`, where SCION library classes and methods are located. The rest of the folders contain files which are not relevant for this project.

The most important files are `main.py`, `scion_elem.py`, `scion.py`, `path.py` and `opaque_field.py`. An overview of the files and folders can be seen in Figure 2. In the scope of this project, the main method is `handle_request`, which is located in the class `Router` in `main.py`. More on the call structure and the call graph will be discussed in Section 4.

The `Router` class inherits from `SCIONElement`, which is defined in `scion_elem.py`. Another very important class is `SCIONL4Packet`, which is the internal representation of a parsed SCION packet. It is located in

[7]https://github.com/sasjafor/scion

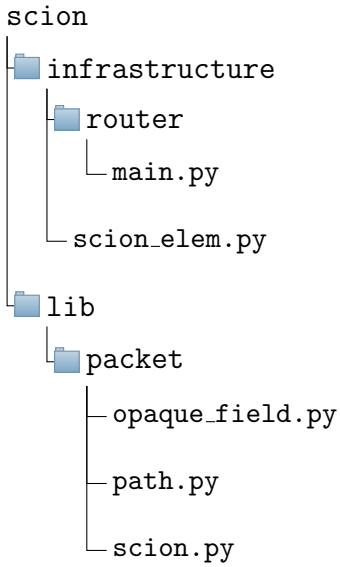

Figure 2: Overview of the codebase; only the important files and folders are included.

`scion.py` and it inherits from `SCIONExtPacket`, which in turn inherits from `SCIONBasePacket`.

A SCION packet contains a field `path` of type `SCIONPath`, which contains the forwarding information of the packet. It contains a list of fields with type `OpaqueField`, which has two subtypes, `InfoOpaqueField` and `HopOpaqueField`. The list is filled with SCION path segments, which start with an info field and get followed by a number of hop fields. An info field contains information about a segment, which includes the number of hops in the segment and a few flags. A hop field contains information about an AS, which includes information about the ingress and the egress interface and also a few flags.

The rest of the classes will not be discussed here, but they will be discussed where needed.

We selected 31 methods for verification, but we verified 24 of them and the rest we annotated with `@ContractOnly`. That constitutes 420 lines of code (LOC), with 334 LOC actually verified. The reasons why we did not verify some of the methods are varied. For some, it would simply have been out of the scope of this project and for others, essential features are not yet

supported by Nagini. Of the 24 verified methods, 6 were annotated as `@Pure`. More statistics about how much code was verified, will be discussed in detail in the results section.

The router code uses mainly standard features of the Python language and built-in types like lists and dicts. As previously mentioned, the code does not have any type annotations. The code also features callables, which are essentially function pointers, however, Nagini does not yet fully support the callable type. This means that we had to rewrite some small parts of the code to equivalent code which does not use callables.

The SCION router code was not written with verification in mind, which means that verification was harder than it would have been otherwise. Also, some features had to be added to Nagini an example is that `enumerate` expressions were not supported previously.

3.3 Python and Types

Nagini runs MyPy[8] on the code before verifying it; thus, we need type annotations. Since the existing Python code of the router does not already have type annotations, we needed to add type annotations. These annotations are described in Section 2.5.

For Python's built-in methods, we used typeshed[9] stub files. These stub files simply have type annotations, but they do not have any Nagini contracts. Nonetheless, using these stub files allows us to assume that those methods are correctly implemented for the purpose of our verification. There might still be mistakes in these built-in methods, but this is a necessary simplification that allows us to only verify the code we are interested in for this project.

3.4 Codebase Additions

For our verification we added a few folders to the codebase. `scion-stubs` contains interface files for SCION classes; there most predicates and SCION methods that only have an interface are located. Those methods were only annotated with contracts, but their bodies were not verified. It also contains all contracts for methods, which are part of the SCION code library in `lib`. The layout of files and folders in `scion-stubs` corresponds directly to the

[8]https://github.com/python/mypy
[9]https://github.com/python/typeshed

locations of the normal files. As an example, the interface file `scion.pyi` is located in `scion-stubs/lib/packet` and it contains annotated versions of the methods in `scion.py`.

As discussed in the previous section, we used typeshed stub files for built-in methods. In `stubs`, there are stub files that override typeshed's stub files. This was necessary to annotate the stubs with Nagini contracts, where it was needed.

The new `adt` package contains only the definition of an abstract data type.

3.5 Verifying Python Code

As mentioned in Section 3.3, we needed to annotate the router code with types first. For a method which we wanted to verify, we needed preconditions and postconditions. If there was a loop, we additionally needed to write a loop invariant.

An example of Nagini contracts can be seen in Listing 8. `Acc(obj.x, 1/10)` denotes a read permission of 1/10. For a read permission, the value of the fraction needs to be positive and less than 1 and we always need at least this permission amount when calling method `foo`. Nagini also supports read permissions without using fraction amounts, but for this verification, it proved to be more reliable and performant to use fractional amounts. Details and statistics about performance and usability of the verifier will be discussed in Section 5.

```
1  def foo(self, obj: bar) -> int:
2      Requires(Acc(obj.x, 1/10))
3      Requires(obj.x >= 0)
4      Ensures(Acc(obj.x, 1/10))
5      Ensures(Result() == 2*obj.x)
6      y = 0
7      while y < obj.x:
8          Invariant(Acc(obj.x, 1/10))
9          Invariant(obj.x == Old(obj.x))
10         Invariant(y <= obj.x)
11         y = y + 1
12     return y + obj.x
```

Listing 8: Example of Nagini contracts.

As discussed in the preliminaries, `Result()` is used to denote the value

of the return value and the `Old` expression can be used in invariants and postconditions to evaluate an expression in the state before the current method was called.

When a method is allowed to raise an exception, we also need a special postcondition to handle the exceptional control flow. Listing 9 shows an example of this. `Exsures(Exception, Acc(obj.x, 1/10))` says that `foo` is allowed to raise an exception of type `Exception`. Furthermore, the second argument to `Exsures` is like a normal postcondition, but it only needs to hold when an exception of the corresponding type was raised. When an exception is raised by a method, the normal postconditions do not need to hold.

```
def foo(self, obj: bar) -> int:
    Requires(Acc(obj.x, 1/10))
    Ensures(Acc(obj.x, 1/10))
    Exsures(Exception, Acc(obj.x, 1/10))
    if obj.x == 0:
        raise Exception("x shouldn't be 0")
    else:
        return obj.x
```

Listing 9: Example of a method with an exceptional postcondition.

We used the termination obligations, discussed in Section 2.8, to reason about progress. Generally, they can simply be added to the precondition, without the need for any changes in the rest of the contracts. However, some additional contracts are sometimes needed to guarantee that the obligations are well-defined. An example would be that the termination measure of a method depends on the length of a list, so we could have this contract: `Requires(MustTerminate(len(l)))`, where `l` is the name of the list. Now, we need to include an additional precondition that states that the length of the list is greater than zero, because the argument to the termination obligation needs to be greater or equal than 1.

For our I/O verification, we used the specification language described in Section 2.9. An example from our verification can be seen in Listing 10. This is our main I/O operation that represents the `udp_send` operation of our I/O specification. It takes two places, `t_pre` and `t_post`, between which it moves the token. Additionally, it accepts some data, which it sends to a destination address (`dst_addr`) and port (`dst_port`), but it does not physically send anything, as this is only specification. By using this I/O operation, we can abstract the actual functionality of `udp_send`. Once again, this is a necessary

step to keep the project in a manageable scope.

```
@IOOperation
def udp_send(t_pre: Place, data: bytes, dst_addr: str, dst_port:
    int, t_post: Place = Result()) -> bool:
    Terminates(True)
```

Listing 10: Example of an I/O operation from our verification.

