Verifying the IO Behaviour of the SCION Router

Practical Work Project Report

Markus Limbeck
Supervised by João Pereira, Dionysios Spiliopoulos
under Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

March 23, 2024

1 Introduction

SCION [1] is a path-based internet architecture. In contrast to conventional destination-based routing, path-based architectures provide control over network paths by enabling endpoints to access relevant information.

The SCION protocol is designed to be resilient and secure. In particular, the protocol has been proven to satisfy properties like valley-freedom, loop-freedom, and path authorisation [2].

To guarantee that the aforementioned properties hold, there has been a joint effort, under the VerifiedSCION project, to prove the properties at the protocol level, and to prove that the open-source implementation of SCION behaves according to the protocol.

VerifiedSCION achieves this by stepwise refinement [3] using the Igloo approach [4]. In Igloo's first step, a very abstract model of the system is formally developed. Next, the model is incrementally refined by incorporating further details of system requirements and environment assumptions and decomposed into smaller components. Finally, IO specifications which describe the functional behaviour of each component are obtained. These specifications can then be used in combination with a deductive verifier, e.g. Gobra [5], to prove functional correctness of the implementation. The soundness of Igloo guarantees that properties proven on higher abstraction levels hold in the actual implementation as well.

At the time of writing, VerifiedSCION has almost fully proven memory safety of the implementation of the SCION border router, the component responsible for forwarding the packets between Autonomous Systems (AS). Furthermore, IO specifications were already obtained using the Igloo approach. The next step involves proving that the implementation adheres to the IO specification, ensuring the correct functional behaviour. Therefore the main objective of this project is verifying the implementation against the IO specification using Gobra.

2 Background

Verification methodologies typically expect programs to be specified via method contracts containing pre- and postconditions. They check that every program execution, fulfilling its precondition,

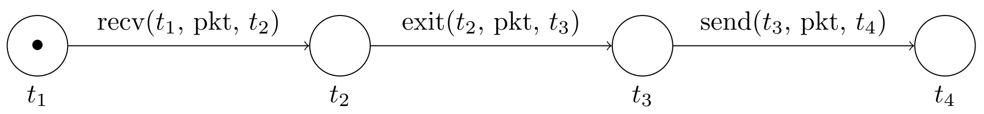

Figure 1: Simple illustrative Petri net.

```
requires token(t_1)
requires recv(t_1, pkt, t_2)
requires exit(t_2, pkt, t_3)
requires send(t_3, pkt, t_4)
ensures token(t_4)
method client(pkt : Int)
```

Listing 1: Simple Petri net from figure 1 written as Hoare triple.

guarantees its postcondition after execution. This permits reasoning about the program's initial and final state, which is sufficient for properties like memory safety. In order to specify the IO behaviour of a program using method contracts, one must first find a way to encode these specifications in terms of contracts.

2.1 Specifying the IO behaviour via contracts

One approach is using separation logic over Petri nets [6]. The high-level idea is assigning permissions to places via so-called tokens. An event can be executed only if the corresponding token is available. When an event is executed, the token is consumed and replaced with a new one. This concept is visualised using Petri nets and demonstrated in Figure 1. In the graph the nodes are called places, the edges denote IO events and the black dot represents the current token. Whenever an event is executed, the black dot moves along the edge to the next place. If a token transitions from an initial place to a final one, it guarantees the execution of all IO events along that path.

Given such a Petri net, it can easily be translated into a method contract as demonstrated in Listing 1. The precondition consists of all IO events with an initial token, while the postcondition only consists of the final token, ensuring that all IO events have happened.

2.2 Igloo

While the previous section clarifies how to specify which IO operations are allowed, it does not assist in obtaining the IO specification in the first place. To address this, a formal framework called Igloo is applied to soundly relate event-based systems to program specifications. The Igloo methodology follows a six-step process outlined in Figure 2. It begins with the definition of an abstract system model for which desired properties can be proven. Next, the model is refined to a distributed view, including an environment. Further refinement considers the IO library interfaces to be used in its implementation. The model is then decomposed from its monolithic form into system components. Each of these components is translated into a trace equivalent IO specification. Finally, these specifications are used to verify the implementation of a component to prove that they adhere to their intended IO behaviour. The overall soundness of this approach guarantees that successful verification of implemented components against their IO specifications implies that all properties proven at higher abstract levels hold for the implementation.

