ETH zürich

Specifying and Verifying the IO Behavior of the SCION Border Router

Master's Thesis

Lino Telschow

March 23, 2022

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

Department of Computer Science, ETH Zürich

Abstract

SCION is a future Internet Architecture that solves most of the problems of today's Internet. SCION implements a new forwarding mechanism that encodes the forwarding path of network packets in their header. The SCION border router is the SCION component that is responsible for receiving, processing, and sending network packets. Two important properties of router software implementations are that they are memory safe and only perform allowed IO operations.

In this work, we specify and verify the IO behavior of the Go implementation of the SCION border router. We use the Gobra verifier to verify memory safety for almost all methods of the border router implementation. Then, we specify an IO specification that captures the IO behavior of the border router. We use Gobra to verify that the border router implements the IO specification. This allows us to prove properties about the IO behavior of the router. For example, we prove that the packet processing methods of the router correctly read and return forwarding information encoded in the packet header. In the end, we measure the verification overhead for the memory verified code with and without IO specifications.

Acknowledgements

I would like to thank my thesis supervisors João Pereira and Felix Wolf for their support during this project. The weekly meetings and their constant feedback helped me to overcome any challenges that I faced. I also would like to thank Prof. Dr. Peter Müller for allowing me to work on this interesting thesis.

Contents

Contents **v**

1 Introduction **1**
1.1 Motivation and Goals 1
1.2 Contributions 2
1.3 Outline 2

2 Background **5**
2.1 SCION 5
2.2 VerifiedSCION 7
2.3 Gobra 7
2.3.1 Basic Annotations 8
2.3.2 Permissions 9
2.3.3 Predicates 11
2.3.4 Termination 12
2.4 Specifying IO Behavior 12
2.4.1 Motivating Example 13
2.4.2 Specifying and Verifying IO Behavior with Petri Nets 13

3 Verifying Memory Safety **17**
3.1 The Structure of the Border Router 17
3.1.1 Shared Data Structure 19
3.1.2 Packet Processing Pipelines 20
3.2 Verification Approach 24
3.2.1 Codebase 24
3.2.2 Bottom-up Approach 24
3.2.3 Specifying other Packages 24
3.2.4 Rewriting Code 24
3.3 Challenges 25
3.3.1 Aliased Slices 26

3.3.2 Verification Time 29

4 Specifying and Verifying IO Behavior 35
4.1 Guarded Event Systems 35
4.2 Guarded Event System of the Border Router 36
4.2.1 Events 36
4.2.2 State 38
4.2.3 Transitions 39
4.3 Encoding the Event System of the Border Router 40
4.3.1 State 40
4.3.2 Events 41
4.3.3 Transitions 42
4.4 Specifying and Verifying IO Behavior 45
4.5 Verified IO Properties 46
4.5.1 Processing Intra BFD Properties 46
4.5.2 Processing Inter BFD Properties 46
4.5.3 Processing OHP Properties 47
4.5.4 Processing SCION Properties 47

5 Evaluation 49
5.1 Verified Properties 49
5.2 Statistics 50
5.2.1 Verified Methods 50
5.2.2 Annotation Overhead 50
5.2.3 Verification Performance 52
5.3 Limitations 54

6 Conclusion 57
6.1 Related Work 57
6.2 Future Work 58

Bibliography 59

Chapter 1

Introduction

During the last decades, the Internet has become one of the most important communication systems in the world. Today, the Internet is so pervasive that society and businesses heavily rely on it.

Our current internet architecture consists of hundred thousands of interconnected autonomous systems (AS). These ASes are held together by a number of network protocols. Some of these protocols, like the Border Gateway Protocol (BGP), were designed in a time when no one could imagine how pervasive the Internet would be in the future. As such, these protocols have some flaws that are hard to fix. For example, BGP suffers from IP hijacking attacks. In this attack, an adversary hijacks an IP prefix and reroutes traffic destined to that IP prefix to itself. This allows the adversary to eavesdrop on the traffic or at least to disrupt operation of the service behind the hijacked IP prefix. Another issue in the BGP dominated Internet is that the senders have almost no choice over the paths their data is forwarded. To solve these and other issues, we need to replace the flawed network protocols or at least design new protocols that can be deployed parallel to the existing legacy Internet architecture. One possible future Internet architecture is the SCION Internet architecture [13].

The SCION Internet architecture [13] (stands for Scalability, Control, and Isolation on Next-Generation Networks) aims to solve the most daunting problems of our current internet architecture. SCION consists of different protocols that aim to replace legacy protocols. For example, SCION proposes with its RAINS name server an alternative to the DNS architecture. Furthermore, SCION implements its own border routers that aim to replace the flawed BGP border routers.

1.1 Motivation and Goals

In order that SCION can deliver all its benefits over the current Internet architecture, it is of crucial importance that the SCION protocols are correct from their high-

level designs down to their code-level implementations. To achieve this goal, the VerifiedSCION project [2] has been founded. VerifiedSCION is a joint research project of the Network Security Group, the Information Security Group, and the Programming Methodology Group at ETH Zürich.

The overall goal of this thesis is to specify and verify the IO behavior of the SCION border router. The border router is a crucial component of the SCION architecture, and it is responsible for receiving, processing, and sending network packets. This makes the border router the backbone of the SCION Internet architecture.

So far, some work has been done on verifying memory safety of selected parts of the SCION border router by using deductive verifiers like Gobra and Nagini [7, 8]. One goal of this thesis is to verify memory safety of the Go implementation of the SCION border router by using the Gobra verifier [15].

Another goal of this thesis is to identify IO events and an abstraction of the border router's state. Given these events and the abstract state, we define an IO specification that captures the IO behavior of the SCION border router.

We verify the IO behavior of the packet processing methods of the border router by using Gobra.

The last goal of this thesis is to evaluate our IO verification technique. Thus, we analyze the annotation overhead of added Gobra assertions and measure the verification time of single methods and the entire border router, respectively.

1.2 Contributions

The main contributions of this thesis are listed as follows:

- Verification of memory safety and termination for almost all methods in the SCION border router. To our knowledge, Gobra has not yet been used for codebases of this size.
- Definition of an IO specification that represents the IO behavior of the SCION border router.
- Verifying IO behavior of the packet processing methods of the border router.

1.3 Outline

In Chapter 2, we present some background about SCION and the Gobra verifier. Furthermore, we explain the IO verification approach from Penninckx et al. [12]. In Chapter 3, we first describe the structure of the SCION border router, and then we explain our verification approach. We conclude this chapter with a discussion of the challenges we faced during verification. In Chapter 4, we define an event system that describes the IO behavior of the border router and encode it in Gobra.

Then, we present our IO specification and verification approach. In Chapter 5, we discuss the evaluated properties and deliver some statistical data about the verification process. Finally, in Chapter 6, we conclude and give an outlook of possible future work.

Chapter 2

Background

In this chapter, we introduce the reader to the required background knowledge. In Section 2.1 we start with a short introduction to the SCION Internet architecture. In Section 2.2 we relate this thesis to the VerifiedSCION project. Then, in Section 2.3 we present the Gobra verifier and show small code examples to explain some of Gobra's features. We conclude this chapter with Section 2.4 that presents the IO verification approach from Penninckx et al. [12].

2.1 SCION

SCION [13] is a new internet architecture that has been designed to complement and replace some of our current internet protocols. Our existing internet infrastructure consists of thousands of autonomous systems (AS) that are interconnected with each other. These ASes use the Border Gateway Protocol (BGP) [14] to exchange status information and to update their forwarding tables. In this setting, the bigger ASes have more influence and determine how packets are forwarded. In general, a smaller AS can only use one forwarding path when it sends a packet to a specific target. This means that the sender has almost no control through which ASes its packets are forwarded. Furthermore, if a link on the path fails, then it can take minutes until BGP announces an alternative path to the affected ASes. Additionally, the current internet infrastructure suffers from other issues like BGP hijacking or DDoS attacks. SCION solves these issues by providing route control, isolation, and trusted end-to-end communication. SCION achieves this by implementing a new forwarding mechanism and by grouping the ASes in Isolation Domains (ISD). Instead of using routing tables and longest prefix matching, SCION encodes the forwarding path in the header of each packet. A path consists of up to three path segments. Each path segment is a list of AS hops. As such, these path segments specify through which ASes the packets are forwarded. The SCION protocol specifies how these path segments are discovered by the ASes and how the sender can combine them to construct valid forwarding paths. The SCION

IngressID:0 EgressID: 2 AS: A	IngressID: 2 EgressID: 5 AS: B	IngressID: 5 EgressID: 0 AS: C

Figure 2.1: A SCION path with one segments that consists of three hops. The hops specify a path from AS A to AS C.

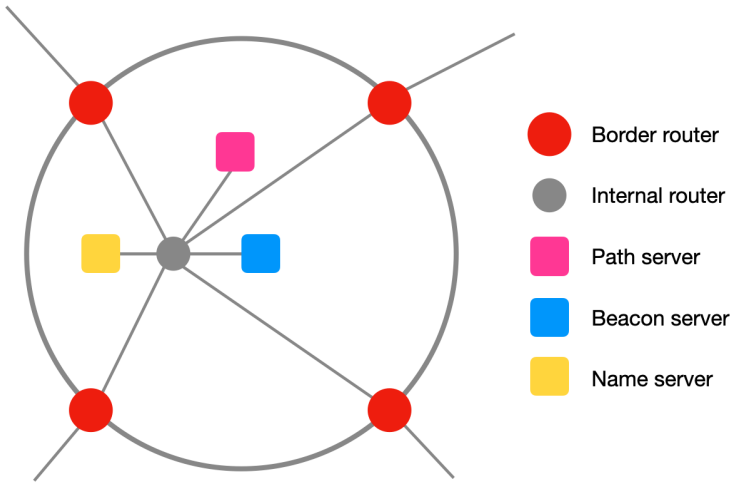

Figure 2.2: A simple SCION AS. The control plane consists of path servers, beacon servers, and name servers. The data plane consists of the border routers, that are depicted by the red circles on the border of the AS. In a real-world setting, each border router might have hundreds of links connecting it with neighboring ASes.

protocol guarantees that business relations between the ASes are preserved in all forwarding paths. Additionally, SCION uses cryptographic primitives to make the paths tamper-resistant.

Figure 2.1 shows a simplified SCION path that consists of three hops. Each hop has an IngressID, an EgressID, and an AS identifier. The IngressID specifies the interface ID over which the packet enters the AS. The EgressID specifies the interface ID over which the packet is forwarded to the next AS. For example, the hop of AS B specifies that the packet enters AS B via interface 2 and leaves the AS over interface 5. If the EgressID is 0, then the packet is forwarded within the AS to its destination.

The components of the SCION internet architecture can be grouped in a control plane and a data plane. The purpose of the control plane is to discover forwarding paths and to make them available to the ASes of the SCION network. On the other

hand, the data plane is responsible for forwarding packets in the SCION network.

The control plane consists of beacon servers, name servers, and path servers. The beacon servers are responsible for discovering paths. The name servers resolve domain names to SCION addresses. They are SCION's counterpart to DNS. Given a destination address, the path servers can be used to obtain paths to that destination.

The data plane consists of border routers (BR) that are responsible for receiving, processing, and sending packets in the SCION network. The border routers are placed at the edges of each AS, and as such, they are directly connected to the neighboring ASes.

Figure 2.2 shows a simple SCION AS with its control plane and data plane components.

In this project, we will focus only on the border router of the SCION architecture. For more details about the other components, we refer to the SCION book [13]

2.2 VerifiedSCION

VerifiedSCION is a joint research project of the Network Security Group, the Information Security Group, and the Programming Methodology Group at ETH Zürich. This project aims to verify the SCION protocol from its high-level design all the way down to its implementation. In the VerifiedSCION project, we focus on verifying safety, functional correctness, and security of the implementation of the SCION routers [2]. So far, we have worked on two theses whose goal was to verify memory safety of a sub-set of the SCION border router's methods [7, 8].

2.3 Gobra

Gobra [15] is a deductive verifier for Go programs. Gobra is developed by the Programming Methodology Group at ETH Zürich and is hosted as an open-source project on GitHub [1]. Gobra supports many Go constructs like channels and goroutines. This allows Gobra to reason about concurrent Go programs and to prove properties such as memory safety and data-race freedom. In order that Gobra can verify a Go program, the user must annotate the Go code with assertions such as preconditions, postconditions, and loop invariants. Gobra is a front-end of the Viper verification infrastructure [11]. As such, Gobra translates the annotated Go code to the Viper language. The encoded Viper program is then verified by the Viper back-end. If the verification process yields an error, then the error is translated to the Gobra source code level such that the user can locate the issue. The Viper toolchain supports two verification back-ends. The verification condition generation back-end is based on Boogie [5] and uses the Z3 SMT solver [10] to discharge proof obligations. On the other hand, the symbolic

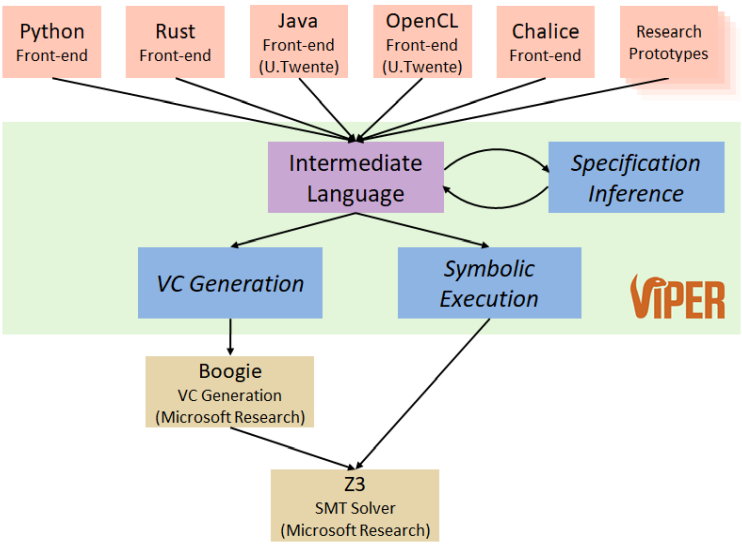

Figure 2.3: The Viper verification infrastructure. The figure is taken from the Viper project website [3].

execution back-end directly translates the Viper program to a set of inputs for the Z3 SMT solver. Figure 2.3 shows the Viper infrastructure with some of its front-ends.

