Specifying and Verifying the IO Behavior of the SCION Border Router

Master's Thesis Project Description

Lino Telschow

Supervisors: João C. Pereira, Felix Wolf, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich

Start: 22rd September 2021
End: 22rd March 2022

1 Introduction

SCION [1] is a new internet architecture that aims to overcome some of the most important problems of the current internet architecture. A lot of work has been put into the design of the SCION protocol and on proving that its intended properties hold. Nevertheless, reasoning on the protocol level alone does not guarantee that all properties hold in an implementation. This has lead to the VerifiedSCION project [2], whose goal is to verify SCION from the high-level protocol design all the way down to the implementation. So far, we have focused on verifying the memory safety of the SCION border router, written in Go [3], using the Gobra verifier [4]. The goal of this thesis is to extend our code level verification efforts and to specify and verify the IO behavior of the SCION border router using IO separation logic as introduced by Penninckx et al. [5]. After specifying and verifying the IO behavior of the router, we can extract a model and prove properties from the protocol level.

2 Background

SCION

SCION is an emerging internet architecture that provides route control, DDoS protection, and trusted end-to-end communication. SCION establishes these properties by grouping the internet's ASes in isolation domains and by employing a new routing mechanism. As opposed to routing tables and longest prefix matching implemented in the BGP [6] dominated internet, SCION forwards packets based on paths encoded in the packet header. The path in the header can be selected by the sender and specifies the ASes through which the packet is forwarded. SCION guarantees that a sender can only construct paths that adhere to the routing policies of all ASes. Furthermore, the paths are protected by cryptographic primitives, such that no adversary can change them. The SCION protocol distinguishes cleanly between a control plane and a data plane. The control plane is responsible for discovering paths and the data plane implements the packet forwarding process. The routers at the edges of an AS are called border routers. These border routers are the entry points of an AS and connect it with the neighboring ASes. Border routers are part of the data plane and are responsible for forwarding packets. Figure 1 shows the simplified structure of an AS with two border routers. `BR 1` is connected to other ASes via interfaces 1 and 2. Similarly, `BR 2` has two external interfaces. All border routers have an internal interface which connects them to all other components within the AS. This is represented by the dashed line connecting `BR 1` and `BR 2`. In order that SCION can deliver all of its benefits, it is of crucial importance that its border router implementation is correct. This means that there should not be any bugs like memory safety issues or unspecified behavior. Additionally, it must be guaranteed that the border router behaves according to the SCION protocol. In particular, the border router should only be allowed to perform the IO operations prescribed by the protocol and nothing else. To achieve this, we will

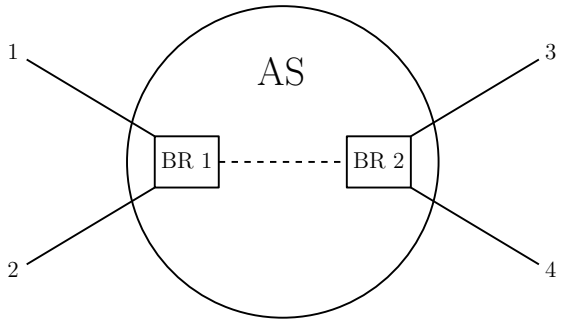

Figure 1: Simplified structure of an AS

specify and verify the IO behavior of the SCION border router by using the IO separation logic from Penninckx et al. [5].

Go

The current SCION implementation is open-source [7] and written in the Go programming language. Go is a compiled and statically typed programming language developed by Google. Go has a structural type system and built-in concurrency primitives like goroutines and channels. Goroutines allow the execution of functions as separate threads. Channels are used to send data between the concurrently running goroutines. These features make Go a very popular programming language for network applications, since these applications exhibit a high degree of parallelism and run on systems with many CPU cores.

Structure of the SCION border router

The SCION border router implementation in Go is a multithreaded application. When a border router instance is launched, it executes the `Run` method. As depicted in figure 2, the `Run` method then starts a set of concurrent running threads. A border router starts a BFD session for each of its interfaces. These BFD sessions run the BFD protocol [8] with the other endpoints of the interfaces to observe the state of the connections and to detect link failures. Additionally, a border router starts an instance of the main loop for each interface. The main loop is responsible for receiving, processing, and sending SCION packets.