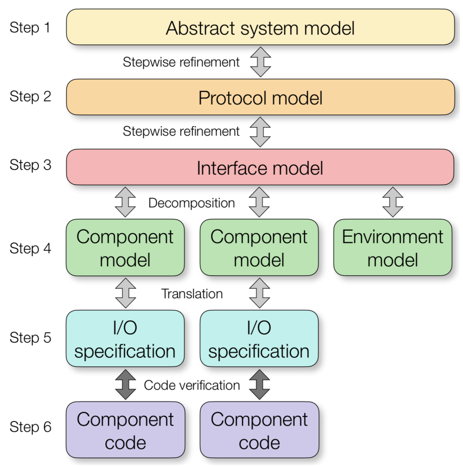

Figure 2: Overview of the six steps in the Igloo methodology. This figure is taken from the Igloo project website [4].

2.3 VerifiedSCION

Igloo allows to formally prove properties like valley freedom or path authorisation on the abstract model rather than the actual implementation via Isabelle/HOL [7]. Valley-freedom protects the economic interests of ASes by preventing routes that include ASes that do not financially benefit from the transmitted traffic. Path authorisation, on the other hand, ensures that packets are exclusively forwarded along authorised segments.

The generated IO specification can be used to prove that some implementation behaves according to the protocol. For the verification the IO specification is manually rewritten from Isabelle/HOL to Go, enabling their use in the deductive verifier Gobra. It consists of a local state and five IO events. The local state has an input buffer for incoming packets and an output buffer for outgoing packets. An IO event may require specific elements to be present in one of the buffers and can also guarantee that certain elements will be in one of the buffers after its execution. The following section offers a more detailed explanation of the IO events.

- **Recv** This event is responsible for processing all incoming packets. It lacks a guard allowing it to accept all types of packets. However, a packet is only added into the local state's input buffer if its type is subject to verification, e.g. a SCION packet will be added, whereas an EPIC packet will not.

- **Enter** All inbound packets, i.e. packets received from another autonomous system (AS), are managed by this event. Unless a segment change is necessary during processing, the enter event is always executed. Thus, the event's guard verifies whether packets destined for internal or external destinations are processed correctly. If this condition holds, the packet is placed into the local state's output buffer.

- **Xover** This event is similar to the enter event. It is needed when a segment change is required during processing. Otherwise, the enter event is used. A segment change involves additional processing steps which have to be checked by the event's guard, resulting in the differentiation between xover and enter. Furthermore, the event's handling depends on whether a core segment is involved or not, which is why the event is internally divided into two. Like the enter event, the xover event adds the packet to the local state's output buffer if all conditions are satisfied.
- **Exit** This event handles packets which have been received from the local AS and are destined for another AS. It requires a packet to be present in the local state's input buffer and processed correctly by the codebase. If both conditions are met, the packet is placed into the local state's output buffer.
- **Send** All outgoing packets are handled by this event. Once a packet is put into the local state's output buffer it is guaranteed that it has been processed correctly and can be forwarded to its destination. Once these criteria are met, the send event can be executed.

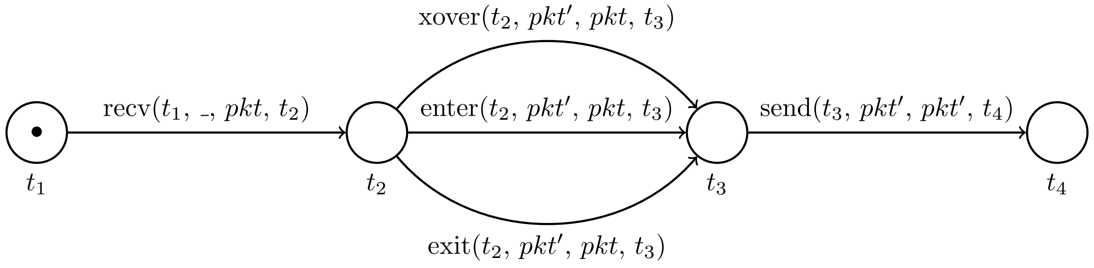
xover(t_2, pkt', pkt, t_3)
recv(t_1, _, pkt, t_2)
enter(t_2, pkt', pkt, t_3)
send(t_3, pkt', pkt', t_4)
t_1
t_2
t_3
t_4
exit(t_2, pkt', pkt, t_3)

Figure 3: Petri net for the IO specification.