In the following, we will present some of Gobra's features. For an explanation of the more advanced features, we refer to the Gobra paper [15].

2.3.1 Basic Annotations

We explain the basic annotation keywords by showing a simple example. Figure 2.4 shows a Go function that computes the square of a positive number. The `requires` keyword denotes a precondition. Preconditions must be satisfied by the callers of a method. For each call of a method, Gobra checks if the preconditions are satisfied. If a precondition is not satisfied, then Gobra stops and shows the user which method call fails. For the `square` function, we restrict the input arguments to positive integers. The postconditions of methods are annotated with the `ensures` keyword. Postconditions describe the program's state at the end of a method call, given that the method was called in a state where its preconditions were satisfied. For the `square` function, the postcondition states that the returned number is the squared value of its input argument. Like other deductive verifiers, Gobra uses loop invariants to specify and verify the behavior of loops. A loop invariant is an assertion that holds before the loop block, after each loop iteration, and after the loop block. In our `square` function, the first loop invariant is required to describe the range of the `i` variable. The second loop invariant specifies the state of the `res` variable. With these two loop invariants, Gobra can deduce the postcondition

```
requires arg > 0
ensures res == arg * arg
func square(arg int) (res int) {
        res := 0
        invariant 0 <= i && i <= arg
        invariant res == i * arg
        for i := 0; i < arg; i++ {
                res = res + arg
        }
}
```

Figure 2.4: A Go method that is annotated with Gobra assertions.

after the loop block.

2.3.2 Permissions

To reason about heap-allocated data structures that can be accessed by multiple methods simultaneously, Gobra needs to keep track of access permissions. Gobra supports both read access and write access permissions. This means if no method has write permission to a data structure, then multiple methods can read it. We use fractional permission amounts to distinguish between write and read permissions. Write permission corresponds to the permission amount of `1/1`. If we have a fractional permission amount in the range $]0, 1[$, then we have read permission. If a method has zero permission amount, then it has no access to the heap location. In the following, we will show some methods that access heap-allocated data.

The `inc` method in Figure 2.5 is an example of a heap manipulating method. The `acc(ptr)` assertion in the precondition states that the method requires write access to the heap location denoted by pointer `ptr`. To call the method, the caller requires access permission to the pointer `ptr`. In this case, the `client` method allocates the pointer argument and has write permission to its heap location. The `@` symbol after the x variable is required to let Gobra know that the variable is allocated on the heap. Thus, Gobra keeps track of access permissions to the x variable. When calling the `inc` method, the caller gives up access permission to the heap location of `x` and transfers it to the method. The first postcondition specifies that the method gives back the access permission to the caller. The `old` keyword in the second postcondition refers to the value stored at the `ptr` location before the method is executed. The second postcondition specifies that the method updates the value at the `ptr` location by one. The `readPtr` method requires only read permission to its `ptr` argument. Read permission is denoted by a fractional permission amount. This means the caller of `ptr` needs to have an access permission amount of at least `1/10` to call the `readPtr` method. In our example, the `client` method has access permission amount of `1/1` to the heap location of `x` and calls `readPtr`. During the `readPtr` is executed the `client` method keeps an access permission amount of `9/10` and thus, it can deduce that the value of `x` does not change since no other thread can obtain write permission to `x`. After the `readPtr` returns, the `client` method has again an access permission amount

```
1  requires acc(ptr)
2  ensures acc(ptr)
3  ensures *ptr == old(*ptr) + 1
4  func inc(ptr *int) {
5          *ptr = *ptr + 1
6  }

7  requires acc(ptr, 1/10)
8  ensures acc(ptr, 1/10)
9  func readPtr(ptr *int) int {
10         return *ptr
11 }

12 requires acc(ptr)
13 ensures acc(ptr)
14 func doSomething(ptr *int)

15 func client() {
16         x@ := 0 // at this point we have full access permission (1/1)
17         inc(&x) // gives up all access permission
18         assert x == 1 // verifies
19         y := readPtr(&x) // gives up 1/10 of access permission
20         assert x == 1 // verifies
21         doSomething(&x) // gives up all access permission
22         assert x == 1 // fails
23 }
```

Figure 2.5: Some Gobra methods with read and write permissions.

of `1/1` to the heap location of `x`. When the `client` method calls the `doSomething` method, then it gives up all access permission to the heap location of `x`. Thus, it can no longer deduce that the value of x remains the same after `doSomething` returns. Thus, the last assert statement fails.

Quantified Permissions

So far, we have discussed how to specify permissions for pointers of primitive types. In regular Go programs, we often use data structures with an arbitrary number of heap locations, such as slices. If a slice is allocated on the heap, then we need a mechanism to keep track of the permissions of its locations. For this purpose, we can use quantified permissions to quantify over an arbitrary number of heap locations.

The example in Figure 2.6 presents the usage of quantified permissions. In the pre- and postconditions, the method specifies that it requires write access to all slice entries. The loop that sets all entries to zero also requires an explicit loop invariant that propagates the write permission from the method's precondition to the loop body. Without this invariant, the loop body has no access to the entries of slice `s`. After the loop is executed, the write permission to the slice entries is given back from the loop body to the method. This allows the method to deduce its postcondition.

```
requires forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
ensures forall k int :: 0 <= k && k < len(s) ==> acc(&s[k]) && s[k] == 0
func reset(s []int) {
        invariant 0 <= i && i <= len(s)
        invariant forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
        invariant forall k int :: 0 <= k && k < i ==> s[k] == 0
        for i := 0; i < len(s); i++ {
                s[i] = 0
        }
}
```

Figure 2.6: A Gobra method that uses quantifiers to reason about access permissions of slice entries.

```
type Foo struct {
        bar int
        baz []byte
}

pred (f *Foo) Mem() {
        acc(f) &&
        f.bar > 0 &&
        forall i int :: 0 <= i && i < len(f.baz) ==> acc(&(f.baz)[i])
}

// verifies
requires f.Mem()
ensures f.Mem()
func client1(f *Foo) {
        unfold f.Mem()
        f.bar = 42
        assert forall i int :: 0 <= i && i < len(f.baz) ==> acc(&(f.baz)[i])
        fold f.Mem()
}

// fails
requires f.Mem()
ensures f.Mem()
func client2(f *Foo) {
        unfold f.Mem()
        f.bar = -1
        fold f.Mem() // fold fails
}
```

Figure 2.7: A Gobra code snippet that illustrates the usage of predicates.

2.3.3 Predicates

Like other program verifiers, Gobra supports predicates. Predicates are constructs that allow us to define parameterized assertions. Furthermore, predicates can be defined recursively. Predicates are used to reason about unbounded and possibly recursive defined heap data structures, such as linked lists.

In Figure 2.7, we show a type `Foo` that consists of two fields. The `bar` field is an integer, and the `baz` field is a slice of bytes. The `Mem` predicate takes as argument a `Foo` pointer and it contains three assertions. The `acc(f)` assertion states that we have access to the locations `&f.bar` and `&f.baz`. The second assertion states that the value `f.bar` is positive. The third assertion states that we have permission to all entries in the `f.baz` slice. Additionally, there are two client methods that

```
requires arg > 0
ensures res == arg * arg
decreases
func square(arg int) (res int) {
        res := 0
        invariant 0 <= i && i <= arg
        invariant res == i * arg
        decreases arg - i
        for i := 0; i < arg; i++ {
                res = res + arg
        }
}
```

Figure 2.8: The `square` method from Figure 2.4 annotated with the `decreases` keyword.

use a `Foo` pointer and its `Mem` predicate. To access the assertions of a predicate, the clients have to unfold it. The unfold operation replaces the predicate with the assertions in its body. After unfolding the predicate, `client1` updates the `bar` field and it checks if it has access to the entries of the `baz` slice. Before returning, `client1` folds the `Mem` predicate of `f`. The fold operation does the inverse of the unfold operation. This means it replaces the assertions of the predicates body with an instance of the predicate. The fold operation fails if an assertion of the predicate does not hold. A failing fold operation can be seen in method `client2`. In `client2` the fold operation fails since the second assertion is violated.

2.3.4 Termination

Another desirable property of programs is termination. To prove termination of programs, Gobra has the `decreases` keyword. This keyword is used to annotate methods that terminate. If a method is annotated with the `decreases` keyword, then Gobra checks if it consists only of statements that terminate. This means the method can only call other methods that terminate, and it is not allowed to have nonterminating loops.

Figure 2.8 shows the `square` method from Figure 2.4 annotated with the `decreases` keyword. The `decreases` assertion in the method's contract states that the method terminates. Additionally, the loop is also annotated with a `decreases` assertion and a lifetime measure. The lifetime measure is a numerical expression that must decrease in each loop iteration. This allows Gobra to deduce that the method terminates.

2.4 Specifying IO Behavior

In this thesis, we specify and verify the IO behavior of the SCION border router. In the context of this thesis, IO behavior describes how the SCION border router interacts with its network interfaces. This means it describes how the border router receives, processes, and sends network packets. Specifying and verifying IO behavior allows us to prove that no unintended IO operation is executed and that

```
1  func incServer() {
2          x := receiveInt()
3          sendInt(x + 1)
4  }
```

Figure 2.9: A simple method the receives, increments, and sends an integer.

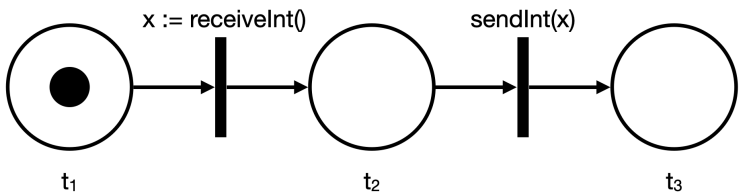

Figure 2.10: Petri net representation of the `incServer` method in Figure 2.9.

all performed IO operations adhere to the program's specification. We specify and verify IO behavior by using a technique inspired by Penninckx et al. [12]. In the following, we will start with a motivating example that has a straightforward IO behavior, and later we will present the IO specification and verification approach from Penninckx et al. [12] and apply it to our motivating example.

2.4.1 Motivating Example

Consider the method `incServer` in Figure 2.9. The intended IO behavior of this method is that it first receives an integer and afterwards, it increments the integer by one and sends it. This means the input of the send operation depends on the output of the receive operation that happened before. In this simple example, it is obvious that the `incServer` method behaves according to its intended IO behavior. However, we need a methodology to specify and verify IO behavior for more complex programs, like the SCION border router.

2.4.2 Specifying and Verifying IO Behavior with Petri Nets

The approach from Penninckx et al. [12] models IO operations as transitions of tokens in a Petri net. Based on this Petri net representation, they describe how such IO behavior can be encoded in pre- and postconditions of methods. Their idea is to constrain the IO behavior of a program by encoding the allowed IO operations in the precondition of the program's methods.

In Figure 2.10 we show a simple Petri net representing the IO behavior of the `incServer` method. The circles in the figure are called places, and the dot in the first place is the token. The vertical stretched rectangles in the figure represent the IO operations. These IO operations have input and output arguments. In this example, the `x` variable is the output argument of the `receiveInt` operation and the input argument of the `sendInt` operation. To execute an IO operation in

this Petri net representation, we require that the IO operation has a token in all places of its preset. The preset is the set of places attached to the left-hand side of the IO operation. In this example, we require a token in place t_1 to execute the `receiveInt` operation. After the `receiveInt` operation has been processed, the variable `x` contains the received integer and a token is moved to all places in the postset of the `receiveInt` operation. The postset is the set of places to which an IO operation points to. Here, the postset of the `receiveInt` operation consists only of the place t_2. After the `receiveInt` operation has been executed, the token is in place t_2 and the `sendInt` operation can be executed next.

In the following, we show how IO behavior can be encoded in pre- and postconditions of methods. The idea of the paper is to constrain the IO behavior of methods by encoding IO permissions in the preconditions of the methods. In the paper, they use predicates to encode IO permissions. Given that the IO permissions of all methods can be encoded with predicates, we can use an arbitrary verifier that supports predicates and prove that a real-world implementation of the program adheres to its IO specification.

The methodology consists of the following three steps:

(1) First, we need to identify our program's basic input and output operations. These operations are called the BIO operations. The BIO operations are the building blocks of all other IO operations. Depending on the code base, BIO operations are either operations of the OS kernel or methods of some library. In our example in Figure 2.9 we assume that `receiveInt` and `sendInt` are the BIO operations.

(2) Second, we need to define the IO permissions for the BIO operations. We do this by defining abstract predicates. Additionally, we define a predicate for the token.

    Figure 2.11 shows the predicates for the `receiveInt` and `sendInt` BIO operations. All BIO predicates have two place arguments. These place arguments model the movement of the token in the Petri net representation. In between the place arguments, the BIO predicates contain the input and output arguments of their corresponding BIO operations.

(3) Finally, we use the token predicate and the predicates of the BIO operations to specify the IO behavior of methods. We first specify the BIO operations, and then we proceed with the composed methods that call BIO operations in their bodies.

    Figure 2.12 shows how we can specify the IO behavior of the `receiveInt` method. We use the `?t1` pattern to bind the current place of the caller of `receiveInt` to the variable `t1`. Furthermore, the `?res` and `?t2` patterns in the receive predicate bind the output value and the output place to the variables `res` and `t2`. If the receive method terminates, then the postcondition ensures

```
// token predicate
pred token(p Place)

// BIO predicates
pred receiveInt_p(pre Place, x int, post Place)
pred sendInt_p(pre Place, x int, post Place)
```

Figure 2.11: The predicates for the `receiveInt` and `sendInt` BIO operations.

```
requires token(?t1) && receiveInt_p(t1, ?res, ?t2)
ensures token(t2)
func receiveInt() int
```

Figure 2.12: The `receiveInt` BIO operation with its contract.

```
pred incServer_p(t1 Place, t2 Place) {
        receiveInt_p(t1, ?x, ?t3 ) &&
        sendInt_p(t3, x + 1, ?t2)
}

requires token(?t1) && incServer_p(t1, ?t2)
ensures token(t2)
func incServer() {
        unfold incServer_p(t1, ?t2)
        // token(t1) && receiveInt_p(t1, ?x, ?t3 ) && sendInt_p(t3, x + 1, ?t2)
        x := receiveInt()
        // token(t3)  && sendInt_p(t3, x + 1, ?t2)
        sendInt(x + 1)
        // token(t2)
}
```

Figure 2.13: The `incServer` method with its IO specification.

that the token ends up in some place `t2`. For the send method, we can specify the IO behavior similarly.