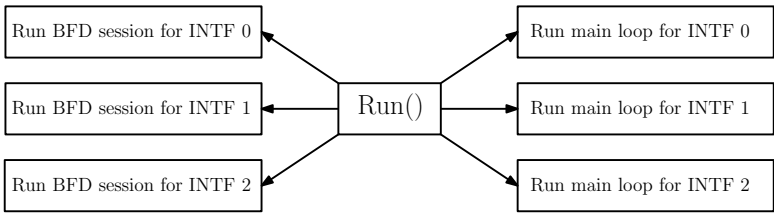

Figure 2: Main Tasks of a border router with 3 interfaces

Figure 3 shows the main loop of the border router that is executed for each of its interfaces. The main loop reads a batch of packets from its interface and then processes it. The `Process Batch` task decides for each packet in the batch whether it is forwarded or dropped. This means the writing of packets to an outgoing network buffer is captured by the `Process Batch` task.

Gobra

Gobra [9] is a modular, deductive verifier for Go programs. Gobra supports many Go specific features like goroutines and channels. Before Gobra can verify a Go program, it is required that the program is annotated with assertions, such as pre- and postconditions. These annotations specify the intended behavior of the program to verify. Gobra then verifies if all callers of each method satisfy the preconditions and if all methods can deduce their postconditions. If Gobra finds a contract violation, then it shows a descriptive error message to the developer. As other deductive

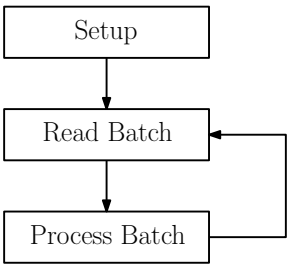

Figure 3: Main loop of the border router

verifiers, Gobra uses mechanisms based on separation logic to reason about shared data structures on the heap by constraining the state of the heap with pre- and postconditions.

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

Figure 4: Simple Gobra example

Figure 4 shows an annotated Go method, which increments an integer by one. The `requires acc(x)` assertion is the precondition of the method and ensures that the method gets write permission to the pointer `x`. With this permission, the method can update the pointer. The postconditions are denoted by the `ensures` keyword. The first postcondition returns the access permission to the caller and the second one describes that, if `inc()` terminates, then it increments the value of `*x` by one. The `old` keyword refers to values evaluated before the method has been executed.

Gobra can be used interactively in the development process as an extension for Visual Studio Code [10]. Gobra is a Go frontend for the Viper verification infrastructure [11]. This means that Gobra encodes the annotated Go code into a program in the Viper verification language. The Viper program is then verified by the Viper toolchain and the errors reported by Viper are back-translated to the Gobra level.

Event Systems

An event system is a system consisting of states and transitions. The states specify the valid configurations of the system. The transitions describe how the state of the system is updated when an event occurs. Each event system starts in an initial state, which describes the start configuration of the system. In the following, we show a small example.

Consider an event system for a bounded counter that can count up to five. The state is represented by a variable x, which stores a natural number. The initial state of our system is $x = 0$. Our system has an $inc()$ event, which increases the counter. For this $inc()$ event, we can define the transition $inc() : x < 5 \triangleright x := x + 1$. Each transition consists of a guard and an update function. In our example, the guard $x < 5$ requires that x has to be below five. The update function $x := x + 1$ describes that the state is incremented by one. A transition is executed if its event occurs and if its guard evaluates to true.

IO Specifications

To specify and verify the IO behavior of the SCION border router, we will define an IO Specification. An IO specification constrains the IO behavior of a program. This specification prevents the program from executing undefined IO actions.

Penninckx et al. [5] show an approach to how IO specifications can be encoded in pre- and postconditions of methods. Their idea is to represent IO actions as movements of tokens in a Petri net. Figure 5 shows an example for a Petri net that describes the IO behavior of an echo server. The circles are called places and the dot in circle t_1 is the token. This figure describes that the program first receives some string M and then sends it.

