Verifying the IO Behaviour of the SCION Router

Practical Work Project Description

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

Start: 25th of September

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 authorization [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 to 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,

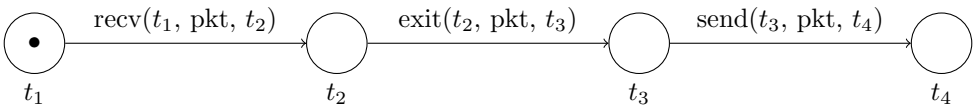

Figure 1: Simple illustrative Petri net.

```
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 visualized 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 an 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.

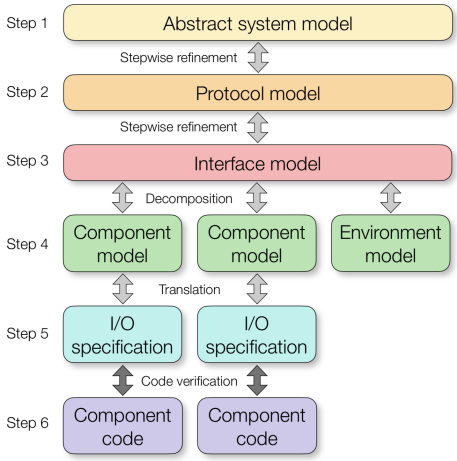

Figure 2: Overview of the six steps in the Igloo methodology. This figure is taken from the Igloo project website [4].

2.3 VerifiedSCION

Igloo allows to formally prove properties like valley freedom or path authorization 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 authorization, on the other hand, ensures that packets are exclusively forwarded along authorized 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 buffer 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.

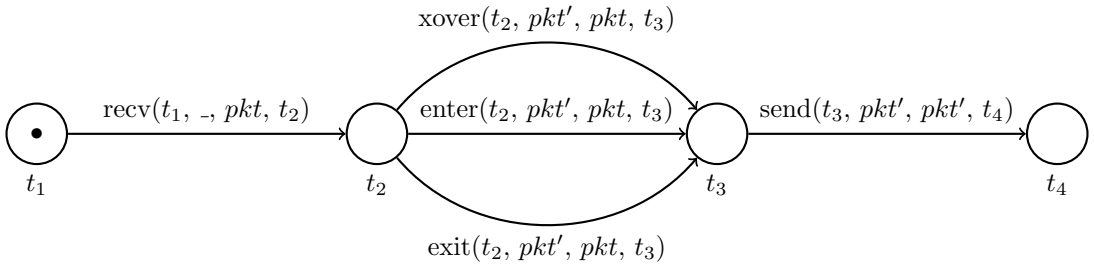

Figure 3: Petri net fof the IO specification.

3 Goals

The overall objective of this work is proving that the implementation satisfies the functional behaviour provided by the IO specification. This is further subdivided into three smaller goals: translating bytes into an abstract representation, annotating the codebase and ultimately verifying against the IO specification. The subsequent section provides a mode detailed explanation.

3.1 Byte translation

The IO specifications do not deal directly with raw packets, i.e., sequences of bytes. Instead, for simplicity, they are written in terms of abstractions of these packets. In order to connect what happens concretely at the network level to the specification, we must first define how to abstract the concrete packets into abstract ones.

At the time of writing, two incomplete versions of abstraction functions have been developed. The first version, while more comprehensive, relies on global information that is inaccessible. This issue is described in detail in Appendix A. The second version utilizes abstract functions to avoid

this challenge. However, these abstractions introduce additional assumptions. Consequently, further effort is needed to ensure that these assumptions do not introduce unsoundness.

Lastly, some IO specification functions still include existential quantifiers. To optimize performance these functions need to be rewritten.

3.2 Code annotation

The next step involves annotating the codebase. This entails identifying the locations where IO events are occurring and adding the IO specification to the main loop's contract.

At the time of writing, two documents have been written that provide guidelines on the usage of IO events and the locations where the guards must be proven in the codebase. The annotation work has been completed for the `run()` method, although the annotations have not yet been proven due to the ongoing memory safety proof. However, additional work is needed for both the `processPkt()` and `process()` methods, for which memory safety has been proven, in order to finalize the annotations.

3.3 Verifying

The final step involves verifying the annotated code supplied by the previous goal. This entails confirming that for every IO event its corresponding guard has been satisfied. This ensures that the implementation aligns with the intended functional behaviour.

At the time of writing, only the `recv` guard has been formally verified, while the remaining four guards are incomplete. It is expected that the majority of time will be dedicated to this goal. Moreover, verifying might uncover shortcomings in the IO specification, which subsequently need adjustments. Some challenges are already identified and described in more detail in Appendix A.

References

[1] Scion-architecture. Accessed on 2023-09-21. [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 2023-09-21. [Online]. Available: https://infsec.ethz.ch/research/projects/igloo.html

[5] Gobra. Accessed on 2023-09-21. [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 2023-09-21. [Online]. Available: https://isabelle.in.tum.de/

[8] Scion documentation. Accessed on 2023-09-21. [Online]. Available: https://scion.docs.anapaya.net/en/latest/protocols/scion-header.html

Appendix A Challenges

- One challenges lies in the disparities between the IO specification and the implementation. In some cases events are handled differently even though they are logically equivalent. This divergence adds complexity to the verification process and will require more time and effort as a consequence. For instance, the IO specification differentiates between segment changes based on the involvement of a core, whereas the implementation treats them uniformly.

- Inbound packets destined for the local AS pose 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 specifications 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. Furthermore, 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 less constraints compared to the IO specification and consequently does not entirely align with the IO specification.

- Another challenge lies in the authentication of hopfields, which differs fundamentally between the implementation and the IO specification. To simplify the following explanation, we assume that a path is used in construction direction. Furthermore, we assume that authentication depends solely on the ingress interface (InIF) and the egress interface (EgIF). While the complete mechanism is somewhat more complex, this simplification serves as a reasonable compromise to illustrate the problem.

  Let us begin by taking a look on how both the implementation and the IO specification authenticate hopfields. In the implementation, a hopfield is represented as a sequence of bytes, as shown in Equation 1.

  $$HF_i := \langle InIF \,||\, EgIF \,||\, \sigma_i \rangle \tag{1}$$

  In Equation 1, the variable σ_i represents a six-byte Message Authentication Code (MAC) utilized for authenticity of the hopfield's content. Authenticity is guaranteed if $\sigma_i = \sigma'_i$, where σ'_i is defined as shown in Equation 2. Here we need a cryptographic function $\texttt{MAC}_{\texttt{key}}()$, which hashes the input using the key K_i provided by the router performing the authentication. Furthermore, β_i is equivalent to $SegID$, a value obtained from the infofield of the packet.

  $$\sigma'_i := MAC_{K_i}(\langle \beta_i \,||\, InIF \,||\, EgIF \rangle) \tag{2}$$

  To enable the next router to authenticate the subsequent hopfield, it is necessary to update the $SegID$. This update is performed as follows: $\beta_{i+1} := \beta_i \oplus \sigma_i[:2]$, where $\oplus$ represents bitwise `XOR`, and $\sigma_i[:2]$ is the truncated version containing only the first two bytes. Subsequently, the $SegID$ is assigned the value $SegID := \beta_{i+1}$.

  It is crucial to note that each hopfield overwrites the previous $SegID$ value, and the old value is no longer available. Detailed information about the hopfield authentication process can be found in the SCION documentation [8].

In the protocol the hopfield is not a sequence of bytes, but an algebraic data type with three fields, as shown in Equation 3.

$$AHF_i := (|InIF \in \mathbb{N}, EgIF \in \mathbb{N}, HVF \in \mathbb{T}|) \tag{3}$$

While the InIF and EgIF remain similar to the previous case, we replace σ_i with a hop validation field (HVF) that belongs to the cryptographic algebra $\mathbb{T}$.

$$\mathbb{T} := \mathbb{N} \,|\, K_{\mathbb{N}} \,|\, \langle \mathbb{T}, \mathbb{T}, \ldots, \mathbb{T} \rangle \,|\, \{\mathbb{T}, \mathbb{T}, \ldots, \mathbb{T}\} \,|\, H(\mathbb{T}) \tag{4}$$

This algebra is defined as shown in Equation 4. It includes natural numbers, e.g., to represent interfaces, a key for each router, finite sequences/sets of algebraic terms, and cryptographic hashes of terms. MACs are defined within this algebra using the hash term of the algebra, specifically as $MAC_{K_i}(m) = H(\langle k, m \rangle)$. Further details about this algebra can be found in the SCION book [2].

Given this definition we can compute σ'_i as shown in Equation 5. Additionally, Equation 6 demonstrates how a router's key is defined by the function $MacKey : \mathbb{N} \to \mathbb{T}$ by using its ASID as input. Furthermore, β_i is equal to the protocol's representation of $SegID$, denoted as $Uinfo \in \mathbb{T}$. It is important to note that, for any i, we require ASID_i to compute σ'_i.

$$\sigma'_i := H(\langle K_i, \langle InIF, EgIF, \beta_i \rangle \rangle) \tag{5}$$

$$:= H(\langle MacKey(ASID_i), \langle InIF, EgIF, \beta_i \rangle \rangle) \tag{6}$$

To update the *Uinfo* field for the authentication of the subsequent hopfield, we need to compute $\beta_{i+1} := \{\sigma_i\} \cup \beta_i$ and assign this new value $Uinfo := \beta_{i+1}$.

The fact that β_{i+1} uses $\bigcup$ to update *Uinfo* causes a problem. In sub goal one of this work, our objective is to define a function $f : HF \to AHF$. This function, given a sequence of bytes that represent a hopfield, should return the abstract version of an hopfield as described previously. Let us assume that we were given the bytes of the i^{th} hopfield. It is evident that $\beta_i = \sigma_{i-1}, \sigma_{i-2}, \ldots, \sigma_1$. Additionally, we have established that the computation of σ_j requires knowledge of ASID_j for all $j \in \{1, \ldots, i-1\}$. This implies that all ASIDs along a path must be known. Unfortunately, this information is not accessible by any individual router, making it unfeasible to construct such a function. While one option is to abstract the entire authentication process and assume its correctness, the ideal one involves the formal proof of its correctness.

- The codebase requires a substantial amount of specifications solely for the memory safety proof. Adding additional conditions to verify the IO specification has the potential to cause performance issues. Furthermore, the IO specification contains numerous mathematical sequences which pose challenges in deductive verification due to their incompleteness. Incorrect triggers can easily cause verification overhead and subsequently performance issues. Based on previous experience performance issues occur rather unexpectedly and require substantial time and effort to resolve.