For the real `UDPSocket.send` method, only the contracts are now regarded in the verification. Therefore, we annotated it with `@ContractOnly`. How the `udp_send` I/O operation is used in the contract of the real `send` can be seen in Listing 11.

```
IOExists1(Place)(lambda t2: (
            Requires(MustTerminate(1)),
            Requires(dst is not None and token(t, 1) and
                     udp_send(t, data, dst[0], dst[1], t2)),
            Ensures(Result()[1] is t2 and token(t2))
        ))
```

Listing 11: Example of an IOExists contract.

An `IOExists` statement starts with a number after the name, which simply states how many variables we want to quantify over. Then, a list of the types of the variables follows, and after that, there is an existential quantifier, which actually introduces the variables. In our example, we have `IOExists1`, for one variable, then the type, which is `Place`. Afterwards, `lambda t2:` quantifies over the variable `t2`. Within the inner brackets, we find the normal pre- and postconditions, but additionally, there are also the I/O contracts. For the contract of `send`, we have the precondition stating that a token resides in place `t` and the method promises to perform the `udp_send` I/O operation. In the postcondition, we ensure that we return the resulting place and that the token has now been moved there.

The reason why this syntax might seem convoluted, is because MyPy's type checker needs to be satisfied.

4 Verification Process

This section describes in detail how we worked on the four tasks of the project, which are described in Section 1.2. Generally, we worked through the tasks in order, because memory safety verification had to completed before a method could be verified for progress, and similarly, before starting to write I/O specifications, all methods had to already be verified for memory safety and progress. However, the definition of a packet abstraction and a mapping function for the third task could be written in parallel with the first task. We started work on the fourth task only once all other tasks were completed.

4.1 Memory Safety

The first task of the project was verifying memory safety for a subset of the methods in the SCION codebase. We spent at least half of the time on this task, as a number of issues with the verifier arose. One big problem was one of performance, this slowed work down in the beginning, since a verification command could take a long time to terminate. Performance will be discussed in detail in the results in Section 5.

4.1.1 Code Overview

First, we are going to describe the code that we verified in more detail. The following figures will show call graphs of some of the verified methods with a limited depth.

Figure 3 shows the call graph for `handle_request`, which is the topmost method that we looked at in the call hierarchy. It receives a SCION packet

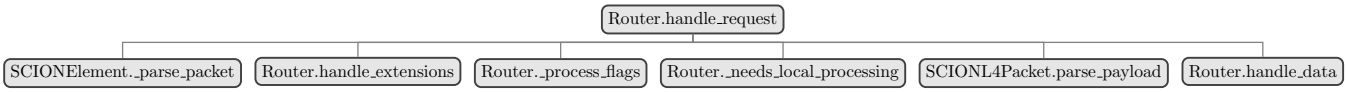

Figure 3: Call graph of `Router.handle_request`.

in byte representation and a boolean variable, which says if the packet was received on a local socket, i.e., if the packet was sent by a client in the same AS as the router. First, it calls `_parse_packet` to parse the packet into an internal representation and then it calls `handle_extensions` to process SCION extensions. Then, `_process_flags` gets called to process any flags and it determines if the packet needs local processing. If it does, then

`parse_payload` gets called and finally, the packet object that was parsed is passed to `handle_data`, which performs some additional checks and then forwards the packet. `handle_request` gets called every time a packet is received by the router.

We continue with the call graph of `handle_data` (Figure 4), which calls `_process_data` and catches any exceptions that `_process_data` might raise and then writes them as errors to a log. `_process_data` calls many methods to check validity of the packet and also to process the packet and then to forward it. Specifically, it calls `verify_hof`, which checks validity of the packet, and `_calc_fwding_ingress` and `inc_hof_idx`, which process the packet by advancing the hop field index by one. Finally, `deliver`, `send` and `_egress_forward` are called to forward the packet, each on a different path in the code of `_process_data`. The call graph for `verify_hof` can be found in Figure 5. `_process_data` also has a call to `Router._validate_segment_switch`, which we also verified, but it later became apparent that this method does not get called in our path in the code. This time was not wasted, however, as it can be used for any future verification efforts of the SCION router code.

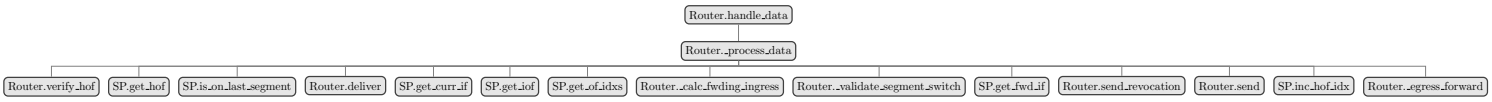

Figure 4: Call graph of `Router.handle_data`, where SP stands for SCIONPath.

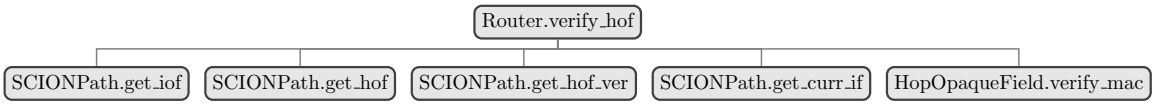

Figure 5: Call graph of `Router.verify_hof`.

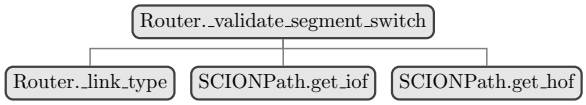

Figure 6: Call graph of `Router._validate_segment_switch`.

4.1.2 Approach and State Predicates

Quickly, after starting work on this first task, it became evident that it would be most efficient to work in a bottom-up manner. When verifying a method, it makes sense to first know about all the preconditions of all called methods, because if this is not the case, a lot of changes might have to be performed later. Additionally, when we are verifying a method, it will only definitely fulfil its contract when the contracts of all called methods stay the same. If a contract of a called method changes, the method will have to be verified again. On the other hand, there is also a disadvantage to working bottom-up, which appears when a postcondition of a called method is insufficient. Insufficient means that a condition which is required after the method call is not guaranteed by the postcondition of the called method. A simple example would be that the called method takes a permission and then does not return it again in the postcondition. This is of course simply a mistake, but if there are a lot of contracts, it can happen rather quickly that a required postcondition is forgotten and the mistake is only discovered later, when the the caller gets verified.

To simplify contracts in our verification, we used state predicates for the different classes. These predicates shall reflect the invariants which must be established by the parser of the SCION router. This means that after parsing a SCION packet successfully, we can rely on these invariants holding at all times, if we ensure that no method violates them. The largest part of a state predicate is made up of access permissions to fields and permissions to the state of child objects.

To write assertions about the state predicates of list elements and other collections, we used *quantified permissions*. Here is an example using a `Forall` expression: `Forall(path._ofs lambda of: of.State())`. The example asserts that for every element of the list `_ofs`, the element's state predicate holds. Alternatively, we could have used recursive predicates, but we decided against it.