In the end, we specify the IO behavior of the `incServer` method from Figure 2.9. As earlier mentioned, the `incServer` method receives an integer, increments it by one, and then it sends the updated value. Figure 2.13 shows how we can specify this behavior. We first define the `incServer_p` predicate that uses in its body the `receiveInt_p` and `sendInt_p` predicates. As such, the `receiveInt_p` predicate specifies the IO behavior of the `incServer` method. The `incServer_p` predicate is then used in the contract of the `incServer` method. In order to execute the `receiveInt` and `sendInt` operations in the body of `incServer`, the `incServer_p` predicate needs to be unfolded such that its definition is available and such that a verifier can deduce that the sequence of IO operations can be executed. The comments in the body of the `incServer` method show the IO permissions before and after the two IO operations.

Chapter 3

Verifying Memory Safety

In this chapter, we explain how we verified memory safety of the SCION border router. In Section 3.1, we explain the structure of the border router by showing its concurrency patterns and some of its packet processing pipelines. Next, in Section 3.2, we outline our verification approach. Finally, in Section 3.3, we discuss challenges that we faced during verifying memory safety.

3.1 The Structure of the Border Router

As mentioned in the background section, the SCION border router is the backbone of the SCION protocol. The border router is responsible for receiving, processing, and sending packets. The border router can receive packets from internal routers within the AS or from border routers of neighboring ASes. Since SCION encodes the forwarding path in the packet header, the border router needs to look at the path in the packet header and determines the next hop. In the following, we show the structure of the border router.

Figure 3.1 illustrates at an abstract level how a border router with three interfaces

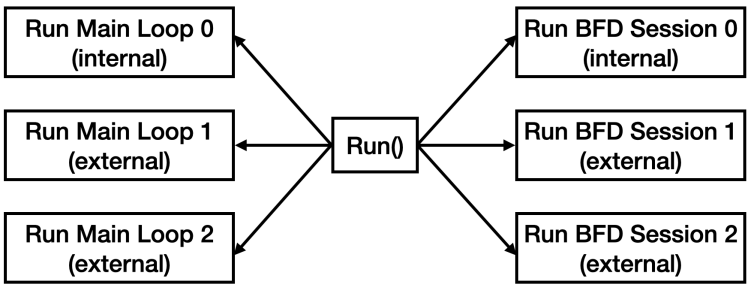

Figure 3.1: Launched threads for a border router with three interfaces. The internal interface connects the border router with its AS and has identifier 0. The two other interfaces represent connections with border routers of two neighboring ASes.

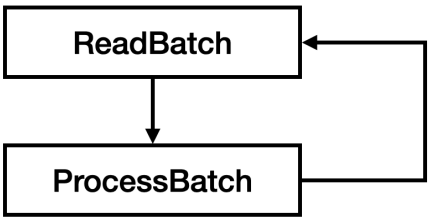

Figure 3.2: The main loop of the border router. For each interface, the border router runs this loop in a separate thread.

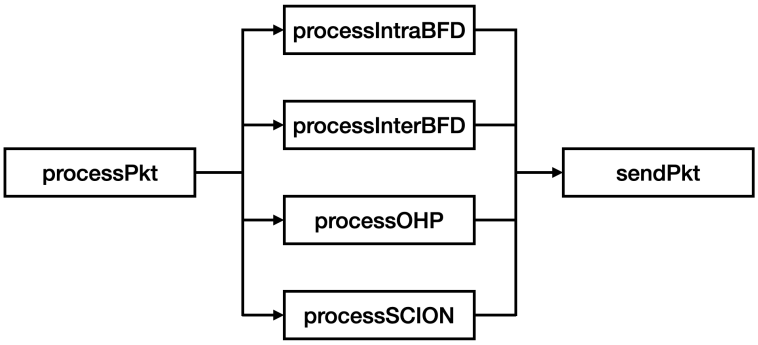

Figure 3.3: For each packet processed by the border router, the `processPkt` method calls one of the four packet processing methods. Note, the `sendPkt` method does not exist in the original SCION code, we introduce it here to simplify the explanation.

launches threads. For each interface, the border router launches two new threads. One of these threads runs the main loop of the border router, and the other one runs an instance of the BFD [9] protocol. The main loop is responsible for processing and sending network packets that are received from the interface. The BFD protocol is used to observe the state of a connection between two routers. As such, the BFD protocol can detect link failures. The SCION implementation of the BFD protocol is defined in its own package. In the following, we will focus on the structure of the border router's main loop.

The actions of the main loop are depicted as in Figure 3.2. The main loop first reads a batch of packets and afterwards processes each packet in the batch before it reads the next batch. For each packet in the batch, the ProcessPkt method in Figure 3.3 is invoked.

The border router distinguishes between these four types of packets:

- **Intra BFD:** A BFD packet that is sent from a node within the AS of the border router. Intra BFD packets are forwarded to a BFD session thread that executes an instance of the BFD protocol.
- **Inter BFD:** A BFD packet that is sent from a border router of a neighboring AS. Like in the case of the intra BFD packet, inter BFD packets are forwarded

```
1  type processResult struct {
2          EgressID uint16
3          OutConn  BatchConn
4          OutAddr  net.Addr
5          OutPkt   []byte
6  }
```

Figure 3.4: The definition of the `processResult` type.

to a BFD session thread that processes them according to the BFD protocol.

- **SCION:** A regular SCION packet. The border router processes the SCION packet and forwards it to the next AS, or if the packet reaches its destination AS, then it resolves the target address of the packet and forwards it to the next hop within its AS.
- **OHP:** A one-hop SCION packet. This packet has the same header fields as the regular SCION packet, but its path consists of only one hop entry. This means OHP packets are sent from one AS to a neighboring AS, but no further.

Figure 3.3 depicts the four different packet processing methods. Each received packet is decoded and then processed by one of these methods. A packet that cannot be decoded is dropped. All these four branches process the packet and return a `processResult` struct. The `processResult` struct captures all possible results of the four branches.

Figure 3.4 shows the definition of the `processResult` struct. If a packet is of type intra BFD or inter BFD, then the returned `processResult` is always a struct with uninitialized fields. The processing of intra BFD and inter BFD packets does not require a result since BFD packets are processed by the BFD session threads. If the `sendPkt` method in Figure 3.3 receives a struct with uninitialized fields as argument, then it will return without sending anything. If the `processOHP` and `processSCION` methods correctly process a packet, then they serialize the processed packet to the `OutPkt` field. Additionally, the `EgressID`, `OutConn`, and `OutAddr` fields contain valid entries. These entries are required for sending the packet in the SendPkt method. If the processing of an OHP or SCION packet fails, then the returned `processResult` is a struct with uninitialized fields, or it contains an encoded SCMP packet in the `OutPkt` field (SCMP is the SCION equivalent to ICMP in IP). If the `processResult` contains an SCMP packet, then the packet is sent by the `SendPkt` method.

3.1.1 Shared Data Structure

As mentioned earlier, the SCION border router is a multi-threaded program that executes multiple packet processing loop instances in parallel. The border router has one instance of the heap-allocated `DataPlane` struct that is shared among all packet processing threads.

```
1  type DataPlane struct {
2          external          map[uint16]BatchConn
3          linkTypes         map[uint16]topology.LinkType
4          neighborIAs       map[uint16]addr.IA
5          internal          BatchConn
6          internalIP        net.IP
7          internalNextHops  map[uint16]net.Addr
8          svc               *services
9          //macFactory      func() hash.Hash
10         bfdSessions       map[uint16]bfdSession
11         localIA           addr.IA
12         mtx               sync.Mutex
13         running           bool
14         Metrics           *Metrics
15 }
```

Figure 3.5: The `DataPlane` struct is shared between all instances of the main processing loop. This struct contains the configuration of the border router.

Figure 3.5 shows the `DataPlane` data structure that is shared among all instances of the main processing loop. In the following, we explain the most interesting fields of the `DataPlane` struct. The `external` map maps an interface identifier to its `BatchConn` struct. These `BatchConn` structs represent network endpoints that are connected to border routers of neighboring ASes. The `internal` field represents the network endpoint that connects the border router with its local AS. The `neighborIAs` map is used to look up the ISD and AS identifiers for a border router of a neighboring AS. The `internalNextHops` map contains network addresses.

3.1.2 Packet Processing Pipelines

In this subsection, we look at the packet processing pipelines for the inter BFD and the SCION packets. We omit the explanation of the pipelines of the intra BFD and OHP packets since they are similar to the pipelines of the inter BFD and SCION packets, respectively.

Inter BFD Pipeline

Before we explain the inter BFD pipeline, recall that the border router runs for each neighboring AS, an instance of the BFD protocol (see Figure 3.1). The pipeline for inter BFD packets is straightforward. It consists of two operations. First, a decoding method is called that decodes the BFD packet. Then, the decoded BFD packet is sent via a channel from the packet processing thread to its intended BFD session thread. Afterwards, the BFD packet is processed by the BFD session thread. Figure 3.6 shows the BFD processing pipeline.

SCION Packet Pipeline

In the following, we will look at the pipeline of a SCION packet. Before a SCION packet is processed, the border router creates a new instance of the

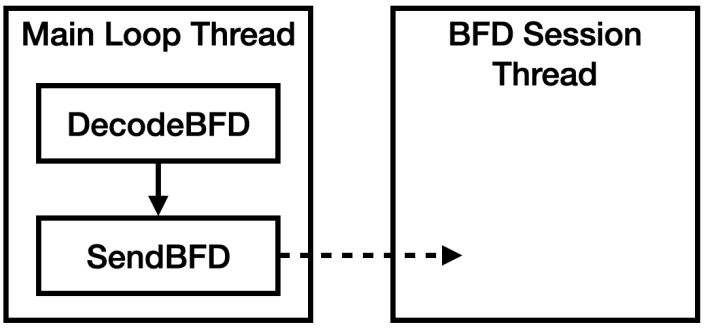

Figure 3.6: The processing pipeline of an inter BFD packet. The dashed arrow represents the sending of the BFD packet to the BFD session thread.

```
1  type scionPacketProcessor struct {
2          d *DataPlane
3          ingressID uint16
4          rawPkt []byte
5          scionLayer *slayers.SCION
6          origPacket []byte
7          buffer gopacket.SerializeBuffer
8          path *scion.Raw
9          hopField *path.HopField
10         infoField *path.InfoField
11         segmentChange bool
12 }
```

Figure 3.7: The definition of the `scionPacketProcessor` struct.

`scionPacketProcessor` struct (Figure 3.7 shows its definition). This struct contains important fields that are accessed by the packet processing methods. For example, the `rawPkt` field contains a byte representation of the SCION packet, and the `origPacket` contains a copy of the unmodified original packet. Additionally, the `scionPacketProcessor` struct contains a pointer to the heap-allocated `DataPlane` struct. This pointer is used to look up global properties of the border router that are shared with all main loop instances.

After the `scionPacketProcessor` struct has been created, the SCION packet pipeline is executed. Figure 3.8 shows a simplified representation of the SCION packet pipeline, which omits some methods.

As depicted in Figure 3.8, all methods in the pipeline use the `scionPacketProcessor` object as receiver. The pipeline begins with executing the `parsePath` method that decodes the AS path. Afterwards, methods are called that check if the packet has the correct length and a if it has a valid MAC entry (`validPktLen` and `verifyCurrentMAC` methods). In the middle of the pipeline, there is a case distinction. If the packet reaches its destination AS, then the packet's address is resolved such that it can be sent to its destination within the AS. If the AS of the border router is just another hop in the packet's path, then some packet fields are updated by the `doXover` method, and afterwards, the packet is forwarded to the

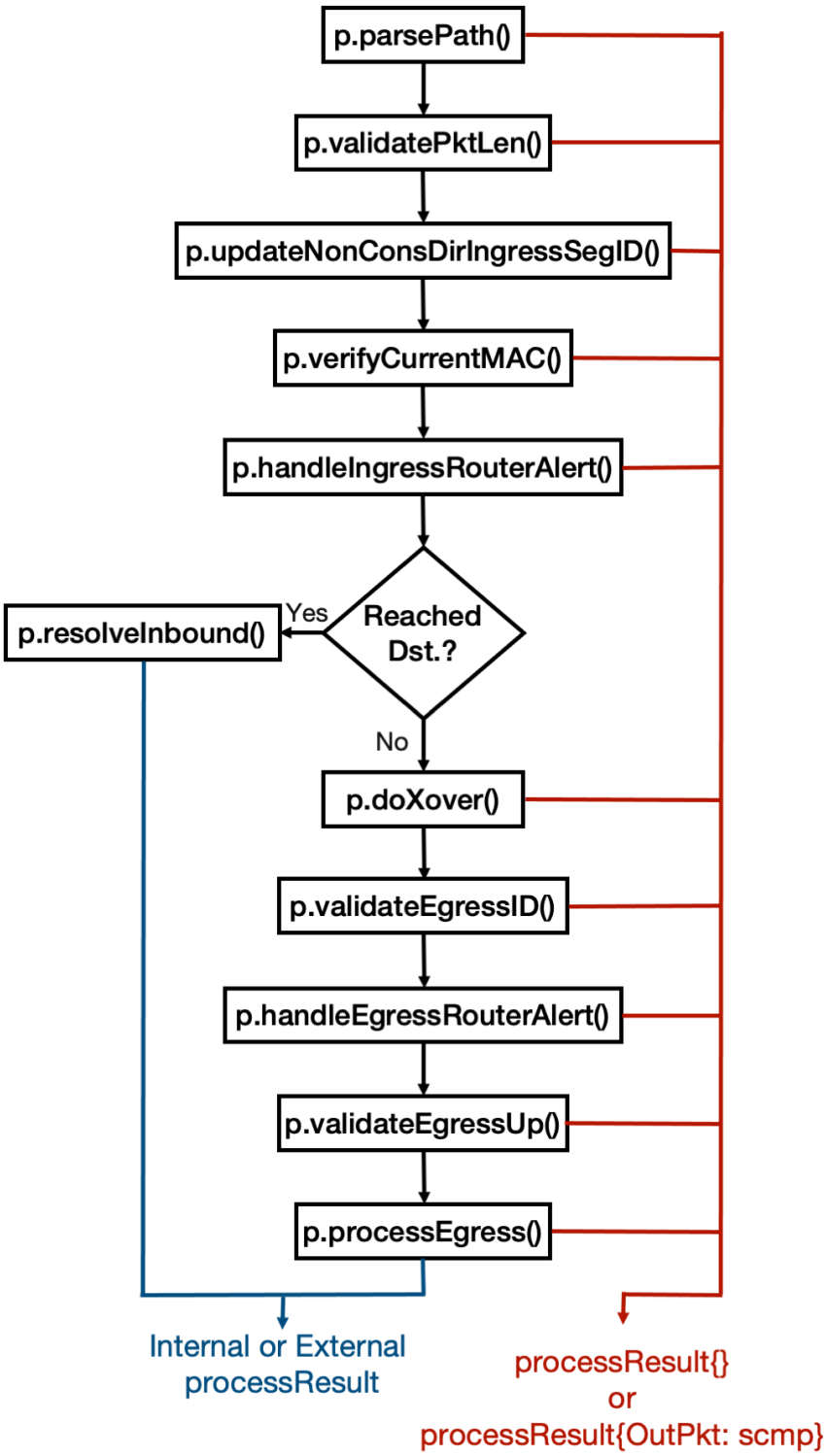

Figure 3.8: The processing pipeline of a SCION packet. The receiver object `p` is an instance of the `scionPacketProcessor` type.