3 Challenges

The overall objective of this work was to prove that the implementation satisfies the functional behaviour provided by the IO specification. This was further subdivided into three smaller goals: translating bytes into an abstract representation, annotating the codebase and ultimately verifying against the IO specification. Although substantial progress has been made on the first two objectives, verification remains an active effort at the time of writing. The subsequent section provides a description of the challenges encountered and their respective solutions.

3.1 Abstract state

One challenge lay in the disparities between the IO specification and the implementation. In some cases, events were handled differently even though they were logically equivalent. For instance, the IO specification differentiated between segment changes based on the involvement of a core router, whereas the implementation treated them uniformly.

This challenge was addressed by introducing a ghost state for the dataplane of the router. This ghost state stores information about the network that was not available in the actual dataplane, such as whether the router is part of the core routers. A further challenge was ensuring coherence between the actual dataplane and the abstract dataplane. However, this immediately led to performance issues because too much information had to be exposed to the verifier. While reducing the amount of exposed information, it uncovered incompleteness issues within the verifier. Overall, resolving this challenge took a significant amount of time.

3.2 Egress Interface

Inbound packets destined for the local AS posed a challenge as well. According to the definition, a packet is classified as inbound only if its ingress interface is not equal to 0. While this definition aligns with both the IO specification and the implementation, the classification of packets destined for the local AS differs. In the IO specification, these packets require the egress interface to be equal to 0, whereas the implementation relies on matching the source address with its own AS source address.

Although initially perceived as an issue, it was discovered during the verification of this case that using the egress interface or the source address makes no difference in the verification process. Consequently, this challenge was easily resolved.

3.3 Internal packets

The IO specification requires the link type to be of an allowed form, such as provider-child, which poses an issue. Firstly, the interfaces of the final hopfield (no further hopfields are involved for local destinations) do not form a valid link type. Secondly, the implementation imposes fewer constraints compared to the IO specification and consequently does not entirely align with the IO specification.

This challenge required changing IO specification. It turned out that this was a corner case which was mishandled by the IO specification. Now the IO specification handles internal packets differently than external packets. While these changes only minimally affected the verification of the codebase, fixing all the proofs on the protocol side required a substantial amount of time. Although this task was outside the scope of the project, it is still worth mentioning here.

3.4 Providing Witnesses

The IO specification is fairly large and contains numerous constraints, primarily expressed through existential quantifiers. Unfortunately, existential quantifiers are known to cause performance issues in SMT-based verifiers. Therefore, it is necessary to replace them with witnesses. This approach offers the added benefit that many of the constraints are satisfied by construction, meaning that they are automatically fulfilled by the witness on the call side. Subsequently, determining which constraints require verification and which are trivially true became necessary. To achieve this, a small example resembling the actual codebase but with most of the functionality abstracted away was created. This approach allowed testing the necessary annotations with a verification time of only a few minutes instead of a few hours. Furthermore, this approach had the advantage of ensuring the correctness and satisfiability of the IO specification.

3.5 Proving IO guards

Incorporating the extensive ghost code from the above-mentioned example directly into the codebase would compromise both maintainability and performance due to its size. To circumvent this issue, a significant portion of the ghost code is encapsulated within ghost methods that are subsequently invoked by the codebase. For that purpose, multiple functions were written to specify the processing of a packet in terms of how they influence the abstract representation of a packet. This approach allows for reasoning only about the changes to the abstract packet, while the equivalence between the actual bytes of the packet and the abstract packet can be proven in separate lemmas. This splits the proof obligations into smaller chunks, resulting in improved performance.

3.6 Performance Issues

Given the size of the project, performance issues are inevitable. These issues often emerge unexpectedly, either due to unfolding complex predicates or by excessively increasing ghost code within a method. To address this challenge, Gobra introduced `opaque` functions. This feature enables the concealment of information within a pure function, thereby reducing the workload of the verifier. Whenever the concealed information is required, it can be explicitly accessed via the `reveal` keyword. While this approach offers significant performance benefits, it also places an additional burden on the user, as the verification could fail due to missing information.

3.7 Importance of triggers

Working with sequences requires a lot of quantifiers, which pose challenges in deductive verification due to their incompleteness. While identifying an initial working trigger is a critical first step, the pursuit of more efficient triggers can lead to significant performance enhancements. This approach not only reduces overall verification time but also plays a decisive role in determining the successful completion of a proof. This can be seen in the IO specification. While the original trigger sufficed, its dependency on the heap led to performance issues. Switching to heap-independent triggers solved the performance issues and decreased verification time. As a general principle, heap-independent triggers are preferable to heap-dependent ones.