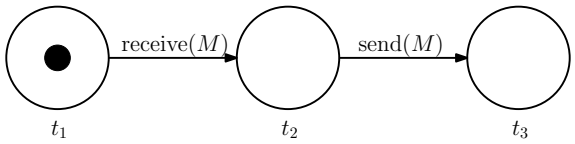

Figure 5: Petri net describing the IO actions of an echo server

Now, we want to specify the IO behavior depicted in figure 5 with pre- and postconditions. The procedure consists of the following three steps:

(1) First, we identify the basic input and output actions (BIO actions) of the program. BIO actions are the building blocks of all other IO actions. If the codebase uses an unverified library, then the BIO actions are the functions of this library. If all libraries are verified but the operating system is not, then we can identify the system calls as BIO actions.

For our example, we assume to have an unverified network library with the BIO actions: `func receive() string` and `func send(str string)`.

(2) Next, we define for each BIO action a corresponding predicate. In general, these predicates have at least three arguments. The first argument is the source place and the last one is the target place. The place arguments describe how the IO actions move the token from a source place to a target place. The arguments between the places are the input and output arguments of the corresponding BIO action.

Form our BIO actions, we can deduce the following predicates:
`pred receive(`T_1`,`str`,`T_2`)` and `pred send(`T_1`,`str`,`T_2`)`.

Additionally, we introduce a predicate that models the place of the token. The predicate `token(`T`)` describes that the token is currently at position T.

(3) Finally, we use the token and the BIO predicates to specify the IO behavior in the pre- and postconditions of methods.

For the receive method, the pre- and postconditions are defined as shown in figure 6. The receive method requires a token in some place T_1 and it also needs the receive predicate to move from place T_1 to some place T_2. The receive predicate has an existentially quantified variable `M`, which acts as a placeholder for any possible string returned by `receive`. The postcondition guarantees that after the execution of the method the token is in place T_2. For the send method, we can define the pre- and postconditions in a similar way.

Now, we want to use both receive and send in one method to implement the echo server. The code snippet in figure 7 shows how we can use predicates to specify the IO behavior of the echo server. Since the `receiveAndSend` method performs first a receive operation, we need a token in some place T_1 and the receive predicate. This allows us to receive some message `msg` and to move the token forward to some place T_2. From this place, we are then allowed to execute the send operation with the input `msg`. When the send function terminates, then the token will be in some place T_3. Thus, the postcondition will be deduced in the end.

```
1                requires token(T1) && receive(T1,M,T2)
2                ensures token(T2)
3                func receive() (M string)
4                {
5                  /* some implementation */
6                }
```

Figure 6: IO specification of the receive method

```
requires token(T1) && receive(T1,msg,T2) && send(T2,msg,T3)
ensures token(T3)
func receiveAndSend()
{
  msg := receive()
  send(msg)
}
```

Figure 7: IO specification of the echo server example

3 Core Goals

The overall goal of this thesis is to specify and verify the IO behavior of the SCION border router. We plan to specify the IO behavior in the style of the IO separation logic introduced by Penninckx et al. [5]. The tasks of the project can be grouped in four core goals:

(1) The first goal is to verify memory safety of the SCION border router. For this, we build on top of previous work, which verified memory safety of some parts of the SCION border router [4]. For this project, we plan to verify the memory safety of the border router's main loop, which implements the receiving, processing, and sending of packets and which can be found in the `DataPlane.Run()` method in the file `scion/go/pkg/router/dataplane.go` of the SCION codebase [7]. Unlike previous work, we will also verify concurrent code, which makes verifying memory safety more challenging. Furthermore, if Gobra does not support some code fragments, we will rewrite them such that they can be verified.

(2) As mentioned in the Background section, we will specify an IO specification. The second goal is to fix the model state used by the IO specification. The model state abstracts over implementation details and contains only information that is relevant to represent the IO behavior.