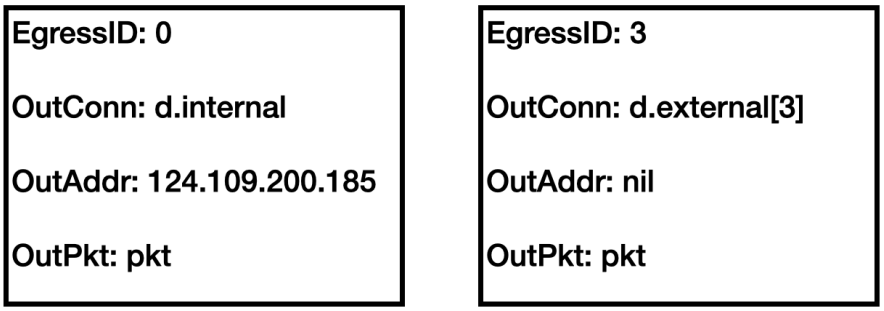

Figure 3.9: Examples for two possible `processResult` values that can be returned by the SCION packet processing pipeline if no error or SCMP error occurs.

next neighboring AS on the packet path. If the border router cannot forward the packet to the next AS then it will forward it to another border router within its local AS. This case happens if the border router does not have have a connection to the AS of the next hop. The processing can fail at any stage in the pipeline. However, under any circumstances, the pipeline will return a `processResult` value.

In case the pipeline does not run into an error, then the returned `processResult` value embeds the processed packet in the `OutPkt` field and the other fields of the `processResult` value are set according to one of the following cases:

- **Internal:** The `processResult` value encodes that the processed packet is internally forwarded. An internal `processResult` value has an `EgressID` of 0 and the `OutConn` field is set to the internal network connection that connects the border router with its local AS. Additionally, the `OutAddr` contains the IP address of the destination within the AS. An example of such a `processResult` can be seen in Figure 3.9 on the left.

- **external:** The `processResult` value encodes that the processed packet is forwarded to a neighboring AS. An external `processResult` Contains a positive nonzero EgressID that specifies the interface over which the packet is forwarded. The `OutConn` field contains the network endpoint over which the packet is sent and the `OutAddr` is set to nil. An example of such a `processResult` can be seen in Figure 3.9 on the right.

If the pipeline runs into an error, then the `processResult` value is a struct with uninitialized fields or if the error is an SCMP error, then the the `processResult` contains a serialized SCMP packet in the `OutPkt` field.

After the SCION packet is processed, the resulting `processResult` is handed to the `sendPkt` method as depicted in Figure 3.3. This method will then send the packet if the `processResult` is not empty.

3.2 Verification Approach

This section presents the codebase, our verification approach and explains how we have rewritten Go code fragments that are not supported yet by Gobra.

3.2.1 Codebase

The SCION border router is implemented in the `router` package. The files of the `router` package are located in the `pkt/router` directory of the SCION code base. In this project we verified the `dataplane.gobra`, `svc.gobra`, and `metrics.gobra` files that are contained in the `pkt/router` directory.

3.2.2 Bottom-up Approach

When we started the project, we considered two possible approaches of how we could verify the methods of the border router. We considered both a top-down and a bottom-up approach. The top-down approach starts with verifying the parent methods in the call graph of the border router. The top-down approach has the disadvantage that we need to make assumptions about the pre- and postconditions of the called methods. This is challenging for big methods that have many dependencies. For example, it can happen that we verified all callers of a method, and when verifying the method itself, we find out that verifying it requires a stronger precondition. Consequently, all callers have to be reverified again. In contrast, the bottom-up approach starts the verification process at the leaf methods in the call graph. As such, the callers of the method know exactly the contracts of the callees. Thus, we decided to use the bottom-up approach.

3.2.3 Specifying other Packages

The SCION border router depends on a lot of other packets. The `dataplane.go` file that contains all the methods of the border router imports 31 different packages. Ten of these packages are packages of the Go standard library, four are third-party libraries, and the other 17 are packages of the SCION code base. Verifying all these dependencies would have been out of scope of this project. For the dependencies from third-party libraries, we specified trusted contracts for methods and interfaces. Specifying trusted contracts requires caution since these contracts correspond to assumptions under which verification holds. For the dependencies of the SCION code base, we either specified trusted contracts or verified single methods.

3.2.4 Rewriting Code

The codebase of the border router has a considerable size and contains some code fragments that cannot be verified with Gobra. In this subsection, we explain how we have rewritten unsupported code fragments.

```
1 func mapIterator(m map[uint16]string) {
2         for k, v := range m {
3                 // do something with the entries
4         }
5 }
```

Figure 3.10: A method the iterates over the entries of a map.

Closures

Gobra does not yet support closures. One of the challenges of closures is that they can capture data structures of the closure's environment and modify them when the closure is executed. In the original border router code, the whole main loop thread is defined as a single closure. To verify the main loop, we converted the closure to a regular Go method. When the closure is converted to a regular Go method, we add additional parameters to the converted closure method to pass the previously captured data structures as arguments of the method. Like a Go closure, the converted closure method can be launched as a separate thread using the `go` keyword.

Continue and Break

At the moment, Gobra does neither support Go's `continue` nor its `break` keyword. We adapted loops and switch statements that use these keywords.

Range

Another Go feature that is not supported yet is the `range` operator. The `range` operator is commonly used to define loops that iterate over maps or lists.

Figure 3.10 shows a loop that uses the range operator to iterate over a map of type `map[uint16]string`. Since the index set of this map is the set of integers in the range from 0 to $2^{16} - 1$ we can replace the `range` loop by the loop shown in Figure 3.11. The loop in Figure 3.11 is semantically equivalent to the original loop but less performant since it iterates over the whole index set. This transformation only works if we have a map with a finite index set. Fortunately, in the border router code, all maps are indexed by the `uint16` type. This means we could apply this transformation to any `range` loop in the border router that iterates over a map.

3.3 Challenges

This section presents some of the most interesting challenges we faced while verifying the border router. In Subsection 3.3.1, we discuss the difficulties we faced when we verified code that uses aliased slices. In Subsection 3.3.2, we show the challenges we faced with long verification times.

```
1  requires acc(m)
2  ensures acc(m)
3  decreases
4  func mapIterator(m map[uint16]string) {
5          mapKeyUpperBound := 65536 // 2^16

6          invariant 0 <= k && k <= mapKeyUpperBound
7          invariant acc(m)
8          decreases mapKeyUpperBound - k
9          for k := 0; k < mapKeyUpperBound; k++ {
10                 key := uint16(k)
11                 v, isEntry := m[key]
12                 if isEntry {
13                         // do something with the entries
14                 }
15         }
16 }
```

Figure 3.11: Like the loop in Figure 3.10, this loop iterates over all entries in the map `m`, but without using the range operator. The loop iterates over the whole key set of the map and checks if there is a value for each key. If `isEntry` is true, then we have a valid entry that can be used in the if clause.

3.3.1 Aliased Slices

In this subsection, we show the challenges we faced when verifying aliased slices. Before discussing the challenge in detail, we first explain the concept of magic wands.

Magic Wands

The assertion `A --* B` denotes a magic wand. A magic wand consists of a left-hand side (`A`) and a right-hand side (`B`). Both `A` and `B` are resources. The magic wand is also a resource that can be used to obtain resource `B` given that we have resource `A`. For example, `B` could be the permission to a binary tree `T` and `A` could be the permission to a subtree `T'` of the tree `T`. Then the magic wand `A --* B` represents the permission to all nodes of the tree `T` that are not part of the subtree `T'`. Given that we have the permission resource `A` (permission to the nodes of subtree `T'`) and the magic wand `A --* B` (represents permission to all nodes of `T` that are not in `T'`), we can apply the magic wand to obtain permission `B` (access to the entire tree `T`).

Challenge

The term aliasing describes a setting where a heap-allocated data structure has multiple pointers pointing to it. In such a situation, the data structure can be updated by any of its pointers. This makes keeping track of the state of the data structure challenging. In systems programming languages like C and Go, aliasing is commonly used to optimize performance-critical code.

The SCION border router is designed to achieve high throughput and low latencies. As such, aliasing is heavily used to keep the processing overhead as low as

```
1  type SCION struct {
2          EmbeddedBaseLayer layers.BaseLayer
3          Version uint8
4          TrafficClass uint8
5          FlowID uint32
6          NextHdr common.L4ProtocolType
7          HdrLen uint8
8          PayloadLen uint16
9          PathType PathType
10         DstAddrType AddrType
11         DstAddrLen AddrLen
12         SrcAddrType AddrType
13         SrcAddrLen AddrLen
14         DstIA addr.IA
15         SrcIA addr.IA
16         RawDstAddr []byte
17         RawSrcAddr []byte
18         Path Path
19 }
```

Figure 3.12: The SCION type that represents a decoded SCION packet.

possible. This means the border router has methods that directly work on pointers or that use data structures that contain aliased slices.

The `scionPacketProcessor` type as depicted by Figure 3.7 is an example of a struct type that uses aliasing. The `scionPacketProcessor` type contains the `scionLayer` field of type `*SCION` (the definition of this type is shown in Figure 3.12) that stores a decoded representation of a SCION packet. Additionally, the `scionPacketProcessor` type also contains the `rawPkt` field that is the raw byte representation of the SCION packet. This means the `scionPacketProcessor` has both a byte-level representation and a decoded representation of the same SCION packet. In the following, we analyze how the fields of the `scionPacketProcessor` struct are aliased and what challenges aliasing imposes on program verification.

When a `scionPacketProcessor` struct is created, the router decodes the raw byte representation of the SCION packet that is stored in the `rawPkt` field to a decoded representation of type `*SCION` that is stored in the `scionLayer` field. After the SCION packet is decoded, the border router reaches a program state where the fields of the `scionLayer` are aliased with the `rawPkt` slice.

Figure 3.13 shows the aliasing between the `scionLayer` and the `rawPkt` fields. The decoded `scionLayer` field contains slices that point to sublices of the `rawPkt` field. This means if these slices in the `scionLayer` struct are updated, then the corresponding bytes of the `rawPkt` slice will be overwritten. Furthermore, the `scionLayer` contains also a pointer to a `Path` object. This `Path` object is also directly accessible in the `scionPacketProcessor` struct via the `path` field. Additionally, this path object contains another byte slice that is aliased with the `rawPkt` slice. As we have seen in Figure 3.8, the SCION packet processing pipeline invokes all methods on the `scionPacketProcessor` struct. Depending on the method they either update the decoded `scionLayer` representation of the packet and serialize the modified `scionLayer` struct to the `rawPkt` slice or they overwrite the `rawPkt`

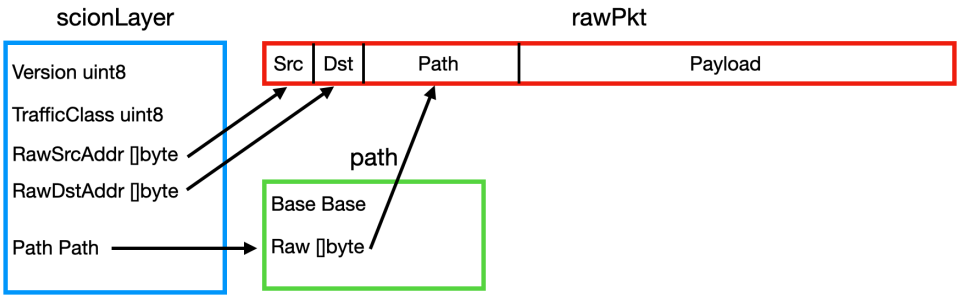

Figure 3.13: The aliasing situation between the `scionLayer`, `rawPkt`, and `path` fields of the `scionPacketProcessor` type.

slice directly. This means when we verified these methods, they either required write permission to the `scionLayer` struct or to the `rawPkt` slice. This setting makes managing access permissions more difficult since it is not possible to give write permission to both structs at the same time. So, we required a technique that allowed us to transfer write permission from the `scionLayer` struct to the `rawPkt` slice and vice-a-versa.

Solution

We solved the aliasing challenge by using magic wands and abstract ghost methods to manage access permissions between the `scionLayer` and the `rawPkt` fields. The idea of our solution is to use an abstract ghost method to transfer access permission from the `scionLayer` field to the `rawPkt` slice. This ghost method also returns a magic wand that can be used to transfer access permission back from the `rawPkt` slice to the `scionLayer` field. In the remainder of this subsection, we outline our solution in detail.