For state predicates that are part of the packet classes, these permission assertions must hold for any successfully parsed packet. Additionally, the state predicates of the packets contain assertions specific to our assumptions and they must also be established by a successful parsing. For the state predicate of the `Router` class, the assertions have to hold in any instance of a SCION router as no assumptions about the state of the router were made.

An example of a state predicate from the class `SCIONBasePacket` can be

seen in Listing 12.

```
@Predicate
def State(self) -> bool:
    return (Acc(self.cmn_hdr) and
        Implies(self.cmn_hdr is not None,
                self.cmn_hdr.State()) and
        Acc(self.addrs) and
        Implies(self.addrs is not None, self.addrs.State()) and
        Acc(self.path) and
        Implies(self.path is not None, self.path.State()))
```

Listing 12: State predicate of `SCIONBasePacket`.

In Nagini, classes will inherit predicates declared in the class scope. If a predicate overwrites the predicate from the super class, the resulting predicate will be a conjunction of the predicate from the superclass and the class' own predicate. Since state predicates are declared within classes, this also holds for them.

When an object is passed as a parameter to a method, the necessary permissions to fields and any additional conditions can simply be passed via the state predicate of the object. We included any conditions which only need to hold for a certain method, but not at every point in the code after parsing, in the method's precondition.

In the following sections, some of the previously mentioned methods will be looked at in detail. We chose to discuss the methods which were the most difficult to verify, as to give an insight into what kinds of difficulties can arise when verifying real-world code. They will, generally, be treated in a bottom-up manner.

4.1.3 `SCIONPath.get_hof_ver`

This method's purpose is to retrieve a hop field, which will be used for verification of the current hop field. Put simply, the method either returns the result of `_get_hof_ver_normal` or calculates an offset and then retrieves the hop field at that index. As a reminder, hop fields are stored in segments, and each segment is simply a list of hop fields. The result of `_get_hof_ver_normal` is also just the hop field at an offset. In our path in the code, the offset for `_get_hof_ver_normal` is 1, which means that we simply retrieve the next hop field, if there is one.

```
def get_hof_ver(self, ingress: bool =True) -> Optional[HopOpaqueField]:
```

Listing 13: Definition of `SCIONPath.get_hof_ver`.

The biggest challenge when verifying this method was adapting the `State` predicate and writing an appropriate precondition. What made it challenging was the fact that hop fields are stored in a segment along with info fields. A segment starts with a field of type `InfoOpaqueField`, which contains some flags and other info about the segment, followed by all the hop fields in the segment, which have type `HopOpaqueField`. Retrieving a field, hop or info, from the list of type `OpaqueFieldList`, is performed with `get_by_idx`.

Now, the difficult part was guaranteeing that `get_by_idx` returned a `HopOpaqueField` in `get_hof_ver`. The offset with which `get_by_idx` gets called depends on `ingress` and two flags of the info field, the up-flag and the peer-flag. From our assumptions in the state predicate of `SCIONPath`, we know that the up-flag is `True` and the peer-flag is `False`. So, now the offset only depends on `ingress` and is either `None` or `1`.

```
Implies(not ingress,
    self.get_hof_idx() + 1 < self.get_ofs_len() and
    isinstance(self.ofs_get_by_idx(self.get_hof_idx() + 1),
               HopOpaqueField))
```

Listing 14: Part of the precondition of `get_hof_ver`.

Listing 14 shows the relevant part of the precondition. The first part gives us the necessary condition that we are allowed to call `get_by_idx` with an offset of 1 and the second part states that the returned field is of type `HopOpaqueField`. This gives us enough information to guarantee that the return type of `get_hof_ver` is `HopOpaqueField`.

Listing 15 shows the implementation and contracts of `get_by_idx`. Since we needed this method in contracts, we had two choices, either annotate it as pure and contract-only, or write a separate function that is pure and then write a postcondition for `get_by_idx`, which states that `Result() is self.get_by_idx_pure(idx)`. We decided on the first approach, because this method is quite far down in the call graph and as discussed before, we had to limit the scope of the project to make the verification viable. This would also make verification faster, because if there were two versions of the same method, one for contracts and one for method bodies, the verifier would need to match them and check the postcondition of `get_by_idx` to know that it yields the same result. The `@ContractOnly` annotation is needed because the

method contains a for-each loop, which is not allowed in a pure function.

Another option might have been to rewrite the method using recursion and thus making it pure, but the aim of this project was to verify the existing code and not rewrite whole methods.

```
@Pure
@ContractOnly
def get_by_idx(self, idx: int) -> OpaqueField:
    Requires(Acc(self.State(), 1/20))
    Requires(idx >= 0 and idx < self.get_len())
    Ensures(Result() is
    Unfolding(Acc(self.State(), 1/20), self.contents()[idx]))
    Ensures(Result() in
    Unfolding(Acc(self.State(), 1/20), self.contents()))
    if idx < 0:
        raise SCIONIndexError("Requested OF index (%d) is
                               negative" % idx)
    offset = idx
    for label in self._order:
        group = self._labels[label]
        if offset < len(group):
            return group[offset]
        offset -= len(group)
    raise SCIONIndexError("Requested OF index (%d) is out of
                          range (max %d)" % (idx, len(self) - 1))
```

Listing 15: Code snippet of `OpaqueFieldList.get_by_idx`.

The many getter functions, like `self.get_hof_idx()`, are used for performance reasons. We found out that many `Unfolding` expressions in a method contract or a method body cause performance issues that are clearly related to the amount of unfoldings present, whereas performance would not be affected if they were hidden in the bodies of pure functions. For this reason, we had to move as many unfoldings as possible to separate functions, so that every getter only includes a single unfolding expression. A getter function simply returns the corresponding field, in this case `self._hof_idx` or calls the corresponding function, `self._ofs.get_by_idx(k)` for `self.ofs_get_by_idx(k)`. In the results section, these getters and the performance improvements they bring will be discussed in more detail.

4.1.4 SCIONPath.inc_hof_idx

This method actually moves the pointer in a SCION segment by one hop. It would also handle a segment switch, but because we assumed that there is only one up-segment in the path and that the router is not the recipient of the packet, we did not verify this functionality. These assumptions are partly included in the state predicate of `SCIONPath`, but the majority is included in the precondition of `inc_hof_idx`, because it is specific to only this method and not required anywhere else. So, by including it in the precondition, changes can later be made, without affecting the whole path class. Initially, the assumptions about the packets we receive are established as part of the postcondition of the packet parsing. This is a simplification, as in reality, the assumptions should be established as part of the precondition of handle_request, but there we would have had to to write contracts about bytes, since we did not yet have an abstraction of the packets, so we decided to use the postcondition of `_parse_packet`.

Since this method actually changes a field of an object, it needs a full permission. This is the only method, of the ones we verified, that performs a field write and is as such different from all the other methods. It is evident that the callers of `inc_hof_idx` also need a full permission. In this case, this means a full permission to the state predicate of the packet and for `inc_hof_idx` a full permission to the state predicate of the SCION path.

```
def inc_hof_idx(self):
        iof = self.get_iof()
        skipped_verify_only = False
        while True:
            self._hof_idx += 1
            if (self._hof_idx - self._iof_idx) > iof.hops:
                # Switch to the next segment
                self._iof_idx = self._hof_idx
                iof = self.get_iof()
                # Continue looking for a routing HOF
                continue
            hof = self.get_hof()
            if not hof.verify_only:
                break
            skipped_verify_only = True
        return skipped_verify_only
```

Listing 16: Implementation of SCIONPath.inc_hof_idx.

The state predicate of the path was already quite complicated, but some additions had to be made to accommodate the conditions needed for `inc_hof_idx`. The difficult part was to guarantee the state predicate in the postcondition and for this purpose an extensive loop invariant was needed.

In SCION routing, some segments have a verify-only hop field at the end of the segment. These hop fields are only used for the purpose of hop field verification, which we will not explain here. In Listing 16 it can be seen that the only exit from the loop is via a `break` statement, which gets executed when the current hop field is not verify-only. Breaking out of the loop means that our invariant does not need to hold after the loop. In the loop, a verify-only hop field gets skipped, but this results in a segment switch, because the verify-only hop field is the last field in a segment. With our assumptions, we know that no segment switch can take place and thus, the loop only gets executed once. So we wrote a precondition for `inc_hof_idx` which asserts that our hop field index is lesser than the second last index and that the next hop field is not a verify-only hop field. We need to write about the next hop-field, because in the loop the index always gets incremented first.

With the addition of some further preconditions, which helped the verifier prove that the state predicate still holds after the method breaks out of the loop, we could finally verify that the state predicate also holds after the execution of `inc_hof_idx`.

4.2 Progress

Verification of progress was the simplest task of the four. For most methods a simple `MustTerminate(k)` termination obligation had to be added. Here, the bottom-up approach was quite advantageous, because we could simply start with the methods in the leaves of the call graph by adding an obligation with `k=1` to their preconditions. Then, we worked our way up through the call graph by adding preconditions to the callers with `k=k+1`, i.e., we just added 1, when we went to the parent in the call graph.

The only method where a slightly more elaborate termination obligation was needed was `Router._link_type`. Here, we wrote a precondition that includes `MustTerminate(self.get_topology_border_routers_len() + 4)`. This means that the number of steps until termination depends on the length of the list that stores other border routers, which are in the network topology. Additionally, we also require that this length is non-negative,

because otherwise the argument to the termination obligation might not be positive. This is guaranteed by the postcondition of the getter function for the border router list length. The `+4` is required for the case when the length is 0, so that we still have a high enough number of steps to call the methods in the body of `Router._link_type`.

4.3 Modelling SCION Packets

The ADT, which stands for abstract data type, serves as an abstraction of a SCION packet. Our ADT has fields like a normal object, but we require no permissions to access them, since the ADT only appears in contracts and is immutable. This means that this ADT is only defined in the static context of our verification. We call the ADT pure or stateless.

An ADT is needed as an abstraction to write I/O contracts for our main method `handle_request`. Since `handle_request` only receives the SCION packet as bytes, it is the method which calls the parse method. In order to write an I/O contract for `handle_request` about the packet without reasoning directly about bytes, we need to abstract the bytes to an abstract packet, using a function we call `bytes_to_adt`. Then, we use a function called `incremented`, which takes an abstract packet and returns a new abstract packet that is identical to the input except for the hop field index, which is incremented by one. Finally, we pass the result to a function called `adt_packed`, which maps the abstraction to a byte representation of a SCION packet.

Figure 7 shows a diagram of the different functions in relation to the three important objects. `packed` and `adt_packed` map a SCION packet and a packet abstraction to bytes, respectively. `map_scion_packet_to_adt` is a function that maps a SCION packet to our defined abstract packet.

We defined the `ADT_Packet` type in the file `adt/adt.py`. In Nagini, there is support for ADTs with a predefined ADT type. Then, we needed to define the base class for our ADT, which we chose as `ADT_base(ADT)`, and this class then inherits from the Nagini ADT type and all our ADT classes inherit from `ADT_base`. A class extending `ADT_base` and `NamedTuple` then defines an ADT constructor. The arguments to `NamedTuple` define the arguments of the constructor, where the first argument to each tuple is the name and the second argument is the type. Such a constructor is used like a normal constructor, e.g., `ADT_Packet(addrs, path)`. The definition of the `ADT_Packet` type can be seen in Listing 17.

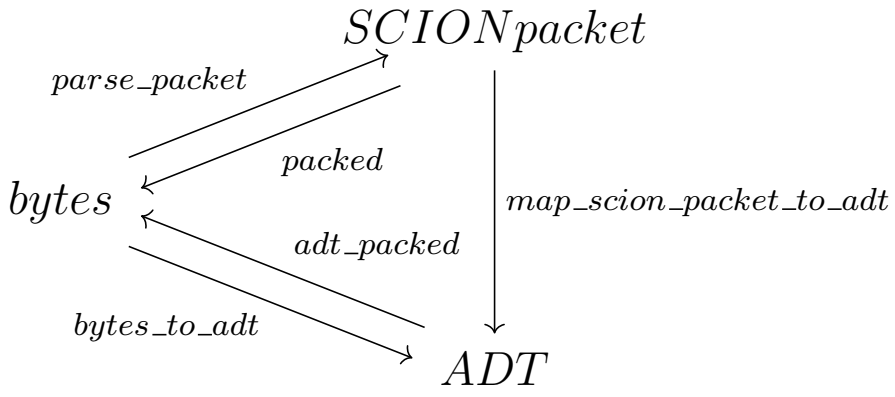

Figure 7: Diagram showing the relation of parsing, mapping to the ADT and mapping to bytes.

```
class ADT_Packet(ADT_base, NamedTuple('ADT_Packet', [('addrs',
    ADT_AddrHdr), ('path', ADT_Path)])):
    """
    Constructor for ADT_packet
    """
    pass
```

Listing 17: Definition of ADT_Packet type.

We kept the structure of the ADT quite close to the structure of the `SCIONBasePacket` type. However, the ADT does not include all the information that is included in a packet, it only contains the information which is necessary in contracts written using the ADT. The only major difference is that info and hop fields are kept separate, but the rest of the structure is shared between the ADT and the SCION packet class. Based on our assumptions, we only need to store one up-segment, so we chose to have a field for the single info field and a pure (stateless) Nagini sequence to store the hop fields. We need to use a sequence because our abstract packet has to be stateless and immutable, so we could not use a normal list, as it has a state.

For our verification, we used the ADT as the return type of a function that maps a SCION packet to an abstract packet, which we called `map_scion_packet_to_adt`. Since we need to iterate through the list of hop fields when mapping them to the ADT, we had to use recursion to keep our

mapping function pure.

4.4 I/O Behaviour

The aim of this task was to verify correctness of the I/O behaviour of the SCION router implementation. We used a Petri net, as discussed in Section 2.9, to specify I/O behaviour. Figure 8 shows the Petri net, which is a representation of the SCION router's intended I/O behaviour. It specifies that first, a data string is received and stored in *bs*. Then, there is a *split* operation, to allow a repetition of the *recv* operation. The first token will be moved to *p1* and the second token from the *split* operation is moved to *p3* in the Petri net and a check is performed. This check tests if the received data is a well-formed SCION packet; if it is, the packet is processed and then forwarded with the *send* operation, represented by *send(bs')* in the figure. *bs'* is a byte representation of the processed SCION packet and it is the result of applying the `packed` function after the packet has been processed. If the check fails, the data is simply ignored and no further action is performed. In either case, a token will reach either *p4* or *p5*, which represent the end of the path. For every received packet, this path is followed again.

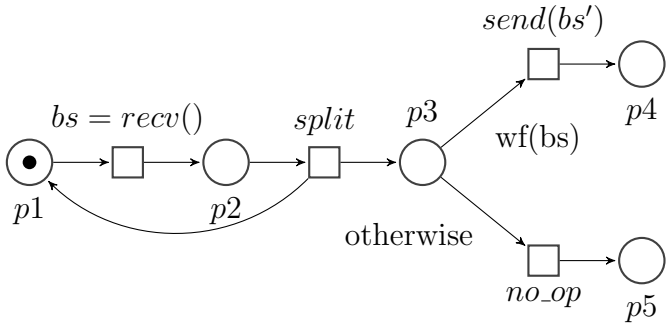

Figure 8: Petri net specifying the intended behaviour of the SCION router.

However, the *receive* operation and the *split* operation are outside of the scope of this project and thus only the *send* operation and the *no_op* operation are relevant. We decided not to use an explicit *no_op* operation, as it simply represents not performing any action. So, it was enough to specify that if the received data is well-formed, the *send* operation is performed.

In Listing 18 the *send* operation is displayed. `t_pre` and `t_post` are the input and the output places. The I/O operation will move a token from

the pre place to the post place. Since this I/O operation represents a real library method from the SCION codebase, namely `UDPSocket.send`, the arguments from the real send operation need to be accepted by the I/O operation as well. Here we have some `data` to be sent in bytes, and `dst_addr` and `dst_port`, which represent the destination the packet will be forwarded to. We assumed that the *send* operation is non-blocking and so we added the `Terminates(True)` contract.

```
@IOOperation
def udp_send(t_pre: Place, data: bytes, dst_addr: str,
             dst_port: int, t_post: Place = Result()) -> bool:
    Terminates(True)
```

Listing 18: Definition of the send operation.

Figure 19 shows the actual send method. This method is central to our I/O specification, as the whole specification is based around it. For the purpose of the progress verification, we have a termination obligation in the precondition of send. Then, we also require that the destination tuple, which contains the address and the port, is not `None`, so that we can perform the actual send on a networking socket. Furthermore, we use the previously described syntax to specify that a token is in the input place and we promise to move it within one step. In the postcondition, we state that the output place is returned in the return value and that a token now resides there. The input place is passed as an argument to `send` and the output place is returned together with the normal return value of the method. Finally, we have the contract of `udp_send`, with which we promise to perform this I/O operation and where we pass the arguments.

```
@ContractOnly
def send(self, t: Place, data: bytes, dst: Tuple[str, int]=None)
    -> Tuple[bool, Place]:
    IOExists1(Place)(lambda t2: (
        Requires(MustTerminate(1)),
        Requires(dst is not None and token(t, 1) and
                 udp_send(t, data, dst[0], dst[1], t2)),
        Ensures(Result()[1] is t2 and token(t2))
    ))
```

Listing 19: Contract of `UDPSocket.send`.

For the I/O specification, we also chose to work in a bottom-up manner, as

it appeared to be the best approach. Every method that calls `UDPSocket.send` will accumulate a path constraint, which leads to the call of `send`. This path constraint will be used in the precondition to state that the `udp_send` operation will be performed only under these conditions. Evidently, accumulating the path constraint works best when working from the `UDPSocket.send` method upward to the methods calling send and their callers. In the topmost method, `handle_request`, we accumulate the complete path constraint under which the send will be performed. There, it can be checked if this path constraint complies with the SCION protocol.

Figure 20 shows an example of such a path constraint. In addition to the normal pre- and postcondition about the places and tokens, we have an implication, with the path constraint on the left side and the I/O operation on the right side. Generally, the arguments passed to `send` in the method body have to be the same as the arguments passed to the I/O operation in the precondition. Here, `send` is actually `Router.send`, a method that itself calls `UDPSocket.send`.

```
def deliver(self, t: Place, spkt: SCIONL4Packet, force: bool=True) -> Place:
        IOExists1(Place)(lambda t2: (
            Requires(token(t, 3)),
            Requires(Implies(
                not (not force and
                     not ((spkt.get_addrs_dst_isd_as() is None and
                           self.get_addr_isd_as() is None) or
                          (spkt.get_addrs_dst_isd_as() is not None and
                           self.eq_isd_as(spkt)))) and
                not (spkt.get_path_len() and
                     ((not force and
                       spkt.get_path_hof_forward_only(spkt.get_path_hof())))) and
                not (spkt.get_path_len() and
                     spkt.get_path_hof_verify_only(spkt.get_path_hof())),
                udp_send(t, packed(spkt),
                    str(self.get_srv_addr_pure(
                                    SVC_TO_SERVICE[spkt.get_addrs_dst_host_addr()],
                                    spkt)
                    if ((spkt.get_addrs_dst_host().TYPE is not None and
                         spkt.get_addrs_dst_host().TYPE == AddrType.SVC) and
                        spkt.get_addrs_dst_host_addr() in SVC_TO_SERVICE)
                    else spkt.get_addrs_dst_host()),
                    SCION_UDP_EH_DATA_PORT, t2))),
            Ensures(Result() is t2 and token(t2))
        ))
        spkt_isd_as = spkt.get_addrs_dst_isd_as()
        self_isd_as = self.get_addr_isd_as()
        if not force and not ((spkt_isd_as is None and self_isd_as is None) or
                              (spkt_isd_as is not None and self.eq_isd_as(spkt))):
            raise SCMPDeliveryNonLocal
        if spkt.get_path_len():
            hof = spkt.get_path_hof()
            if not force and spkt.get_path_hof_forward_only(hof):
                raise SCMPDeliveryFwdOnly
            if spkt.get_path_hof_verify_only(hof):
                raise SCMPNonRoutingHOF
        addr = spkt.get_addrs_dst_host()
        if addr.TYPE is not None and addr.TYPE == AddrType.SVC:
            if spkt.get_addrs_dst_host_addr() in SVC_TO_SERVICE:
                service = SVC_TO_SERVICE[spkt.get_addrs_dst_host_addr()]
                addr = self.get_srv_addr(service, spkt)
            else:
                raise SCMPUnknownHost
        return self.send(t, spkt, addr, SCION_UDP_EH_DATA_PORT)
```

Listing 20: Implementation of `Router.deliver`; non-I/O contracts, comments and logging commands have been omitted for brevity.

Once again, we have the data and the information about the destination as arguments. `packed(spkt)` is simply the SCION packet converted to bytes and it is obvious that this conversion is necessary, as `udp_send` takes bytes as an argument and not `SCIONL4Packet`.

Initially, we tried to write the contract for the I/O operation in a way that gives a better overview. We wanted to write separate implications for different paths in the code of the method body leading to the call to `send`. This was not possible, because of a known limitation of the Nagini encoding of I/O contracts.

Based on this condition:

```
((spkt.get_addrs_dst_host().TYPE is not None and
spkt.get_addrs_dst_host().TYPE == AddrType.SVC) and
spkt.get_addrs_dst_host_addr() in SVC_TO_SERVICE),
```

we wrote two implications, one where it is `True` and one where it is `False`, with the corresponding I/O operation on the right side, because the address argument is different. However, this did not work, because Nagini encodes the first mention of the I/O variable `t2` as the definition of this variable and when the variable is then used subsequently in the code, a getter function is used to retrieve the variable. This getter function also takes the arguments of the I/O operation as parameters. In our case, see Listing 21, we had the first mention of `t2` on the right hand side of the first implication, and a part of one argument, `SVC_TO_SERVICE[spkt.get_addrs_dst_host_addr()]`, was only well-defined, because of the implication. `SVC_TO_SERVICE` is a dict and Nagini requires that no dict lookup raises an exception, which means that the key, that is looked up, needs to be contained in the dict. Here, only the first implication guarantees that this is the case and in the second implication, where the getter is used for `t2` in the encoded Viper code, that condition is not guaranteed. This is why we had to use a conditional expressions, as can be seen in Listing 20, to avoid this problem.

```
IOExists1(Place)(lambda t2: (
Requires(Implies(
    some_conditions,
    Implies(((spkt.get_addrs_dst_host().TYPE is not None and
                    spkt.get_addrs_dst_host().TYPE == AddrType.SVC) and
                    spkt.get_addrs_dst_host_addr() in SVC_TO_SERVICE),
    udp_send(t, packed(spkt),
    str(self.get_srv_addr_pure(SVC_TO_SERVICE[spkt.get_addrs_dst_host_addr()], spkt),
    SCION_UDP_EH_DATA_PORT, t2)))
    Implies(not((spkt.get_addrs_dst_host().TYPE is not None and
                    spkt.get_addrs_dst_host().TYPE == AddrType.SVC) and
                    spkt.get_addrs_dst_host_addr() in SVC_TO_SERVICE),
    udp_send(t, packed(spkt), spkt.get_addrs_dst_host()), SCION_UDP_EH_DATA_PORT, t2))
    ))),
```

Listing 21: Original I/O contracts for deliver, where some_conditions are the conditions from the previous listing.

In addition to writing the I/O contracts, we also had to make sure that other methods could guarantee the conditions needed for these contracts. One example is that we needed to guarantee that in `inc_hof_idx` only the index for the hop field gets incremented by one. We need this knowledge for our I/O contracts, specifically, we have a helper method `call_inc_hof_idx`, which has this postcondition: `Ensures(map_scion_packet_to_adt(spkt) is incremented(Old(map_scion_packet_to_adt(spkt))))`. As mentioned earlier, `incremented` simply returns a new abstract packet where the hop field index is increased before.

We could write those contracts for `inc_hof_idx` about either the path object itself or about the abstract representation. First, we tried to write

them using the object itself, but then the verifier could not connect these postconditions of `inc_hof_idx` with the postcondition of `call_inc_hof_idx`. So, we decided to write them using the abstract representations of the packet, by using the mapping functions. The contracts itself were quite simple, as they only needed to assert that nothing, except for the hop field index had changed.

5 Results

In this section we present the results of the project. This includes detailed statistics about how much code was verified, details about bugs in Nagini that we identified and details about some new features that were introduced to Nagini in the course of this project. Finally, we also present comparisons of the performance of the different backends and the results of performance optimisations we performed during the project.

5.1 Verification

5.2 Verified Properties

In the end, we successfully verified 24 methods for memory safety and progress and we verified 21 of these methods in regards to our I/O specification. As previously discussed, we performed the verification under the following assumptions about the router and the received packets:

- The path of a packet only contains one up-segment
- No SCION extensions, peering or inter-domain communication are used
- Nodes are assumed to be available
- The router is not the recipient of the packet

During the verification, we identified conditions under which the verified code is correct. Some of these conditions, specifically, conditions about the state of the router itself, are in the precondition of the main method in our scope. The rest of the conditions concern the received packet and we assume that the parsing establishes them. These are mainly conditions which say if a packet is well-formed and additionally, they reflect the above assumptions that we made.

An additional part of this project was to identify any bugs in the router code, but since we built up preconditions and invariants and did not yet verify the parsing implementation, we did not find any. To identify potential bugs, one has to verify the parsing implementation and increase the scope to include the methods which call our main method. If all the invariants can be guaranteed, the router is proven to be correct and otherwise, one can then identify the bugs. We think that the invariants and preconditions that we

built up look reasonable and do not contain any obvious mistakes. Thus, we can say that the router implementation that we verified does not contain any definite bugs, but can only contain bugs that results from the interaction with the parsing.

We could verify only 21 of the 24 methods for I/O behaviour, because we faced a performance problem. The problem, which caused the verification to not terminate even after a very long time, could not yet be precisely identified, but our conjecture is that it may be caused by a matching loop. The problem does not seem to lie with the specifications written in this project, but rather in the encoding of the Nagini verifier.

We could not verify two methods, `handle_data` and `_process_data`, due to the performance problem. Additionally, we could not verify `handle_request`, as there was not enough time to resolve problems concerning the packet abstraction, but even so, the same performance problem was likely to affect this method as well, as the I/O contract looks very similar. As a reminder, we had to write the I/O contract of `handle_request` using the packet abstraction. Additionally, `deliver` and `_needs_local_processing` were also affected by the performance problem for the symbolic execution backend. However, we could verify them using the verification condition generation backend.

5.2.1 Statistics

In this section we will discuss statistics about the verified code. The following lines of code (LOC) statistics do not contain any empty lines or code comments. They only contain the LOC of the methods that we verified and do not contain any class declarations, referenced constants or exception constructors, among others. The SCION router codebase is quite complex and contains many interdependencies. Therefore, counting exact numbers for LOC statistics is difficult and the given numbers should be regarded as a lower bound for the amount of verified code, as we did not count referenced code like, e.g., exception constructors.

In summary, we verified 334 LOC of methods of the SCION router code for memory safety and progress. These LOC consist of 24 methods of the SCION router code base. This amounts to 334 LOC of verified code versus the 422 LOC initially selected for verification. We chose to assume the correctness of some selected methods to keep the scope of the project more manageable and to focus on the methods that are more important to the path of the code that we wanted to verify. Six of the verified methods we could verify as

pure functions; they consist of 32 LOC. Pure means that we verified memory safety, but assumed functional correctness.

Our declarations of the ADT types comprise 47 LOC and the functions used to map a SCION packet to the ADT consist of 91 LOC. Naturally, we also verified the ADT mapping functions.

In total, we wrote about 690 LOC of method and function contracts, which includes contracts for all methods that were either one of our selected methods or a method or function that is called by one of our selected methods. It also includes contracts for methods that we did not verify, which constitute approximately 130 LOC. This leaves us with about 560 LOC of contracts for the 24 methods we verified. All these numbers include contracts for memory safety, progress and for the I/O specification.

For the state predicates we wrote an additional 164 LOC, where the largest part is the state predicate of `SCIONPath` with 40 LOC. Additionally, we required a variety of small helper functions to aid our verification. We wrote 25 of them and they consist of 187 LOC. These are additional helper functions, which are different from the ones used for performance improvements. The performance helper functions required an additional 1162 LOC and they will be discussed in more detail in Section 5.3.3.

To summarise, we wrote 690 LOC of method and function contracts, 164 LOC of predicates, 91 LOC of ADT mapping functions, 47 LOC of ADT declarations, 187 LOC of small helper functions and 1'162 LOC of performance helper functions, which is a total of 2'341 If we calculate the specification overhead ratio for our verification, which is the ratio of LOC of the whole specification to the LOC of verified methods, which was 334 LOC, the result is a ratio of 7. Without the performance helper functions, the specification consists of 1'179 LOC and the ratio is 3.5.

5.3 Nagini

As a result of this project, the Nagini verifier was improved in multiple ways. Specifically, this project helped identify many bugs in Nagini and also some in the Viper backends. Furthermore, new features were added to Nagini by its maintainer, because we required them for this project. Bugs and new features will be covered in the following two sections.

5.3.1 Bugs & Problems in Nagini

As mentioned earlier, a part of the project was to identify any bugs that the Nagini verifier might have. In the end, we could identify 25 different bugs in Nagini and Viper and most of them were fixed. The rest of the bugs will be fixed in the future.

A major problem that we faced when we started to work on this verification was the fact that currently, wildcard read permissions, i.e., read permissions without a concrete permission amount, are unreliable. Such a wildcard read permission is denoted by `Rd(obj.x)`, which is similar to `Acc(obj.x, 1/10)`, but it foregoes the need for a concrete permissions amount. In theory, this would be beneficial, because there are only two real kinds of permissions, read permissions and write permissions, but by using fractional permission amounts for read permissions, we create arbitrary distinctions between read permissions and increase the annotation overhead. This is generally not necessary, as read and write permissions would be enough to perform permission-based verification. However, we have encountered cases where a method had enough permissions and should have been verifiable, but the verifier could not verify the method. So, we found the wildcard read permissions to be too unreliable as the verifier is incomplete when they are used and we rewrote all wildcard permissions to permissions using a concrete permission amount. This proved to be quite a lot of work, since now, the permission amounts also needed to match for callers and callees.

5.3.2 New Features

Since this project required them, two new features were added to Nagini.

One new feature that was added to Nagini is support for `enumerate` expressions. This expression was used in the SCION router code and previously, Nagini did not support it. `enumerate(iterable)` indexes the `iterable` and returns a new `iterable` with tuples that contain the index, which by default starts at zero, and the original element.

However, a much more powerful addition is the newly introduced `Let` expression. This expression allows us to declare helper variables in Nagini contracts. A trivial example can be seen in Listing 22.

```
1 Let(5, bool, lambda five: five >= 0)
```

Listing 22: Example of a `Let` expression in Nagini.

The first argument to `Let` determines the value of the variable and the second argument determines the type of the expression, which comes after the lambda expression. Essentially, the above example is equivalent to this more commonly used syntax for let-expressions: `let five : 5 in five >= 0`. The type of the sub-expression is needed for the type checker.

5.3.3 Performance

Usually, performance of a verifier is known for small and fabricated examples and possibly for the tests in a test suite, but when a verifier is used with real code, which was not written with verification in mind, the results can differ.

Unfolding Expressions: In our case, the performance of Nagini was very reasonable for small examples, most terminating within just a few seconds. However, when working on this project, we realised that the performance of Nagini can become very bad, even for smaller methods. The biggest realisation was that `Unfolding` expressions of complicated predicates cause Nagini to have bad performance. In our case, we used state predicates, which can have a lot of dependencies. This will result in a large verification problem, which can then take a long time to verify. We discovered that the relation between performance and the number of `Unfolding` expressions in a method was directly related. As a result, methods can take up to multiple hours to complete verification.

However, we found a workaround for the performance problem concerning unfoldings. As briefly mentioned in Section 4.1.3, performance can be improved by separating each unfolding into its own function. Extracting an unfolding leads to completely equivalent code, but nevertheless, the verification performance increases greatly. These need to be pure functions, so that the caller knows the contents of the function. As an example, the expression `Unfolding(Acc(self.State(), 1/10), self.addr)` was extracted into a separate function as in Listing 23.

```
1 @Pure
2 def get_addr(self) -> SCIONAddr:
3     Requires(Acc(self.State(), 1/10))
4     return Unfolding(Acc(self.State(), 1/10), self.addr)
```

Listing 23: Example of a getter function used to improve performance.

This small example shows that the original expression is now simply returned in the new function. Additionally, the function requires a precondition for the permission to the state predicate that is used in the unfolding.

If an expression included two or more unfoldings, it had to be split into multiple functions, where each function only included one unfolding. This was necessary to maximise the performance benefit. For some expressions, a more complicated precondition was necessary to allow for successful verification. An example would be a function call within an unfolding expression. We need to then write preconditions to pass the necessary conditions to the called functions, but this also means that this transformation can not be performed automatically, as different expressions in an unfolding might require more than just permission to the predicate that is unfolded. The more complicated such a getter function or helper function becomes, the smaller the performance benefit becomes as well, as more contracts need to be checked when calling such a function. Thus, no performance could be gained by extracting an unfolding in some cases.

All performance measurements that follow were measured using the symbolic execution backend if not stated otherwise.

As a small example of the performance improvements, we will look at the state predicate of `SCIONPath`. These measurements were made with a previous version of the state predicate, but nonetheless, they can serve as an example of the performance benefits. Table 1 shows the verification time for the state predicate with all unfoldings, with three of them replaced with functions and with four of them replaced.

State Predicate	Verification Time
without changes	32s
three unfoldings replaced with functions	26s
four unfoldings replaced with functions	23s

Table 1: Table showing performance improvements of using separate functions for unfoldings.

However, these measurements of the state predicate do not show how significant the performance improvement was for the verification times of the methods. Table 2 shows the difference in performance of method verification

for four methods in `SCIONPath`. These measurements were all made before we started to work on the I/O verification. In addition to the above optimisations in the state predicate, we also extracted any unfolding expressions that were included in the contracts and the bodies of these methods.

Method	Without Optimisation	With Optimisation
get_curr_if	41m6s	1m11s
get_hof_ver	72m36s	1m36s
get_fwd_if	125m14s	1m22s
_get_hof_ver_normal	22m54s	1m9s

Table 2: Table showing performance improvement of method verification times.

On average, that is a performance improvement of 4691%, which is a very significant change.

After performing these optimisations wherever we could, we achieved somewhat acceptable performance for almost all methods. Ten methods took less than a minute to verify, but 12 took between one and ten minutes. The only outlier is `_process_data`, which still took around 43 minutes to verify. A part of the reason why some methods are still slow to verify could be that there are still some unfolding expressions that we could not extract to a separate function.

With some unfolding expressions, there is a problem that prevents them from easily being separated into functions. The problem concerns function calls which require certain preconditions that might depend on the context of where they get called. Now, in order to successfully separate them into a function, all preconditions need to be passed to the new function. However, these preconditions could get too complicated, such that there was no performance benefit of using a separate function any more.

Experimental Type System Encoding: During the project, we tested an experimental type system encoding in Nagini. We did not have a lot of time to test the performance differences, but we did test it for one method. For `SCIONPath.get_hof`, we measured a verification time of 4m38s with the old type system encoding. With the new experimental type system encoding, we measured a time of 3m38s, which is a performance improvement of 27%. As soon as we measured a performance improvement, we used the new type

system encoding as often as possible, but there were cases where there were still some problems, which had to be fixed. Now, the new encoding proves to be quite reliable as we encountered no further errors.

Comparing Nagini Backends: We were also interested in comparing the performance of the two backends that Viper uses. One is the verification condition generation (VCG) backend and the the other is the symbolic execution (SE) backend. In Table 3, there is a comparison of the verification runtimes of both backends for the majority of the verified methods. These times are measurements taken after all contracts were written, including the I/O contracts.

Method	VCG backend	SE backend
handle_extensions	1m7s	1m43s
_process_flags	50s	55s
verify_hof	55s	3m12s
_calc_fwding_ingress	119m30s	34m39s
_validate_segment_switch	55s	2m2s
send	16m29s	87m15s
_egress_forward	2m12s	87m53s
_link_type	57s	1m17s
get_all_border_routers	11m28s	55s
is_on_last_segment	49s	1m16s
get_curr_if	50s	1m52s
get_fwd_if	50s	1m54s
get_iof	49s	1m20s
get_hof	48s	1m19s
get_of_idxs	48s	1m10s
inc_hof_idx	1m5s	4m5s
get_hof_ver	54s	2m16s
_get_hof_ver_normal	51s	1m59s
verify_mac	49s	44s

Table 3: Table showing the performance differences between the two Viper backends.

The table shows that the VCG backend is faster in most cases, but for three method, it is slower. We could not determine why the performance

was so bad for `_calc_fwding_ingress`. This method is not very complicated, but even with the SE backend it takes a long time to verify. Even more puzzling is the fact that `get_all_border_routers` took almost 12 minutes using the VCG backend. The method simply takes four lists and adds them to a resulting list. For `verify_mac` the performance difference is not big enough to be significant, as we tested the performance on a normal system with other processes, like a web browser running in the background. Thus, a small difference of a few seconds can also be caused by some other process using more resources in one test, but not the other. We consider such a small difference to be around five seconds.

Using these statistics, we can calculate some interesting percentages. Overall, the VCG backend was 286% faster than the SE backend; this was calculated including the cases where the VCG backend was actually slower. However, this number is skewed, because for `_egress_forward`, the VCG backend was 3667% faster. If we exclude this method, then the VCG backend was still 98% faster.

Generally, we can say that the SE backend has more reliable performance, because it is more directly related to how complicated a method is, whereas the VCG backend can seemingly randomly take a long time to verify a simple method like `get_all_border_routers`. For a majority of our methods, the VCG backend did perform significantly faster. In conclusion, we cannot say that one backend is objectively better, so we recommend working with both, as to be able to take advantage of the strengths of both and to try and avoid the weaknesses of either.

Symbolic Execution Backend: For the SE backend we looked at how much of the verification time was spent on verifying pure functions and predicates, which get referenced by a method, and how much time was spent on verifying the method itself. Table 4 shows the measures times for a majority of the methods that we verified.

For all the simpler methods, like getters, the time spent on verifying the functions and predicates dominates the time used to verify the method itself. When looking at more complicated methods, like `send` and `_egress_forward`, the time spent on both parts is comparable, but `_calc_fwding_ingress` is an outlier, as it requires about three times the amount of time to verify the method itself.

Method	Function + Predicate	Method
handle_extensions	16s	24s
_process_flags	1s	1s
verify_hof	103s	26s
_calc_fwding_ingress	492s	1518s
_validate_segment_switch	63s	4s
send	2360s	2810s
_egress_forward	3041s	2056s
_link_type	20s	6s
get_all_border_routers	4s	3s
inc_hof_idx	78s	111s
get_hof_ver	66s	19s
_get_hof_ver_normal	60s	6s

Table 4: Table showing the verification times for functions & predicates referenced by the method and the method itself.

5.4 General Insights

Finally, we will talk about some more general insights that we gained when working on this project. We always worked in a bottom-up manner and we can say that it worked very well, as we rarely had to go back to already verified methods to change a contract. Furthermore, collecting the path constraint for the I/O verification only worked well when working bottom-up and might have been quite difficult to gather when working top-down.

First, we verified memory safety, then progress and finally I/O behaviour and we believe that this was the only way, as any other way might have proven to be impossible or overly complicated, as we first need to have the contracts about permissions to even get a successful verification. It could be argued that progress could be verified after I/O behaviour, but we believe that it makes sense to first know that all methods make progress, so that we know that a packet eventually gets forwarded.

The state predicates we used became more complicated than anticipated. We think that it would have been better to use multiple smaller predicates to keep the contracts more readily comprehensible. Additionally, that might have improved the performance for unfolding expressions of these predicates. However, there is an advantage to the way we used state predicates as permission assertions are generally only included in one state predicate and

it was easy to keep them self-framing. If multiple smaller predicates are used, one needs to carefully separate them, to avoid having a permission to the same field in multiple predicates, as this could make the verification more complicated when permissions amounts are used. When a permission to the same field is included in multiple predicates, then a fraction needs to be used in each one and a method is only allowed to use all these predicates in a contract if the total permission amount to the field is lesser or equal than one. As an example, two predicates have a permission to field `x.f` in their contents. If a method then has a full permission to each predicate in its precondition, the permission amount to the field in both predicates must at most add up to one, e.g., `Acc(x.f, 1/4)` and `Acc(x.f, 1/6)`.

In our experience, the Nagini verifier has proved to fulfil completeness as long as wildcard read permissions are not used. While we had encountered a case where Nagini could not prove a condition, we later rewrote those contracts and no further issues appeared.

6 Conclusion & Future Work

In conclusion, we can say that we successfully verified almost all of the methods that we initially selected for verification. We completed the verification for memory safety and progress for all methods that we aimed for, but we could not finish the I/O verification, because of an extensive performance problem in the verifier. We identified 25 bugs in Nagini and Silicon and we determined that the verification condition generation backend is generally faster. Furthermore, we discovered that Nagini is incomplete when wildcard read permissions are used and we realised that verification performance can be improved greatly when unfolding expressions are separated into their own functions.

Once the performance problem of Nagini has been fixed, the I/O verification can be completed. Most of the groundwork is already laid out, even for the methods whose verification did not terminate. We wrote the contracts as well as we could, given that we could not actually get feedback on whether they are correct and sufficient.

Furthermore, the verification of the SCION router code can continue with the verification of the parsing methods. Currently, the specification expects certain invariants to be established by a successful parsing of a SCION packet and we simply assumed that the parsing could guarantee them. Now, one could verify the parsing implementation and actually write contracts which show that the parsing can guarantee these invariants. In the process of verifying the parsing implementation, one would find any bugs that result from the interaction between the parser and the rest of the code. If there are no bugs, then the router is correct for our scenario.

Another possibility to continue the verification effort of the SCION router code is to drop some or all of the assumptions that we made about the kinds of SCION packets that are received. Specifically, it would be of interest to verify the Router implementation with regards to an arbitrary path in the received packet. Furthermore, one could also drop the assumptions that no SCION extensions are used and verify the code for packets which are using one or more SCION extensions.

In terms of I/O behaviour, further verification could be performed by fully checking compliance of the I/O behaviour of the router with the SCION protocol.

List of Figures

Listings

List of Tables

References

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

[2] Chair of Programming Methodology, ETH Zurich, "VerifiedSCION." `http://www.pm.inf.ethz.ch/research/verifiedscion.html`. Accessed: 22.08.2018.

[3] M. Eilers and P. Müller, "Nagini: a static verifier for python," in *International Conference on Computer Aided Verification*, pp. 596–603, Springer, 2018.

[4] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, (New York, NY, USA), pp. 41–62, Springer-Verlag New York, Inc., 2016.

[5] V. Astrauskas, "Input-output verification in viper," Master's thesis, Department of Computer Science, ETH Zürich, 2016.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*, pp. 364–387, Springer, 2005.

[7] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[8] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames," *ACM Trans. Program. Lang. Syst.*, vol. 34, pp. 2:1–2:58, May 2012.

[9] P. Boström and P. Müller, "Modular Verification of Finite Blocking in Non-terminating Programs," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)* (J. T. Boyland, ed.), vol. 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 639–663, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[10] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *European Symposium on Programming Languages and Systems*, pp. 158–182, Springer, 2015.

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):

Static Verification of the SCION Router Implementation

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Forster	Sascha

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)**
Engelburg, 09.09.2018	S. Forster

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