(3) The third goal is to specify and verify the IO specification using Gobra. The IO specification will be defined and verified incrementally by using a bottom-up approach. This means we begin to add IO constraints to the methods that are the leaves of our call graph. After the leave methods have been verified, we proceed with the specification and verification of the next level in the call graph. We are finished if we have specified and verified all levels of the call graph.

To verify the IO specification, we need memory safety and some functional properties. At this point, we should already be done with verifying memory safety (first goal). While defining and verifying the IO specification, we will have to enhance the functional specification of the border router, such that we can verify the desired IO properties. Furthermore, we will have to deal with concurrent threads performing IO operations. Concurrency makes verifying IO specifications more difficult to tackle. Instead of using the technique from Penninckx et al. [5], we may use an alternative approach.

(4) The last goal is to evaluate our IO verification technique. This thesis is a practical application of IO verification with Gobra. We will write an experience report, which includes statistics such as the number of lines of added Gobra annotations and the verification time of packages. The evaluation also includes a textual analysis of the biggest difficulties that we faced and how they were solved.

4 Extension Goals

(1) The first extension goal is to add features to Gobra which improve the verification process. In the following, we list some interesting features:

- More fine-grained control over the unrolling process of function definitions and predicates. This reduces the number of facts the verification backend has to deal with.
- Support for multiple specifications per method. This allows the developer to define specifications of different complexity.

- A search engine for lemmas.
- Support for inlined code specifications to avoid moving code into separate methods. For verification, moving code into separate methods has the advantage of simplifying the individual proof obligations, but it significantly changes the code. So far, we have a prototype of this feature implemented by L. Halm [4]. This prototype allows us to treat code blocks as outlined methods without changing the structure of the code. However, this prototype does neither support Go's return statements nor Gobra's old keyword.

(2) The second extension goal is to add features to the Gobra IDE. For example, we can add more control over the verification target. In particular, we consider the following features:

- Verification of a single method in the target program.
- Verification of the target's dependencies.
- Check that a particular precondition, postcondition, invariant, or predicate body does not contain a contradiction.

(3) The third extension goal is to verify additional functional properties of the border router. For example, we can verify SCION's serialization and deserialization components and proof properties about the relation between inputs and outputs.

(4) The fourth extension goal is to deduce an event system from the IO specification. This gives us an event system that abstracts the IO behavior of the SCION border router. The Igloo methodology [12] presents one way to relate an IO specification to an event system.

(5) The fifth extension goal requires the event system from the previous extension goal. Given that we have an event system, we can prove properties about it in a model checker. The challenge of this goal is to find interesting properties that are checkable by the model checker.

5 Schedule

Task	**Planed Weeks**
Goal 1: verify memory safety	3
Goal 2: define the model state of the border router	2
Goal 3: specify and verify the IO specification	6
Goal 4: evaluate	3
Extension goals	2
Writing Report	4

References

[1] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat, *SCION: a secure Internet architecture*. Springer, 2017.

[2] "The VerifiedSCION project," https://www.pm.inf.ethz.ch/research/verifiedscion.html, accessed: 2021-10-07.

[3] "The Go programming language," https://golang.org, accessed: 2021-10-01.

[4] L. Halm, "Verification of practical go programs," 2021.

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

[6] Y. Rekhter, T. Li, S. Hares *et al.*, "A border gateway protocol 4 (bgp-4)," 1994.

[7] "GitHub repository of the SCION Internet Architecture," https://github.com/scionproto/scion, accessed: 2021-10-01.

[8] D. Katz, D. Ward *et al.*, "Bidirectional forwarding detection (bfd)," 2010.

[9] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs (extended version)," *arXiv preprint arXiv:2105.13840*, 2021.

[10] S. Walker, "Ide support for a golang verifier," Ph.D. dissertation, Bachelor's thesis, ETH Zurich, 2020.

[11] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *International conference on verification, model checking, and abstract interpretation*. Springer, 2016, pp. 41–62.

[12] C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. Basin, "Igloo: Soundly linking compositional refinement and separation logic for distributed system verification," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–31, 2020.