The first step toward our solution was to add a ghost field to the `SCION` type and to specify its decoding method. Figure 3.14 shows these changes. The specification of the `DecodeFromBytes` method states that it consumes access permission of its byte slice argument and that it returns the `Mem` predicate of its receiver argument. The second postcondition of the decoding method specifies that after decoding, the `RawPkt` ghost field stores the slice from which the `SCION` struct was decoded. Given that we have the predicates and the specified decoding method from Figure 3.14, we defined an abstract ghost method that exchanges access permission to the `SCION` struct with access permission to the byte slice from which the `SCION` struct was decoded.

Figure 3.15 shows the definition of the `getRawPktAcc` method. Given that the caller of this method has the `Mem` predicate of `s` and the byte slice that was used to decode `s`, the method can be invoked. The method then consumes the `Mem` predicate of `s` and returns the `BytesAcc` predicate for the byte slice `raw`. Additionally, this

```
1  type SCION struct {
2          // other fields omitted

3          /* ghost */
4          RawPkt []byte
5  }

6  pred BytesAcc(b []byte) {
7          forall i int :: { b[i] } 0 <= i && i < len(b) ==> acc(&b[i])
8  }

9  pred (s *SCION) Mem() {
10         acc(s) &&
11         BytesAcc(s.RawSrcAddr) &&
12         BytesAcc(s.RawDstAddr) &&
13         s.Path != nil && s.Path.Mem() &&
14         s.Version >= 0 &&
15         s.TrafficClass >= 0 &&
16         s.FlowID >= 0 &&
17         s.HdrLen >= 0 &&
18         s.PayloadLen >= 0 &&
19 }

20 requires BytesAcc(data)
21 ensures s.Mem()
22 ensures data == unfolding s.Mem() in s.RawPkt
23 func (s *SCION) DecodeFromBytes(data []byte)
```

Figure 3.14: The SCION type, its `Mem` predicate, and its specified decoding method.

```
1  ghost
2  requires s.Mem()
3  requires raw == unfolding s.Mem() in  s.RawPkt
4  ensures BytesAcc(raw)
5  ensures BytesAcc(raw) --* s.Mem()
6  decreases _
7  func (s *SCION) getRawPktAcc(raw []byte)
```

Figure 3.15: Ghost method that transfers access to the `raw` byte slice. The method requires that the `raw` argument is the byte slice from which the `SCION` struct was decoded. Additionally, the method returns a magic wand that can be used to transfer access back to the `SCION` struct.

method also returns the magic wand `BytesAcc(rawPkt) --* s.Mem()`. After we are finished with accessing `rawPkt`, we can use the magic wand to regain the `Mem()` predicate of `s`.

3.3.2 Verification Time

Verifiers like Gobra are prone to long verification times if they are used to verify complex methods. The code base of the SCION border router has some complicated methods and the whole code was not written with verification in mind. This made the verification process challenging. We often had to rewrite code fragments such that Gobra terminated in a reasonable time. In this subsection we show some code fragments that caused long verification times and explain how we have rewritten them to simplify verification.

```
1  requires DataPlaneMutexInvariant(d)
2  ensures DataPlaneMutexInvariant(d)
3  decreases
4  func foo1(d *DataPlane) {
5          unfold acc(DataPlaneMutexInvariant(d), 1/2)
6          if d.running {
7                  // do something
8          }
9          fold acc(DataPlaneMutexInvariant(d), 1/2)
10 }

11 requires DataPlaneMutexInvariant(d)
12 ensures DataPlaneMutexInvariant(d)
13 decreases
14 func foo2(d *DataPlane) {
15         if d.isRunning() {
16                 // do something
17         }
18 }

19 requires acc(DataPlaneMutexInvariant(d), _)
20 ensures res == unfolding acc(DataPlaneMutexInvariant(d), _) in d.running
21 decreases
22 pure func (d *DataPlane) isRunning() (res bool) {
23         return unfolding acc(DataPlaneMutexInvariant(d), _) in d.running
24 }
```

Figure 3.16: The `foo2` method shows how we can use a getter method to avoid unfolding operations.

Unfolding Predicates

When using verifiers we commonly use predicates to reason about access permissions of structs. For example, the fields of the `DataPlane` struct from Figure 3.5 are frequently accessed by the border router's methods. In order to reason about access permission to the `DataPlane` struct, we defined the `DataPlaneMutexInvariant` predicate.

The `foo1` method in Figure 3.16 shows how we can access the `running` field by unfolding the `DataPlaneMutexInvariant` predicate. (The `DataPlaneMutexInvariant` captures access the `running` boolean and the other fields of the `DataPlane` struct). Unfolding predicates can significantly increase the verification time. To keep the verification time in bound we commonly use getter and setter methods that access fields of structs. The `foo2` method shows how we can avoid the unfolding operation by using a getter method. Getters can significantly reduce the verification time. We did not perform any benchmarks on how getter methods improve verification time since getters have been used and evaluated in previous projects. For example, Foster et al. [7] extensively used getters, and they measured a significant decrease in verification time.

Quantifiers

Another challenge for the Gobra verifier are quantifiers. Each assertion that contains quantifiers leads to quantifier instantiations during verification. If a method

```
1  preserves p.Mem()
2  func (p *scionPacketProcessor) processEgress_unoptimized() error {
3          // omitted code section

4          unfold p.Mem()
5          unfold HopFieldInv(p.hopField)
6          p.infoField.UpdateSegID(p.hopField.Mac)
7          fold HopFieldInv(p.hopField)
8          fold p.Mem()

9          // omitted code section
10 }

11 preserves p.Mem()
12 func (p *scionPacketProcessor) processEgress() error {
13         // omitted code section

14         unfold p.Mem()
15         // use helper function to reduce verification time
16         // p.infoField.UpdateSegID(p.hopField.Mac)
17         updateSegID(p.hopField, p.infoField)
18         fold p.Mem()

19         // omitted code section
20 }

21 pred HopFieldInv(h *HopField) {
22         acc(h) &&
23         h.ConsIngress >= 0 &&
24         h.ConsEgress >= 0 &&
25         len(h.Mac) == MacLen &&
26         forall i int :: 0 <= i &&
27         i < len(h.Mac) ==> acc(&(h.Mac)[i])
28 }
```

Figure 3.17: Two versions of the `processEgress` method.

uses assertions with quantifiers, then the number of quantifier instantiations can be very high. We have observed that methods that trigger a high number of quantifier instantiations require significantly more time to verify than methods that trigger less quantifier instantiations. We observed this behavior for the `processEgress` and the `updateNonConsDirIngressSegID` methods of `dataplane.gobra`. To illustrate our observations we compare two versions of the `processEress` method.

Figure 3.17 shows two code snippets of the border router's `processEgress` method. The `processEgress_unoptimized` method is the original version of the `process Egress` method. The `processEgress_unoptimized` method calls the `UpdateSegID` method that requires access permission to the `MAC` slice of the `hopField` struct. As can be seen in the definition of the `HopFieldInv` predicate, the access permission to the `MAC` field is defined by a quantified assertion. The unfolding of the `HopFieldInv` predicate exposes the quantifier of the `MAC` field to the context of the `processEgress_unoptimized` method and triggers millions of quantifier instantiations during verification. We optimized the verification of this method by outlining the unfolding of the `HopFieldInv` predicate and the call of the `UpdateSegID` method to the `updateSegID` helper method.

In the following, we compare the verification times and the number of trigger in-

	Verification Time		**Quantifier Instantiations**	
Method	M	SD	M	SD
processEgress_unoptimized	572.3	127.1	1420549.8	980706
processEgress	47.2	0.8	248	0

Table 3.1: Verification time and number of quantifier instantiations for both versions of the `processEgress` method. For both methods, we measured five times the verification time and five times the number of quantifier instantiations. The table shows the mean (M) and the standard deviations (SD) for the verification time (in seconds) and the number of quantifier instantiations. We measured the number of quantifier instantations by using silicon on the generate Viper programs of the two methods. We fixed all seeds of the Z3 SMT solver that could be fixed.

CPU:	Intel Core i7-9750H@2.6 GHz (Turbo Boost disabled)
RAM:	32 GB
OS:	MacOS 11.6.1
Java version:	OpenJDK 11.0.11
Gobra commit	9fd0a6911bb1511b858e81e9f8ae212f1f9033ff
Silicon commit	cd20cf02b31ccd82a2cfdca7003947fd4380fe7b

Table 3.2: Specification of the test system.

stantiations for the `processEgress_unoptimized` and `processEgress` methods. We measured the verification time by verifying both methods five times and computed their mean times and standard deviations in seconds. Then we generated the Viper programs of both methods and used Silicon to measure the number of trigger instantiations for verifying both Viper programs. We ran Silicon with fixed random seeds to make Z3 as deterministic as possible (we cannot make Z3 completely deterministic) and measured the trigger instantiations of both Viper programs five times. All tests were executed on the system depicted in Table 3.2.

As can be seen in Table 3.1, the number of quantifier instantiations and the mean verification time is significantly lower for the `processEgress` method. The measurements in Table 3.1 show that the verification time for the `processEgress` method has a much smaller standard deviation than for the unoptimized one. For the number of quantifier instantiations, we see a similar pattern. For the `processEgress` method, the number of quantifier instantiations is constant 248. Thus, the standard deviation is 0. For the unoptimized method, the standard deviation is high. The high standard deviation of the unoptimized method can be explained by the nondeterministic behavior of the Z3 SMT solver. Even though we fixed all possible random seeds of Z3, the nondeterministic triggering behavior could not be prevented for the unoptimized method.

Verifying the entire Border Router

While we can verify single methods of the border router in isolation, it was not possible to verify all methods of the border router at once. For small Go packages,

Gobra can encode the whole package to one Viper file and verify it. However, for big packages like the `router` package that implements the SCION border router, the generated Viper program gets too big such that the verification does not terminate within a reasonable time. To verify all methods of the border router, we used a script that verifies all methods in isolation.

Chapter 4

Specifying and Verifying IO Behavior

In this chapter, we explain how we specify and verify the IO behavior of the SCION border router. The idea of our approach is to describe the desired IO behavior of the border router as a guarded event system. By verifying that the border router adheres to the specification of this event system, we can prove IO properties about the border router. In Section 4.1, we explain the concept of guarded event systems. Then, in Section 4.2, we define a guarded event system of the border router. In Section 4.3, we encode the guarded event system in Gobra. In Section 4.4, we show how we use our encoded guarded event system to specify and verify IO behavior. In Section 4.5, we conclude this chapter by discussing the verified IO properties of the border router.

4.1 Guarded Event Systems

A guarded event system is a transition system that consists of states, events, guards, and update functions. The states represent the configurations of the guarded event system. The events are actions that can trigger transitions of the guarded event system. For each event, the guarded transition system specifies a guard and an update function. A guard is a boolean expression that describes under which conditions an event triggers a transition. An update function specifies how the state of the guarded event system is updated if its corresponding event occurs and its guard is satisfied. To exemplify the concepts introduced in this section, we look at a guarded event system that describes the IO behavior of a simple echo server. Consider a guarded event system for an echo server that receives messages and sends them afterwards. The state is represented by the variable S. We represent the state as a multiset of facts. The facts in the multiset represent the configuration of our system. The idea of this multiset of facts is that it models that an event can introduce new facts or consume facts from the state of the event system. The initial state of our echo server is $S = \varnothing$. Our guarded event system defines the $receivedFact(msg)$. This fact represents that a message

$$\begin{aligned} receive(msg) &: true && \triangleright\ S := S \cup \{receivedFact(msg)\} \\ send(msg) &: receivedFact(msg) \in S && \triangleright\ S := S \setminus \{receivedFact(msg)\} \end{aligned}$$

Figure 4.1: The receive and send transitions for our example event system.

msg was received. The msg parameter is an abstract data type that represents an abstraction of the concrete message that is received. Abstract data types are immutable mathematical values used to abstract concrete types. Our system has a $receive(msg)$ and a $send(msg)$ event. The transitions of these events are defined in Figure 4.1. For the receive transition, the guard is always true. If a $receive(m)$ event occurs, then the multiset is updated by adding a $receivedFact(m)$. For a $send(m)$ transition, the guard requires that a $receivedFact(m)$ is contained in the multiset S. This constrains that the system is only allowed to perform a $send(m)$ transition if the $receive(m)$ transition has occurred before. If the $send(m)$ transition is executed, then the $receivedFact(m)$ is removed from S.

4.2 Guarded Event System of the Border Router

In this section, we define a guarded event system of the border router. The Idea is to define a guarded event system that represents the behavior of the border router. In Subsection 4.2.1, we define the events of the border router. In Subsection 4.2.2, we describe the state of the border router. In the end, in Subsection 4.2.3, we show the transitions of the event system.

4.2.1 Events

In this subsection, we identify the events of the SCION border router. We distinguish between IO events and internal events. IO events represent IO operations, such as receiving or sending network packets. In the following, when we mention IO events, we only consider network operations and no other IO operations such as reading or writing files or sending and receiving data between threads via channels. Internal events model local computations of the program. In the context of the SCION border router, internal events are those events that model the processing of a packet after it has been received and before it is forwarded.

Figure 4.2 shows the events of the border router. The structure of this figure is similar to that of Figure 3.3 in Section 3.1 . As we can see in Figure 4.2, the four packet types lead to different sequences of events.

All events of the border router represent a method in the router's code and the event is only performed if this method is executed before. The parameters of the events are abstract data types that abstract the concrete types of the corresponding router method. In the following, we describe the events in detail:

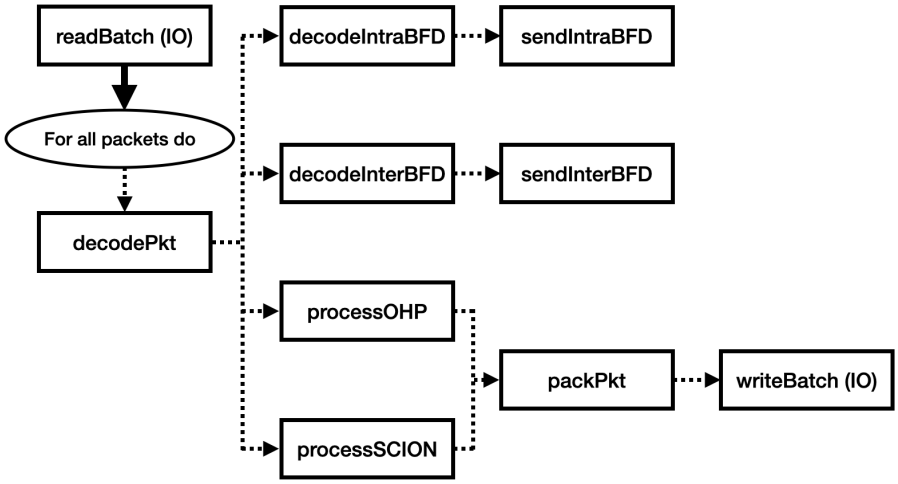

Figure 4.2: The events of the SCION border router.

IO Events

- *readBatch(msgs):* Reads the messages 'msgs' from a network interface. This event represents the execution of the `ReadBatch` method in the `router` package.
- *writeBatch(m):* Writes the message 'm' to a network interface. This event represents the execution of the `WriteBatch` method in the `router` package for a single message.

Internal Events

- *decodePkt(m, spkt):* Decodes the message 'm' to the SCION struct 'spkt'. This event represents the execution of the `DecodeFromBytes` method of the `SCION` struct.
- *decodeIntraBFD(spkt, bfd):* Decodes the SCION struct 'spkt' to the BFD struct 'bfd'. This event represents the execution of the `DecodeFromBytes` method of the `BFD` struct for an intra BFD packet.
- *decodeInterBFD(spkt, bfd):* Decodes the SCION struct 'spkt' to the BFD struct 'bfd'. This event represents the execution of the `DecodeFromBytes` method of the `BFD` struct for an inter BFD packet.
- *sendIntraBFD(bfd):* Sends the BFD struct 'bfd' to a BFD session. This event represents the execution of the `sendIntraBFD` method in the `router` package (This method was added for verification purposes and does not exist in the original SCION code).
- *sendInterBFD(bfd):* Sends the BFD struct 'bfd' to a BFD session. This event represents the execution of the `sendInterBFD` method in the `router` package

(This method was added for verification purposes and does not exist in the original SCION code).

- *processOHP(spkt, res):* Processes the OHP packet encoded in 'spkt' to a process result 'res'. This event represents the execution of the `processOHP` method in the `router` package.
- *processSCION(spkt, res):* Processes the SCION packet encoded in 'spkt' to a process result 'res'. This event represents the execution of the `processSCION` method in the `router` package.
- *packPkt(res, m):* Packs the process result 'res' in the message 'm'. This event represents the execution of the `createMessages` method in the `router` package (This method was added for verification purposes and does not exist in the original SCION code).

4.2.2 State

After we have identified the events of the border router, we define the abstract state of the router. Similar to the example of Section 4.1, we use a multiset of facts to represent the state of the border router. Based on the identified events of Section 4.2.1 we have this list of facts:

- **inFact(m):** Describes that a message was received. The 'm' parameter denotes the abstract representation of the received message.
- **decodedPktFact(spkt):** Describes that a message was decoded to a SCION struct. The 'spkt' parameter denotes the abstract representation of the SCION struct.
- **decodedIntraBFDFact(bfd):** Describes that a BFD struct was decoded from an intra BFD packet. The 'bfd' parameter denotes the abstract representation of the BFD struct.
- **decodedInterBFDFact(bfd):** Describes that an BFD struct was decoded from an inter BFD packet. The 'bfd' parameter denotes the abstract representation of the BFD struct.
- **processedPktFact(res):** Describes that a packet was processed. The 'res' parameter denotes the abstract representation of the processing result.
- **outFact(m):** Describes that a message is ready for sending. The 'm' parameter denotes the abstract representation of the message.

Besides these facts, the border router also contains static facts that are added when the border router is launched and that are never removed from the state. The border router has the following static facts:

- **localIAFact(ia):** Identifies the ISD and AS identifiers of the router. The 'ia' parameter denotes the abstract representation of the ISD and AS identifier pair.
- **supportedEgressIDsFact(egressIDs):** Identifies the supported egress IDs of the router. The supported egress IDs are the identifiers of all interfaces that are attached to a border router. The 'egressIDs' parameter denotes an abstract representation of the set of supported egress IDs.
- **internalConnFact(conn):** Identifies the network connection used to communicate with the internal network. The 'conn' parameter is an abstract representation of the network endpoint.

4.2.3 Transitions

Given the events from Subsection 4.2.1 and the state from Subsection 4.2.2, we define these transitions:

- **readBatch(msgs):**
  $true \rhd S := S \cup \{inFact(m_i) | m_i \in msgs\}$.
- **decodePkt(m, spkt):**
  $inFact(m) \in S \wedge G\_decodePkt(m, spkt) \rhd$
  $S := (S \setminus \{inFact(m)\}) \cup \{decodedPktFact(spkt)\}$.
- **decodeIntraBFD(spkt, bfd):**
  $decodedPktFact(spkt) \in S \wedge G\_decodeIntraBFD(spkt, bfd) \rhd$
  $S := (S \setminus \{decodedPktFact(spkt)\}) \cup \{decodedIntraBFDFact(bfd)\}$.
- **decodeInterBFD(spkt, bfd):**
  $decodedPktFact(spkt) \in S \wedge G\_decodeInterBFD(spkt, bfd) \rhd$
  $S := (S \setminus \{decodedPktFact(spkt)\}) \cup \{decodedInterBFDFact(bfd)\}$.
- **sendIntraBFD(bfd):**
  $decodedIntraBFDFact(bfd) \in S \rhd S := S \setminus \{decodedIntraBFDFact(bfd)\}$.
- **sendInterBFD(bfd):**
  $decodedInterBFDFact(bfd) \in S \rhd S := S \setminus \{decodedInterBFDFact(bfd)\}$.
- **processOHP(spkt, res):**
  $decodedPktFact(spkt) \in S \wedge G\_processOHP(spkt, res) \rhd$
  $S := (S \setminus \{decodedPktFact(spkt)\}) \cup \{processedPktFact(res)\}$.
- **processSCION(spkt, res):**
  $decodedPktFact(spkt) \in S \wedge G\_processSCION(spkt, res) \rhd$
  $S := (S \setminus \{decodedPktFact(spkt)\}) \cup \{processedPktFact(res)\}$.
- **packPkt(res, m):**
  $processedPktFact(res) \in S \wedge G\_packPkt(res, m) \rhd$
  $S := (S \setminus \{processedPktFact(res)\}) \cup \{outFact(m)\}$.

- **writeBatch(m):**
  $outFact(m) \in S \rhd S := S \setminus \{outFact(m)\}$.

Similar to the example from Section 4.1, the transitions have a guard and an update expression that is executed when the transition is taken. The guards in these transitions check if a fact is in the state and if the guard functions evaluate to true. For example, for the *processOHP(spkt, res)* event the transition can only be performed if a $decodedPktFact(spkt)$ is in the state and if the guard function $G\_processOHP(spkt, res)$ evaluates to true. The guard functions are used to express relations between the abstract types of the events. For the *processOHP(spkt, res)* event, the $G\_processOHP(spkt, res)$ evaluates to true if res represents a correct processing result for the abstract SCION struct $spkt$. As we will later see, these guard functions can be used to express IO properties between the input and output parameters of events.

4.3 Encoding the Event System of the Border Router

This section presents how we encode the event system of the border router in Gobra. The code snippets that we use in this section are simplified code fragments of the verified router code. This means that for presentation purposes, we removed some parameters of methods and simplified the contracts of the methods. In Subsection 4.3.1, we show how we encode the state of the border router. Subsection 4.3.2 presents the encoding of the events of the event system. In Subsection 4.3.3, we encode the transitions of the border router's event system.

4.3.1 State

After we have identified the facts of the border router, we encode the abstract state of the router. We first show how we define abstract types in Gobra, then we present the encoding of the border router's facts.

Abstract Data Types in Gobra

Gobra allows defining abstract data types by using the `domain` keyword. Figure 4.3 shows how the domain keyword is used to define an abstract type of the `Message` type. Domain types can have functions. However, these functions do not have bodies and contracts. The lack of preconditions makes domain functions applicable in any program state. To specify properties of domain functions, we use axioms. Axioms are global and specify properties that hold in any state of the program. Additionally, Figure 4.3 also presents an abstraction method that maps an instance of the `*Message` type to an abstract `AbsMessage` value. In the following, all abstractions of concrete types that appear in the code snippets can be identified by the `Abs` prefix in their name.

```
1  type AbsMessage domain {
2          func GetAbsMessagePayload(AbsMessage) seq[byte]
3  }

4  ghost
5  requires acc(m.Mem(), _)
6  decreases _
7  pure func ToAbsMessage(m *Message) (res AbsMessage)
```

Figure 4.3: Definition of the abstract `AbsMessage` type and an abstraction function that maps `*Message` values to the abstract `AbsMessage` type.

```
1  type Fact domain {
2          // incoming Message
3          func inFact(AbsMessage) Fact

4          // decoded Pkt
5          func decodedPktFact(AbsSCION) Fact

6          // decoded Intra BFD
7          func decodedIntraBFDFact(AbsBFD) Fact

8          // decoded Inter BFD
9          func decodedInterBFDFact(AbsBFD) Fact

10         // processed Pkt
11         func processedPktFact(AbsResult) Fact

12         // outgoing Message
13         func outFact(AbsMessage) Fact
14 }
```

Figure 4.4: The abstract `Fact` type with some of its functions.

Facts

Given that we have abstract types, we encode the facts identified in Subsection 4.2.2. The facts are encoded by using a domain type. Figure 4.4 shows the encoding of the `Fact` type. The `Fact` type has functions that are used to construct the facts of the border router.

4.3.2 Events

In this subsection, we will look at the encoding of the border router's events by showing the encoding of the *processOHP* event. The *processOHP* event is the internal event representing the processing of an OHP packet. The other events are encoded similarly.

As mentioned before, all events have a method in the router's code that represents the event and the event is only performed if the corresponding method in the router was executed. This means to perform the *processOHP* event the `processOHP` method in the `router` package needs to be executed before. In Figure 4.5 we show the relation between the `processOHP` method of the router and the *processOHP* event. The `processOHP` method processes a `SCION` struct `scn` and produces a `processResult` struct `pr`. The *processOHP* event represents that an abstract

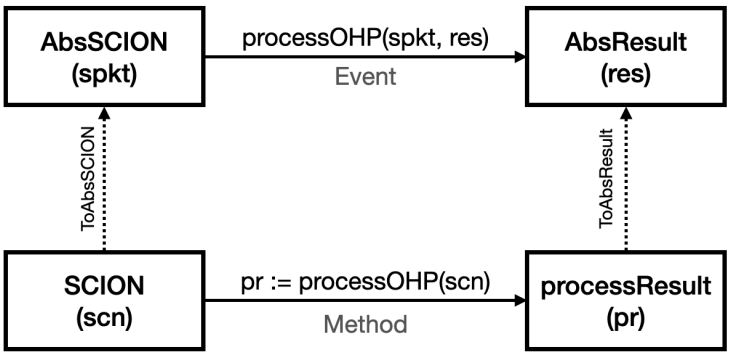

Figure 4.5: The relation between the `processOHP` method and the *processOHP* event.

`AbsSCION` variable `spkt` is processed to an abstract `AbsResult` `res` variable. The `spkt` and `res` variables are the abstractions of the `scn` and `pr` structs, respectively.

We encode the *processOHP* event as a ghost method. Executing this ghost method corresponds to performing the *processOHP* event. We need a mechanism to constrain that this event method can only be executed if the `processOHP` method was executed before. We achieve this by using the methodology from Penninckx et al. [12]. The idea is to constrain the execution of the *processOHP* event method with a predicate and a token. The predicate for the *processOHP* event method can only be obtained if the `processOHP` method was executed before. We later show how this is implemented. For the moment, we assume that the predicate is only available if the `processOHP` method was executed before.

Figure 4.6 shows the encoding of the *processOHP* event. The `processOHP_p` predicate represents the permission to execute the *processOHP* event. The predicate has parameters representing the place, the abstract SCION packet, and the abstract result. As we will see in the next subsection, an instance of the `processOHP_p(t, s, res)` predicate is only obtained if an OHP packet was processed by the routers `processOHP` method and if `res` is a valid abstraction of the actual processing result and if `s` is an abstraction of the processed `SCION` struct. The `get_processOHP_t1` function is used to return the next place when the event is executed. The `processOHP_f` function is used to move the token when the *processOHP* event is performed. As such, executing the `processOHP_f` method represents performing the internal *processOHP* event in the event system. Note the `processOHP_f` method can only be executed if the caller has a corresponding instance of the `processOHP_p` predicate.

4.3.3 Transitions

In Figure 4.7, we present the encoding of the transitions. All transitions are encoded with predicates. The `P` predicate accumulates all these transition predicates.

```
1  pred processOHP_p(ghost t Place, ghost s AbsSCION, ghost res AbsResult)

2  ghost
3  requires processOHP_p(t, s, res)
4  decreases _
5  pure func get_processOHP_t1(t Place, s AbsSCION, res AbsResult) (t1 Place)

6  ghost
7  requires token(t) && processOHP_p(t, s, res)
8  ensures token(t1) && t1 == old(get_processOHP_t1(t, s, res))
9  decreases _
10 func processOHP_f(t Place, s AbsSCION, res AbsResult) (t1 Place)
```

Figure 4.6: The encoding of the *processOHP* event.

The transition predicates have a `pl` parameter of type `Place` and a `s` parameter of type `mset[Fact]`. The `Place` represents the current place of the token and the `s` parameter represents the current state of the event system.

If we need to perform a transition, we have to unfold the `P` predicate and the corresponding transition predicate. In Figure 4.7, we only show the definition of the transition predicate for the *processOHP* event. Unfolding the `P_processOHP` predicate yields a `processOHP_p` predicate instance for all `AbsSCION` and `AbsResult` pairs that satisfy the guard on line 15. Given that we have a pair of `AbsSCION` and `AbsResult` values that satisfy the guard, we assume that the `processOHP` method of the border router was executed and the `AbsSCION` and `AbsResult` values represent the abstractions of the input and output arguments of the `processOHP` method. Thus, we obtain the corresponding `processOHP_p` predicate instance to perform the *processOHP* event in our event system. As described in Subsection 4.3.2 performing an event moves the token to the next place. After performing the *processOHP* event, we get a new instance of the `P` predicate with a new token and an updated state. The `U` function is an update function that removes the `decodedPktFact(scn)` from the state and adds the `processedPktFact(pr)` to the state.

Figure 4.8 shows how we can perform the transition of the *processOHP* event, after the `processOHP` method is executed. This code snippet only verifies if the guard in `P_processOHP` evaluates to true. The precondition of the `ProcessOHPWrapper` implies that the first conjunct of the guard evaluates to true. The second conjunct of the guard is the guard function `G_processOHP`. The code snippet only verifies if this function is true. By specifying IO properties in the guard function `G_processOHP` we can constrain under which conditions the *processOHP* event induces a transition. In the next section, we present how we can specify IO behavior by constraining guard functions.

```
1  pred P(pl Place, ghost s mset[Fact]) {
2          P_readBatch(pl, s) &&
3          P_decodePkt(pl, s) &&
4          P_decodeIntraBFD(pl, s) &&
5          P_decodeInterBFD(pl, s) &&
6          P_sendIntraBFD(pl, s) &&
7          P_sendInterBFD(pl, s) &&
8          P_processOHP(pl, s) &&
9          P_processSCION(pl, s) &&
10         P_packPkt(pl, s) &&
11         P_writeBatch(pl, s)
12 }

13 pred P_processOHP(pl Place, ghost s mset[Fact]) {
14   forall scn AbsSCION, pr AbResult::
15       ((decodedPktFact(scn) in s) && G_processOHP(scn, pr))
16             ==>
17       (processOHP_p(pl, scn, pr) &&
18         P(get_processOHP_t1(pl, scn, pr),
19            U(s,decodedPktFact(scn), processedPktFact(pr))
20         )
21       )
22 }
```

Figure 4.7: The predicate P and the predicate of the *processOHP* event.

```
1  requires token(t) && P(t, s)
2  requires decodedPktFact(absSCION) in s
3  requires scion.Mem()
4  ensures token(t1) && P(t1, s1)
5  func ProcessOHPWrapper(scion SCION, ghost absSCION AbsSCION,
6  ghost t Place, ghost s mset[Fact])
7  (ghost t1 Place, ghost s1 mset[Fact]) {
8          // process OHP packet
9          processRes, absPR  := processOHP(scion, absSCION)

10         // relate abstract state to program state
11         unfold P(t, s)
12         unfold P_processOHP(t, s)
13         assert processOHP_p(t, absSCION, absPR)
14         ghost s1 = U(s, decodedPktFact(absSCION),
15                 processedPktFact(absPR))
16         ghost t1 = get_processOHP_t1(t, absSCION, absPR)
17         ghost processOHP_f(t, absSCION, absPR)
18         assert token(t1)
19         assert P(t1, s1)
20 }
```

Figure 4.8: A simplified code snippet from the verified border router code that verifies the IO behavior of the `processOHP` method.

```
1 ghost
2 decreases
3 pure func G_processOHP(s AbsSCION, r AbsResult) (res bool) {
4         return GetAbsSCIONEgressID(s) == GetAbsResultEgressID(r)
5 }
```

Figure 4.9: The `G_processOHP` guard function with a simple constraint that expresses a relation between the `s` and `r` arguments.

4.4 Specifying and Verifying IO Behavior

In this section, we show how we specify and verify the IO behavior of methods that correspond to the events of the router's event system. As an example, we present how we specify and verify IO behavior of the `processOHP` method. Our Approach consists of the following two steps:

(1) **Strengthening:** Strengthening means we add constraints that specify IO behavior to the definition of the guard function. In the beginning, we can start with a guard function that always returns true.

(2) **Verifying:** We verify the method with their strengthened guard function. This step requires relating the abstract types of the events with their concrete types by using lemma functions. Lemma functions are abstract ghost functions with pre- and postconditions. Given that the preconditions of a lemma function are satisfied, it deduces the properties in its postconditions. If we have relations that hold for the concrete types, we can use lemma functions to deduce the relations for the abstraction of the types. This allows us to deduce the properties encoded in the strengthened guard functions.

These two steps are repeated until we have specified the complete IO behavior of the method. In the following, we explain how we specify and verify IO behavior of the `processOHP` method. The code snippets that we use to illustrate our approach are simplified fragments from the IO verified code base.

Strengthening Assuming that some relations between the `SCION` and `processResult` types are preserved if we map them to their abstract representations, we can specify the IO behavior of the `processOHP` method by adding constraints to the `G_processOHP` method. In Figure 4.9, we present the `G_processOHP` method with a simple constraint. The method evaluates to true if the egress ID contained in the abstract SCION packet is equal to the egress ID of the abstract processing result. This condition needs to hold such that an outgoing OHP packet is correctly forwarded to the next hop (for simplicity, we ignore the case for incoming OHP packets).

Verifying So far, we have assumed that relations between non-abstract types are preserved for their abstract counter parts. In practice, this does not hold since by abstracting two types, we cannot directly deduce their relations for their

```
1 ghost
2 requires scn.Mem()
3 requires scn.GetEgressID() == pr.EgressID
4 ensures scn.Mem()
5 ensures  GetAbsSCIONEgressID(scn) == GetAbsResultEgressID(pr)
6 decreases _
7 func relateEgressID(scn *slayers.SCION, pr processResult
8         absSCION AbsSCION, absResult AbsResult)
```

Figure 4.10: A lemma function that establishes a relation between the `AbsSCION` and the `AbsResult` types.

abstract representations. To establish relations between the abstract types, we use lemma functions. The code snippet in Figure 4.10 shows a lemma function that establishes a relation between the abstract `AbsSCION` and the `AbsResult` types, given that the relation holds for the non-abstract types. By using this lemma function in the `processOHP` method, we can establish the assertion of the guard in Figure 4.9.

4.5 Verified IO Properties

In this section, we present the verified IO properties for the processing of intra BFD, inter BFD, OHP, and SCION packets.

4.5.1 Processing Intra BFD Properties

Recall that an intra BFD packet is processed by the `processIntraBFD` method. This method first decodes the intra BFD packet to a `BFD` struct and then it forwards the `BFD` struct to a BFD session thread. We verified the following property:

- The `processIntraBFD` method only forwards a `BFD` struct to a BFD session, if the `BFD` struct was decoded successfully by its `DecodeFromBytes` method (we consider the `DecodeFromBytes` method decoded successfully if it does not return an error).

4.5.2 Processing Inter BFD Properties

Similar to the `processIntraBFD` method, the `processInterBFD` method first decodes the inter BFD packet to a `BFD` struct and then it forwards the `BFD` struct to a BFD session thread. For the `processInterBFD` method we verified the following property:

- The `processInterBFD` method only forwards a `BFD` struct to a BFD session, if the `BFD` struct was decoded successfully by its `DecodeFromBytes` method (we consider the `DecodeFromBytes` method decoded successfully if it does not return an error).

4.5.3 Processing OHP Properties

Recall that an OHP packet consists only of one hop and is forwarded from a source AS to a destination AS. For the processing of OHP packets, we verified properties about the `processResult` struct that is returned by the `processOHP` method. For each OHP packet that is processed by the `processOHP` method, the border router takes one of the following roles if its condition holds:

- **Receiver**
  Condition: The border router's AS is the destination AS of the OHP packet.
  Task: The border router resolves the packet's destination address and forwards it to the next hop within its local AS.
- **Sender**
  Condition: The border router's AS is the source AS of the OHP packet and the router is connected to the destination AS of the OHP packet.
  Task: The border router forwards the packet to the destination AS.

We verified these implications:

- If the border router is the receiver, then the `egressID` field is set to 0.
- If the border router is the receiver, then the `OutConn` field is set to the internal network endpoint.
- If the border router is the sender, then the `egressID` field is set to the egress ID as encoded in the hop field of the OHP packet.
- If the border router is the sender, then the `OutConn` field contains the network endpoint of the destination AS.

If the border router does not take any of the roles listed above, then it returns an error. For error cases, the implications trivially hold.

4.5.4 Processing SCION Properties

Similar to the OHP properties, we verified properties about the `processResult` struct that is returned by the `processSCION` method. For the processing of a SCION packet, the border router takes one of the following roles if its condition holds:

- **Inbound router:**
  Condition: The border router's AS is the destination AS of the SCION packet, and the current hop field of the packet's path is the last hop in the path.
  Task: The border router resolves the packet's destination address and forwards it to the next hop within its local AS.
- **Proxy router:**
  Condition: The border router's AS is not the destination AS of the SCION packet and the border router is not connected to the AS of the next hop.

Task: The border router forwards the packet to another border router within its local AS, that is responsible for forwarding the packet to the next AS.

- **Transit router:**
  Condition: The border router's AS is not the destination AS of the SCION packet and the border router is connected to the AS of the next hop.
  Task: The border router forwards the packet to the next AS.

We verified these implications:

- If the border router is the inbound router, then the `egressID` field is set to 0.
- If the border router is the inbound router, then the `OutConn` field is set to the internal network endpoint.
- If the border router is a proxy router, then the `egressID` field is set to 0.
- If the border router is a proxy router, then the `OutConn` field is set to the internal network endpoint.
- If the border router is a transit router and the packet traverses in the construction direction of the SCION path, then the `egressID` is set to the egress ID as encoded in the current hop field of the packet's path.
- If the border router is a transit router and the packet traverses against the construction direction of the SCION path, then the `egressID` is set to the ingress ID as encoded in the current hop field of the packet's path.
- If the border router is a transit router, then the `Outconn` field is set to the network endpoint of the AS of the next hop.

If the border router does not take any of the roles listed above, then it returns an error. For error cases, the implications trivially hold.

Chapter 5

Evaluation

In this chapter, we evaluate the results of this thesis. In Section 5.1 we summarize which properties we verified during this project. In Section 5.2, we present some statistics about the verified code of the border router. In Section 5.3, we conclude this chapter by discussing the limitations we faced during the project.

5.1 Verified Properties

In this thesis, we verified almost all methods of the SCION border router. Some methods could not be verified since they depend on Go packages that are not supported yet by Gobra. For these methods, we specified trusted contracts. These trusted contracts were selected carefully since they represent assumptions under which verification holds. Additionally, we verified and specified methods of other SCION packages that are used by the border router. Besides verifying memory safety, we also proved termination of most of the border router's methods. We could not prove termination for all methods since there is one method (the `Any` method in `svc.gobra`) in the border router's call graph that acquires a lock. This method may not terminate since it might be blocked by another thread that does not release the lock. Consequently, all callers of the method may not terminate as well.

In this project, we found two issues in the SCION code base. One of these issues is a missing check in a decoding method. This missing check could lead to a situation where an invalid hop index is decoded from a serialized SCION packet. The other bug we found was a race condition in the `processIntraBFD` and `processInterBFD` methods. This bug could lead to a situation where a slice could be accessed by two threads simultaneously. We reported both bugs to the developers of the SCION protocol. The SCION developers have confirmed both bugs and already fixed the race condition bug.

After we proved memory safety of the border router, we added IO specifications

File	OM	VM	PV	MS	NV	Helper
pkt/router/dataplane.gobra	49	39	4	2	4	46
pkt/router/metrics.gobra	1	1	0	0	0	5
pkt/router/svc.gobra	5	4	0	1	0	2
Total	55	44	4	3	4	53

Table 5.1: Overview of the methods of the border router after verifying memory safety. Abbreviations: OM: number of original methods, VM: number of verified methods, PV: number of partially verified methods, MS: number of specified method stubs, NV: number of not verified methods, and Helper: number of added helper methods that were required to make the verification of memory safety feasible.

to prove IO properties. With our IO specification approach, we managed to verify some crucial IO properties about the packet forwarding process of the border router (as discussed in Section 4.5).

5.2 Statistics

In the following, we discuss statistics about the verified code of the border router. We analyze the annotation overhead and the verification performance. We focus our analysis only on the `router` package, which implements the SCION border router.

5.2.1 Verified Methods

The verified methods of the SCION border router are contained in the `svc.gobra`, `metrics.gobra`, and `dataplane.gobra` files. In the original code base, these three files contain 55 methods (excluding closures). In Table 5.1, we show the number of verified methods of the border router after verifying memory safety. As shown in the table, 44 methods are verified, four are partially verified, three have specified contracts, and four are not verified or specified. These four methods that have not been verified or specified are defined in the `dataplane.gobra` file but are not used by the main processing loop of the border router. To verify the 44 methods of the border router, we specified and verified 53 helper methods. A lot of these helper methods are getters, as we have seen them in Subsection 3.3.2. Others of these helper methods contain code fragments of the original methods. These methods were introduced to outline code fragments to reduce the verification overhead. The last type of helper methods that we introduced are ghost methods. These ghost methods were added to simplify the modification of additionally introduced ghost state.

5.2.2 Annotation Overhead

When using deductive verifiers like Gobra, an important benchmark is the annotation overhead of the verified code. In general, annotation overhead is measured by

```
1  requires acc(DataPlaneMutexInvariant(d), _)
2  ensures res == unfolding acc(DataPlaneMutexInvariant(d), _) in d.running
3  decreases
4  pure func (d *DataPlane) isRunning() (res bool) {
5          return unfolding acc(DataPlaneMutexInvariant(d), _) in d.running
6  }
```

Figure 5.1: A getter function that was introduced to simplify verification.

counting the number of lines of specification (LoS). For Gobra, the LoS includes all Gobra constructs such as predicates and pre- and postconditions. Additional lines of code that were introduced to rewrite unsupported code fragments are not counted as lines of specification. The LoS values are often compared to the number of lines of code (LoC) of the verified code. Given that we have LoC and LoS measurements for a verified program, we can define the specification overhead ratio. The specification overhead ratio is the ratio of the LoS value to the LoC value.

In our measurements, the LoC values are determined by counting only the number of lines of code that contain Go statements, including round and curly brackets. This means comments and empty lines are not counted as LoC. We counted the LoC for verified and partially verified methods, but we did not count the LoC of the unverified methods and the method stubs. For the four partially verified methods we counted all lines as LoC since all four pratially verified methods are almost verified. We approximate the LoS values with a lower bound by using a script since counting by hand is error-prone and requires a lot of time. Our script counts the number of lines of code that contain a Gobra keyword. Our script cannot count the number of lines that are contained in a predicate definition. Thus, we count the LoS of all predicate definitions by hand and add them to the value that the script computes. Our script also fails to recognize regular Go code that does not contain any Gobra keyword, but that is part of added getter functions. For the getter function in Figure 5.1 the script recognizes that lines 1 to 5 are specification code since they contain Gobra keywords, but it cannot detect that the closing bracket at line 6 is also a line of specification. We consider that a lower bound of the LoS value that ignores these lines is a reasonable approximation to the actual LoS value. Consider that some programmers would define the method in Figure 5.1 in just one line. Then the number of LoS would be 4. Thus, the coding style of the programmer has always an influence on the LoS values.

Table 5.2 shows the number of lines of code (LoC) that we (partially) verified and the number of LoS that are required to verify memory safety, and the additional number of LoS that are required to verify IO behavior. The files with an ”io” prefix were added to verify the IO behavior. Thus, we consider all their lines of code as required specifications. This means that for the ”io” files, the LoS and the LoC values are the same. From the total number of LoC and LoS in Table 5.2, we deduce the specification overhead ratio for verifying memory safety

File	Original LoC	Memory Safety	IO Properties
pkt/router/dataplane.gobra	1053	1093	275
pkt/router/metrics.gobra	134	41	0
pkt/router/svc.gobra	45	63	0
pkt/router/io_events.gobra	0	0	98
pkt/router/io_facts.gobra	0	0	58
pkt/router/io_place.gobra	0	0	5
pkt/router/io_spec.gobra	0	0	188
pkt/router/io_types.gobra	0	0	111
Total	1232	1197	735

Table 5.2: This table shows the number of lines of code (LoC) that we verified in each file of the border router. Additionally, the table shows a lower bound for the number of lines of specifications (LoS) required to verify memory safety and the number of additional LoS to verify IO Properties.

and for additionally verifying IO properties. For verifying memory safety, we get a specification overhead ratio of 0.97, and for additionally verifying IO properties, we get a specification overhead ratio of 1.57. These specification overhead ratios are significantly smaller than those from a similar project. For example, Foster et al. [7] verified memory safety and IO behavior of a part of the Python implementation of the SCION border router by using the Nagini verifier [6]. They got a specification overhead ratio of 3.5. A reason that might explain the significant difference between our and their specification overhead ratio could be that Go code is often more verbose than Python code. This means that for the same program, the Go code has a higher LoC value. If we assume that the LoS are comparable for Gobra and Nagini, then the specification overhead ratio for the Go code is smaller.

5.2.3 Verification Performance

Another important benchmark of verifiers is verification performance. This subsection presents statistics about the verification times for the Gobra files of the border router and for 20 selected methods from the `dataplane.gobra` file.

As discussed in Subsection 3.3.2, we cannot verify the entire border router package at once. We verified each file of the border router in isolation by using the chopper feature of the Gobra verifier. The chopper feature allows us to specify one or multiple methods that we want to verify. This reduces the verification time considerably since smaller and less complex Viper programs are generated. For the `dataplane.gobra` file, we tried to verify multiple methods with the chopper at once, but this often leads to long verification times. Thus, we have decided to verify the methods of `dataplane.gobra` one by one by using the chopper on single methods. After verifying all methods of `dataplane.gobra`, we added their verification times up. For the other files of the border router, we verified all methods of the files at once by using the chopper. Table 5.3 shows the mean verification times

	Memory Safety		**IO Properties**	
File	M	SD	M	SD
pkt/router/dataplane.gobra	7464.8	247.6	8927.2	308.8
pkt/router/metrics.gobra	47.0	1.5	50.0	2.4
pkt/router/svc.gobra	36.4	0.9	41.4	1.1
pkt/router/io_events.gobra	0	0	33.4	1.6
pkt/router/io_fact.gobra	0	0	32.5	0.9
pkt/router/io_place.gobra	0	0	6.1	0.3
pkt/router/io_spec.gobra	0	0	49.0	1.1
pkt/router/io_types.gobra	0	0	37.7	0.9

Table 5.3: Verification time for the files of the border router. All files were verified five times after we added memory safety assertions and after we added memory safety assertions and IO specifications. The table shows the mean times (M) and the standard deviations (SD) in seconds.

and the standard deviations in seconds for the Gobra files of the border router. We performed the measurements five times for both the memory safe implementation with and without added IO specifications. All tests were executed on the system described by Table 3.2. As we can see in Table 5.3, verifying the `dataplane.gobra` file requires significant more time than verifying the other files. This is not surprising since `dataplane.gobra` contains many methods, and some of them require a considerable amount of time to verify. The measurements also show that adding IO specifications increase the mean verification times. The longer verification time of `dataplane.gobra` after adding IO specifications can be explained by the additional introduced specifications in the methods of `dataplane.gobra`. We assume that the slightly higher mean verification times for `svc.gobra` and `metrics.gobra` in the version with added IO specifications comes from longer parsing times since Gobra needs to parse the additional "io" files. In the end, we note that adding IO specification increases the mean verification time of the border router by 20% if we only consider the verification of `dataplane.gobra` and ignore the verification times of the other files since they are negligible in comparison to `dataplane.gobra`.

In addition to verifying the Gobra files of the border router, we have chosen 20 methods of `dataplane.gobra` and analyzed their verification times. Among these 20 methods are methods of all levels of complexity, from the straightforward getter function up to the sophisticated packet processing method. Table 5.4 shows the mean verification times and the standard deviations for the 20 selected methods from `dataplane.gobra`. We performed the measurements five times for both the memory safe implementation with and without added IO specifications. All tests were executed on the system described by Table 3.2. The results in Table 5.4 show that for most methods, the mean verification times of the methods increases when we add the IO specifications. This is what we expected since the verification time of `dataplane.gobra` increases when adding IO specifications. For some methods such as the `alreadySet`, `DP.SetIA`, and `DP.Run` methods, the increase of

the verification time after adding the IO specification is almost negligible. The table shows that there are methods like the `processMessage`, `processMessageHelper`, and `SPP.process` methods that have a significant higher mean verification time with added IO specifications. The increase in verification time for these three methods can be explained by the fact that these methods required adding additional pre- and postconditions and ghost code to encode our IO specification. The verification time for the `SPP.packSCMP` method shows an odd behavior. The method's implementation did not change after adding IO specification to the router. However, its verification time decreased. We have no explanation for this observation. The `processMesage` and the `SPP.packSCMP` methods have significantly higher mean verification times than all the other methods in the table. These two methods require more time to verify than any other method in the border router's package. Our measurements show that most methods have small standard deviations compared to their mean verification times. An exception is the `processMesage` method. The `processMesage` method shows high standard deviations in both measurement series. From experience, we know that complex methods with high mean verification times tend to have higher standard deviations in comparison to their mean verification time. This is because complex methods are translated to more complex proof obligation inputs to the Z3 SMT solver. The Z3 SMT solver shows a higher variance in verification time when the complexity of its inputs increases since the variances of nondeterministic decisions of Z3 are added up.

5.3 Limitations

As mentioned before, we are not able to verify all methods of the border router. For example, some of the setup methods that configure the border router before it is launched were only partially verified. These methods require Go packages like `fmt`, `hash`, or `crypto` that are not supported yet by Gobra.

Another limitation that we observed is that Gobra can have performance problems when it verifies methods that use quantified assertions. We measured that quantified assertions can cause long verification times.

The last limitation we faced is that Gobra has performance issues when verifying complex packages. For the `router` package that implements the SCION border router, we were not able to verify the whole package at once since the verification did not terminate in a reasonable time.

	Memory Safety		**IO Properties**	
Method	M	SD	M	SD
DP.GetExternalBatchConn	37.4	1.0	42.4	1.3
alreadySet	30.6	0.7	33.2	0.9
DP.SetIA	45.7	1.0	48.6	1.4
DP.Run	56.8	1.0	59.4	0.9
RunReadClosure	43.6	0.6	50.6	0.9
initReadClosureBuffer	61.2	7.1	60.6	6.4
resetReadClosureBuffers	34.8	0.8	38.0	1.0
processBatch	40.1	0.8	48.3	1.0
processMessage	1048.8	116.6	1677.3	426.2
processMessageHelper	56.0	0.7	318.6	1.3
createMessages	56.2	0.8	75.1	1.2
DP.processPkt	47.0	1.3	77.9	1.4
processIntraBFD	113.8	5.1	140.2	6.2
processInterBFD	72.0	1.8	100.1	13.8
SPP.process	100.5	1.4	227.5	4.8
SPP.parsePath	44.8	1.0	47.2	1.2
SPP.processEgress	47.2	0.8	51.0	0.9
SPP.packSCMP	1099.4	42.7	1040.5	35.2
DP.processOHP	49.9	1.1	54.0	1.2
DP.processOHP2	207.2	4.4	290.3	27.2

Table 5.4: Verification time for 20 methods of the border router. These methods were verified five times after we added memory safety assertions and after we added memory safety assertions and IO specifications. The table shows the mean times (M) and the standard deviations (SD) in seconds. Abbreviations: DP: DataPlane, SPP: scionPacketProcessor.

Chapter 6

Conclusion

In this project, we used the Gobra verifier to verify memory safety of the SCION border router. We verified almost all methods invoked by the main processing loop of the border router. A few methods could not be verified since they depend on Go packages that are not supported yet by Gobra. While verifying memory safety, we also analyzed the triggering behavior of methods that use Gobra assertions with quantifiers. We found a relation between long verification times and a high number of quantifier instantiations.

After we verified memory safety of the border router, we specified an IO specification that captures the IO behavior of the SCION border router. Using our IO specification, we verified important IO properties about the packet processing methods. We could verify that the packet processing methods correctly read forwarding information from the packet's path and that they store this information in the `processResult` struct that is used by the router to forward the packet.

We measured the specification overhead and verification time for the memory verified code with and without the IO specification. Our measurements show that the specification overhead ratio is 0.97 for verifying memory safety and 1.57 for verifying memory safety and IO behavior. Our benchmarks of the verification times show that adding IO specification increases the verification time of the border router by 20%.

6.1 Related Work

There have been two projects on verifying the SCION border router. The insights gained from these previous works helped us in our work.

In the first project, Forster et al. [7] verified memory safety, termination, and IO behavior for a part of the Python implementation of the SCION border router by using the Nagini verifier [6]. Their IO specification approach was different then our. Instead of capturing the IO behavior with a guarded event system as we did,

they used the IO specification methodology from Astrauskas et al. [4] and proved properties about the packet send operation.

In the second work, Halm et al. [8] verified memory safety for a part of the Go implementation of the SCION border router by using Gobra. They verified some methods of the SCION packet processing pipeline. Additionally, they specified numerous library methods used by the border router. The results of their work provided a good basis for our project.

6.2 Future Work

Currently, there are methods in the border router that are only partially verified or not verified. To verify all methods in the border router we require that Gobra supports the Go packages `crypto`, `fmt`, `hash`, and `math`. A first possible project could be to extend Gobra such that we can complete the memory verification of the border router. The border router also depends on many SCION packages. Another project for the future would be to verify other SCION packages. As discussed in Section 5.1, there is one method that may not terminate since it acquires a lock. Because of this method, all other methods in the border router that depend on it may also not terminate. It would be desirable if we could find a way to replace this method with another method that terminates. This would allow us to prove that an iteration in the packet processing loop of the SCION border router always terminates.

In this work, we specified an IO specification and verified IO properties of the packet processing methods of the SCION border router. However, the IO specification is not complete yet. In a future project, we can continue by verifying more IO properties about the processing of intra BFD, inter BFD, OHP, and SCION packets. For example, for the `processOHP` and the `processSCION` methods we could additionally add IO constraints that specify that only a valid address is set in the returned `processResult` struct. Another interesting project would be to specify other IO specifications for other packages in the SCION codebase. For example, we could define an IO specification for the `bfd` package.

We have used Gobra to verify a complex codebase that was not optimized for the Gobra verifier. Thus, we had to adapt the code and introduced many helper methods such as getters to make verification feasible in a reasonable time. In future work, we could try to research in the other direction. For example, we could try to find methodologies that allow us to write Go code that is optimized for the Gobra verifier. Such a methodology could significantly simplify the development of verified Go programs.

Bibliography

[1] The Gobra project. https://github.com/viperproject/gobra. Accessed: 2021-2-11.

[2] The VerifiedSCION project. https://www.pm.inf.ethz.ch/research/verifiedscion.html. Accessed: 2021-2-10.

[3] The Viper project. https://www.pm.inf.ethz.ch/research/viper.html. Accessed: 2021-2-11.

[4] Vytautas Astrauskas. *Input-output verification in viper*. PhD thesis, Master's thesis, Department of Computer Science, ETH Zürich, 2016.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[6] Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *International Conference on Computer Aided Verification*, pages 596–603. Springer, 2018.

[7] Sascha Forster. Static verification of the scion router implementation. 2018.

[8] Luca Halm. Verification of practical go programs. 2021.

[9] Dave Katz, David Ward, et al. Bidirectional forwarding detection (bfd). 2010.

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.

[12] Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In *European Symposium on Programming Languages and Systems*, pages 158–182. Springer, 2015.

[13] Adrian Perrig, Pawel Szalachowski, Raphael M Reischuk, and Laurent Chuat. *SCION: a secure Internet architecture*. Springer, 2017.

[14] Yakov Rekhter, Tony Li, Susan Hares, et al. A border gateway protocol 4 (bgp-4), 1994.

[15] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs (extended version). *arXiv preprint arXiv:2105.13840*, 2021.

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):

Specifying and Verifying the IO Behavior of the SCION Border Router

Authored by (in block letters):

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

Name(s):	**First name(s):**
Telschow	Lino

With my signature I confirm that

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

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

Place, date	**Signature(s)**
Zürich, 23.3.2022	LTelschow

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