4 Suggestions of improvements for Gobra

Although Gobra is a powerful tool with excellent functionality, several areas could be improved to enhance the user experience further. The following section outlines potential improvements.

4.1 Fine-grained verification errors

When verification fails for a constraint with several conjuncts, it raises the question of which specific conjunct failed. Currently, the only way to determine this is by manually testing each conjunct. While this may not be an issue for smaller problems, it becomes highly time-consuming for projects the size of the IO specification. Gobra could significantly enhance its utility by offering more detailed verification error messages, allowing for a more fine-grained understanding of the failure.

4.2 Reducing parsing time

As a first step, Gobra translates Go code into Viper code. Despite the Chopper allowing verification of individual methods, the entire program is still parsed, type-checked, and translated into Viper.

With increasing project size, this process can take several minutes. This is very time-consuming, especially as methods are verified repeatedly. Implementing encapsulation could significantly decrease this processing time, thereby improving productivity.

4.3 Syntax error reporting

Syntax errors in Gobra can be quite challenging. Certain mistakes, such as forgetting the keyword `in` or mismatched parentheses, can confuse Gobra's error reporting, making it difficult to pinpoint the exact error location. Once a syntax error is introduced, it becomes a daunting task to locate. A more sophisticated parser could significantly improve user experience and productivity. Additionally, Gobra imposes seemingly arbitrary rules regarding newlines and comments, such as requiring the opening brace of a function to be on the same line as the function body. These rules can unknowingly be broken and are time-consuming to rectify.

4.4 Folding

In this project, necessary permissions for slices are concealed within predicates for performance reasons. However, this implies that whenever working with these slices, these predicates must first be unfolded. In many instances, it would be advantageous to have a folding syntax within pure functions. Particularly, proving properties of subslices in pure functions becomes challenging when only the predicate of the original slice is accessible. Even though there are possible workarounds, introducing such a syntax would make managing permissions much more straightforward and efficient.

4.5 Asserting

Once again, it can be found that working with quantifiers is very tricky, especially within pure functions. Within a method, a trigger can always be explicitly forced by asserting it. However, in a pure function, such an option does not exist. Even though an easy workaround is available, it is often resorted to, leading to the question of whether assert statements should be allowed in pure expressions.

5 Conclusion

In conclusion, this work aimed at verifying the IO behaviour of the SCION router ensuring the functional correctness of the router's implementation in alignment with its IO specification. Through the use of Gobra and the Igloo methodology, a structured process was undertaken to translate abstract protocol specifications into verifiable code constraints. This work highlights the complexities involved in aligning theoretical models with practical implementations.

Furthermore, the project illuminated areas for improvement within verification tools like Gobra and Viper. These insights could lead to a more efficient and user-friendly verification process in future projects.

6 Acknowledgements

I would like to thank my supervisors, João Pereira and Dionysios Spiliopoulos for their exceptional support during our countless meetings. Whenever I needed help, they always provided valuable advice. Working with them has been an incredibly rewarding experience.

Additionally, I would like to extend my special thanks to Prof. Dr. Peter Müller and the Programming Methodology Group for granting me the opportunity to contribute to their projects and research.

References

[1] Scion-architecture. Accessed on 2024-04-22. [Online]. Available: https://scion-architecture.net/

[2] L. Chuat, M. Legner, D. Basin, D. Hausheer, S. Hitz, P. Müller, and A. Perrig, *The Complete Guide to SCION.* Springer Cham, 16 May 2022.

[3] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759190224P

[4] Igloo: End-to-end verification of distributed systems. Accessed on 2024-04-22. [Online]. Available: https://infsec.ethz.ch/research/projects/igloo.html

[5] Gobra. Accessed on 2024-04-22. [Online]. Available: https://www.pm.inf.ethz.ch/research/gobra.html

[6] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *European Symposium on Programming*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:18589224

[7] Isabelle. Accessed on 2024-04-22. [Online]. Available: https://isabelle.in.tum.de/

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

[✓ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].]

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Verifying the IO Behaviour of the SCION Router

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Limbeck	Markus

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
March 23, 2024	Markus Limbeck

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard