Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router

João Pereira
ETH Zurich

Tobias Klenze
Category Labs

Sofia Giampietro
ETH Zurich

Markus Limbeck
ETH Zurich

Dionysios Spiliopoulos
ETH Zurich

Felix Wolf
ETH Zurich

Marco Eilers
ETH Zurich

Christoph Sprenger
ETH Zurich

David Basin
ETH Zurich

Peter Müller
ETH Zurich

Adrian Perrig
ETH Zurich

Abstract

We present the first formally-verified Internet router, which is part of the SCION Internet architecture. SCION routers run a cryptographic protocol for secure packet forwarding in an adversarial environment. We verify both the protocol's network-wide security properties and the low-level properties of its implementation. Namely, we develop a series of protocol models by refinement in Isabelle/HOL and we use an automated program verifier to prove that the router's Go code satisfies crash freedom, freedom from data races, and adheres to the most concrete model in our series of refinements. Both verification efforts are soundly linked together.

Our work demonstrates the feasibility of coherently verifying a security-critical network component from high-level protocol models down to performance-optimized production code, developed by an independent team. In the process, we uncovered critical attacks and bugs in both the protocol and its implementation, which were confirmed by the code developers, and we strengthened the protocol's security properties. This paper presents the challenges we faced when verifying an existing real-world system, explains our approach to tackling these challenges, summarizes the main results, and distills valuable lessons for the verification of secure systems, in particular for the techniques and tools employed.

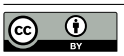
CCS Concepts

• **Security and privacy → Formal security models**.

Keywords

End-to-end verification, Network security, Secure routing, Verified systems

ACM Reference Format:
João Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. 2025. Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765104



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765104

1 Introduction

Faulty software poses a serious threat to the reliability and security of critical computing infrastructures and can lead to catastrophic system failures or devastating attacks. In light of this problem, substantial progress has been made over the past decades on the theory and tools for software verification and several major software verification projects have been completed in the area of operating systems [25, 29], compilers [34, 36], distributed systems [26, 50, 60], security protocols [11, 18, 19], and cryptographic libraries [48, 68].

Given that Internet routers are a central part of our critical networking infrastructure and deployed on a large scale, their security and reliability are of paramount importance. In this paper, we present the first comprehensive formal verification of Internet routers and their protocols.

Our verification effort focuses on the SCION Internet architecture [16], a clean-slate redesign of an inter-networking infrastructure with a focus on strong security and reliability properties. SCION border routers are responsible for packet forwarding, which is simple and efficient, since each packet's path is embedded in the packet header, alongside cryptographic authenticators that authorize the path. SCION has real-world deployment: SCION services and connectivity are offered by over 20 Internet service providers reaching networks on 5 continents [61], and is used, for instance, as the networking layer for Swiss interbank clearing [54].

The security and correctness of Internet routing depend on both *system-wide, global, security properties of the protocol* (e.g., that packets can travel only along previously authorized paths, a property called *path authorization*) and *local properties of the code* (e.g., that the protocol is implemented correctly). Verifying them involved a host of challenges, in particular:

- **C1:** How to model a packet forwarding protocol and verify its security for arbitrary network topologies under a strong attacker model?
- **C2:** How to verify the pre-existing router implementation, a substantial code base that uses complex language features?
- **C3:** How to verify the security and correctness of an existing, deployed router *comprehensively*, from protocol to code?
- **C4:** How to organize the verification effort efficiently, given the size of the project, both of the protocol and codebase?

Challenge C1 concerns the protocol: We must account for the complex adversarial environment in which SCION routers operate. This involves an arbitrary network topology, an arbitrary set of authorized paths, and active and possibly colluding attackers. Moreover,

the protocol is designed to achieve global properties like path authorization and loop freedom. Efficiency mandates that each router performs only local checks related to its own position on a packet's embedded path. Showing that these checks imply the global properties requires stating and proving suitable invariants over recursive data structures that hold despite the attackers' presence. Doing this in full generality requires an expressive specification language like higher-order logic.

Challenge C2 concerns the implementation: The router code was developed independently of our verification effort and without verification in mind, with an emphasis on performance. Accordingly, the SCION router is implemented in roughly 4,700 lines of optimized, concurrent Go code that heavily uses aliasing to minimize memory consumption and copying. Despite significant advances in program verification techniques, verification of complex properties about an entire pre-existing code base written in an expressive language like Go has so far been out of reach. It requires tools that support all the used language features and idioms, express all needed properties, and can do so at scale.

Challenge C3 is concerned with formally integrating these two lines of reasoning to provide sound guarantees from protocol to code. Finally, challenge C4 is concerned with finding techniques and processes to organize a multi-year project into manageable tasks, distributing work over several team members, and dealing with the changes that are inevitable in any software project.

All the verification projects previously mentioned employ techniques that significantly simplify the verification task, such as the co-development of the implementation and its verification, the adoption of a programming language and a software design that ease verification, and code extraction from correctness proofs. However, we verify existing, optimized, and independently developed code, which prevents the application of these techniques and requires a different verification approach.

Verification for networking data planes has so far focused on network configurations [9, 28, 33, 59] or on network functions [38, 47, 64–66]. The former only consider network *models* and their configurations and is mostly restricted to either local properties of single nodes or network-wide properties of fixed, concrete networks. The latter only considers local properties of *implementations*, but no global, network-wide properties. Moreover, all works in this category are restricted to non-adversarial settings.

Approach. To tackle the above challenges, we decompose our overall project into smaller tasks using different forms of modularity. Concretely, we employ the Igloo methodology [56], which soundly combines protocol verification by refinement with code verifiers based on separation logic [51]. To verify the SCION router, we model the protocol and its adversarial networking environment as a labeled transition system in Isabelle/HOL [43]. We start from an abstract protocol with a weak adversary and refine it into more concrete ones with a strong, symbolic (Dolev-Yao [20]) adversary. We build on existing basic protocol models [31, 32] and extend them with bi-directional and multi-segmented paths. We prove network-wide properties: path authorization, valley freedom, and loop freedom. The latter two were not considered in previous work.

We extract a router model from the most concrete protocol model and extend Igloo's tooling to automatically translate it into a program specification that completely describes the router's allowed I/O behavior. We use the Gobra verifier [62] to verify the router's Go code against this specification. This ensures that the code correctly follows the protocol and establishes additional code properties such as crash freedom, and freedom from data races. Since verification is performed statically, the executable code and its performance are entirely unaffected. Igloo's soundness result implies that the protocol properties are preserved for the code executing in its adversarial network environment. Hence, our protocol verification results are seamlessly and soundly extended to the router implementation.

While we have presented the above tasks as sequential steps, we worked on protocol and code verification *in parallel* with two teams using the best tool for the respective task, interfaced only by the generated program specification. This decomposition substantially sped up and simplified our verification. We believe that our approach is useful for other extensive verification projects.

Contributions. We summarize our main contributions.

- We present the first comprehensive verification of a full-fledged Internet router, and indeed of any large networking infrastructure component. We establish system-wide protocol security properties and local code properties. We discovered and fixed vulnerabilities and bugs in the protocol and its implementation that had escaped extensive prior reviews and testing.
- We model the SCION protocol in detail, covering crucial features of the implementation, and verify two new network-wide security properties.
- We demonstrate the feasibility of verifying advanced, performance-optimized production code developed by a different team. This required combining,extending, and scaling up various verification techniques to cover the language features and software designs used by this program, and substantial performance improvements to Gobra to handle the complexity of production code.
- We summarize our experience in verifying a complex system independently of its development and distill lessons learned that can be useful for future verification projects.

Our Isabelle/HOL and Gobra formalization and proofs are available online [1].

2 Background: SCION overview

SCION is a secure inter-domain network architecture, providing connectivity between autonomous systems (AS). The control plane, performing path discovery and dissemination, executes on a service infrastructure, whereas the data plane, providing packet forwarding, runs on border routers. Although this paper presents the verification of the border router, this section provides an overview of the entire architecture.

SCION network. ASes in the SCION network are organized hierarchically, where each AS has bidirectional links to neighboring ASes: *CustProv* links point to provider ASes and *ProvCust* links point to customer ASes. ASes at the top of the hierarchy, i.e., without providers, are *core ASes*, which connect to each other via *Core* links. Fig. 1 shows an example. Every AS assigns to each of its inter-AS

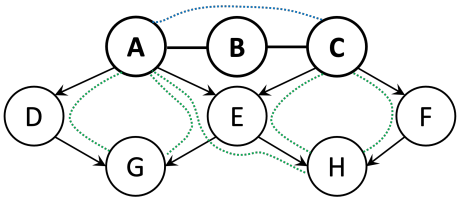


Figure 1: Network topology example. Autonomous systems are linked hierarchically (arrows), except among core ASes (bold). Dotted lines represent authorized path segments.

links an *interface*, which, in combination with the AS identifier, is globally unique. SCION routers are *border routers* as they connect the internal AS network to other ASes via inter-AS links.

Segment construction. The control plane discovers and simultaneously authorizes forwarding paths between ASes. For scalability, paths are established along several *authorized segments*. *Down-segments* are constructed along *ProvCust* links starting at a core AS, and *core segments* along *Core* links. *Up-segments* are obtained by reversing down-segments. Core segments can also be reversed for the traversal in the opposite direction. All segments are authorized by the on-path ASes using nested message authentication codes (MACs), which we will present in Sec. 4.2. Authorizing only certain paths allows network operators to enforce their own policies according to their economic and compliance requirements.

Segments consist of a sequence of *hop fields*, each carrying the forwarding information of one AS. Each hop field contains the interfaces of the incoming and outgoing links (*prev* and *next*), and a cryptographic *authenticator* containing the nested MACs, which the respective router uses to verify that the segment was authorized by the control plane. Crucially, not only the adjacent inter-AS links, but the entire segment is authorized. This rules out subtle *path splicing* attacks [32], in which hop fields from different segments would be combined to craft a new, unauthorized segment, for example crafting path C-E-G without E's consent in Fig. 1.

Segment combination. To send a packet, the source end host combines one or more authorized segments to form an AS-level path from its own AS to the destination host's AS, and embeds this path in the packet header. Such a path can consist of up to three segments: an up-segment from the source AS to a core AS, a core-segment to another core AS, and a down-segment to the destination AS. Typically, there are multiple paths that the source can choose from.

The combination of segments follows rules that protect the economic interests of ASes. One central goal is to avoid *valleys*, where an AS forwards packets from a provider to a provider (e.g., from *A* to *C* via *E* in Fig. 1). Since ASes are paid by their customers, and must pay their providers, such packets only create costs but no revenue for the valley AS.

Forwarding. In the data plane, routers forward each packet according to their hop field's forwarding information after validating the authenticator. Inter-domain forwarding tables are not required since each packet contains its own forwarding state. Routing and forwarding *within* each AS are performed by the AS's intra-domain routing system and are not part of SCION. We verify SCION's forwarding protocol, which we further explain in Sec. 4.

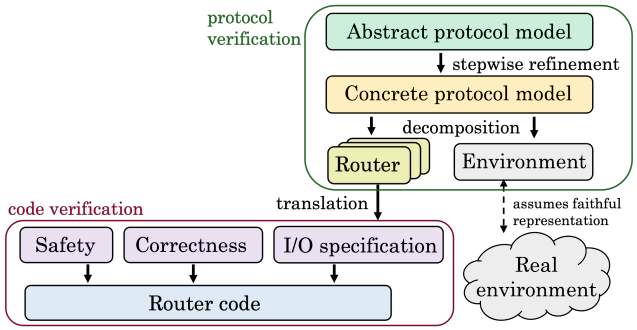


Figure 2: Overall verification approach.

3 Overview of properties and verification

In this section, we summarize the protocol and code properties we establish and the resulting overall security guarantee, present our verification approach (cf. Fig. 2), and state the assumptions on which our verification rests. Sec. 4 and 5 provide more details on the protocol and code verification, respectively.

3.1 Verification guarantees

We proved the central security properties of SCION's data plane protocol. Prior work [31, 32] considered only the first property.

- *Path authorization*: each segment that a packet traverses is contained in an authorized segment.
- *Valley freedom*: packets that already traversed a *ProvCust* link must not traverse a *CustProv* link.
- *Loop freedom*: packets do not follow the same link twice.

These properties protect the honest ASes' interests and rule out malicious end hosts sending packets along unauthorized, uneconomical, or impractical paths. We verify them against a strong attacker model consisting of malicious end hosts with access to the secret MAC keys of a set of *compromised ASes*.

On the code level, we verified the following properties:

- *Safety:* the implementation is well-behaved, that is, it neither crashes nor causes data races.
- *Functional correctness:* the functions compute the intended results, for instance, they perform the required MAC check.
- *Protocol compliance:* the router validates and forwards SCION packets as prescribed by the protocol model.

Thus, our verification ensures that there are no bugs in the code that compromise the security of the overall system, for example, because the code lacks some check prescribed by the protocol.

As explained below, we prove the above-mentioned protocol and code properties separately and soundly link them using the Igloo methodology [56], leading to the following end-to-end guarantee:

Theorem 3.1. *Any SCION network, whose routers execute the verified router code, satisfies the network-wide data plane security properties (under the assumptions from Sec. 3.6).*

Liveness properties (e.g., sent packets are eventually received) would require additional bandwidth reservation [23] and DDoS prevention mechanisms, and are hence out of scope of this work.

3.2 Modular verification approach

A key enabling factor in our project is the decomposition of the overall verification effort into manageable chunks, addressing challenge C4. We use five decomposition strategies:

(1) *Separation of protocol and implementation:* We verify the SCION data plane protocol and its implementation separately and link the two parts soundly.
(2) *Protocol refinement:* We use refinement to develop the protocol model step by step and focus on one protocol aspect at a time.
(3) *Protocol parametrization:* We parametrize our protocol models on the cryptographic mechanisms used to secure segments, allowing us to quickly adapt to protocol changes.
(4) *Layered code properties:* We annotate and verify the implementation in three layers, each focusing on a different aspect of the intended behavior.
(5) *Program modularity*: We use modular program verification, which verifies each function of the program independently.

3.3 Separating protocol and code verification

The first form of modularity mentioned above offers several crucial benefits. First, it allows us to leverage the strongest and most appropriate techniques and tools available for each purpose: we use stepwise refinement in Isabelle/HOL for the protocol (Sec. 3.4 and 4) and automated deductive verification in Gobra for the implementation (Sec. 3.5 and 5).

Second, it enables protocol and code verification to proceed largely *in parallel*. Furthermore, since the code properties are *layered* and the safety and many functional correctness properties are independent of the protocol verification (cf. Fig. 2), verifying those properties allowed us to quickly identify bugs in the pre-existing implementation (and to deliver value immediately).

Finally, there is a clear and explicit interface between protocol and code verification in the form of an *I/O specification*, which we automatically generate from the router model and completely describes the router's intended I/O behavior. By verifying the implementation against this I/O specification, we establish the implementation's protocol compliance. Importantly, code verification does not require any reasoning about the attacker. A soundness result [56] guarantees that the router model and the resulting I/O specification have the same trace properties and, using compositional refinement, also guarantee a trace inclusion between the protocol models and the implementation. Consequently, all properties proved for the protocol model also hold for the implementation, as stated in Theorem 3.1, thereby addressing challenge C3.

3.4 Protocol verification

Security protocol models are commonly verified using state-of-the-art automated protocol verifiers such as Tamarin [40, 52] and ProVerif [12, 13]. These tools are based on the (Dolev-Yao) attacker model, which identifies the attacker with the network. Their network model is unstructured and has no representation of a network topology. While modeling a topology is possible, one would have to limit the topology's size and/or the maximal length of authorized segments to avoid non-termination issues. Moreover, verification time would scale poorly as these limits increase.

Since we aim for much stronger results, which hold for *arbitrary* network topologies and *arbitrary-length* authorized segments, these tools are not suitable for verifying SCION's data plane protocol. We therefore use the general-purpose theorem prover Isabelle/HOL [43], whose higher-order logic specification language provides the necessary expressiveness to construct a fully general protocol model that combines a Dolev-Yao attacker with an arbitrary network topology. We develop the SCION data plane protocol in several steps using *refinement* and *parametrization* to structure our models and proofs [4, 39]. These modeling and verification techniques allow us to tackle challenge C1.

We formalize the protocol models as labeled transition systems, where each label, called an *event*, is associated with a transition relation. A sequence of such events is called a *trace*. We formulate a series of such models at different abstraction levels and relate them via refinement proofs (cf. Fig. 2):

Abstract model: contains all the essential protocol functionality, but only a restricted attacker, and no cryptography.
Concrete model: introduces cryptography to protect the segments, as well as a strong attacker (see Sec. 4.1).
Decomposed model: splits the concrete model into an environment model and a router model, and introduces separate I/O events, corresponding to the implementation's I/O library calls.

We express and prove the desired security properties as *invariants* over the models' state, which we also equivalently express as trace properties. Since refinement guarantees trace inclusion, it preserves trace properties and hence invariants. We can therefore prove each security property at its most suitable abstraction level.

3.5 Code verification

In contrast to most existing verification projects, the SCION router was developed independently from our verification effort and optimized for speed rather than verification (cf. challenge C2), which substantially complicates verification. In particular, the implementation uses most of Go's language features, including some that are difficult to reason about. These include threads, global state, closures, and interfaces, which require advanced reasoning about concurrency, higher-order specifications, and structural subtyping. Moreover, the router implementation minimizes memory usage. The employed memory-efficient data structures and memory reclamation strategies are difficult to reason about and, to our knowledge, have not been considered in other verification projects.

Choosing the right verification tools and techniques is key to addressing these challenges. To enable modular reasoning in the presence of concurrency, we use concurrent separation logic (CSL) [44]. This choice is almost forced on us, as other widespread logics either do not handle concurrency at all or require explicit, non-modular reasoning about thread interleavings that does not scale. To better handle the size and complexity of the code, we perform automated verification based on SMT solvers, which significantly reduces the effort of writing proofs, at the cost of limiting specifications to first-order logic (which, e.g., makes specifying closures challenging).

Thus, to prove that the router implementation is well-behaved, functionally correct, and satisfies the router model's I/O specification, we use Gobra [62], an SMT-based automated deductive verifier for Go. Gobra is based on a variant of separation logic and allows

for *function-modular* reasoning about concurrent programs, that is, it verifies each function independently against a user-written specification. Each function specification consists of a precondition, which expresses constraints on the parameters and states in which a function may be called, and a postcondition, which describes the function's results and its side effects. As usual for deductive verification tools, programmers must supply loop invariants. Before our project, Gobra supported most, but not all, of the language features used in the router implementation; thus, we had to extend it to support the missing features. Crucially, Gobra already supported some advanced separation logic specification constructs not available in many automated verifiers, which significantly simplifies reasoning about some code patterns used by the implementation (see Sec. 5).

Gobra is designed to be sound, i.e., not miss errors. However, it may produce spurious errors (false positives), mostly when the SMT solver fails to prove a valid condition. In these cases, programmers can introduce additional annotations to guide the proof search so that verification succeeds.

3.6 Assumptions

Our guarantees hold under the following assumptions:

Environment. We prove the security of the SCION router based on a model of the environment consisting of a network and an attacker model; we assume that these models cover the behavior of the real environment. We also assume the correctness and security of the control plane. Moreover, we verify the router's source code, but assume the correctness of the compiler, runtime system, operating system, and hardware.

Libraries. The router implementation uses external libraries (Go's standard library, Prometheus [2], and GoPacket [24]), whose behaviors we specified using pre- and postconditions, but we did not verify their implementations. We did the same for auxiliary SCION libraries that perform logging, allocation of error values, and declaration of new gopacket's layer types.

Tool soundness. We assume that the Isabelle/HOL and Gobra verification tools are sound. We connect the protocol model to code via an I/O specification, which we translated manually from Isabelle to Gobra syntax. This simple, mostly syntactic translation is also trusted.

4 Protocol modeling and verification

We describe our modeling and verification of the SCION data plane protocol. We build upon an existing model [31, 32], which covers only the first two levels in Fig. 2 and only considers up-segments, ignoring other segment types and their combination. Furthermore, that model addresses path authorization, but neither valley freedom nor loop freedom. We hence (i) extend this prior model to cover all features of the actual SCION protocol and all desired security properties, (ii) add the third level in Fig. 2, and (iii) extract the I/O specification for the code verification, tripling the size of the prior work. Moreover, for the new protocol attacks that we identified (see Sec. 4.5), all but one rely on the additional protocol features.

4.1 Environment model

Network and control plane model. The state of our transition systems consists of packets in the sets $int(A)$, the internal network of AS A, and $ext(A, i, B, j)$, the inter-AS link between interface i of AS A and interface j of AS B. We do not model intra-AS networking, as it is not managed by SCION.

Representing the segments produced by the control plane, we parametrize our model over a set of authorized up-, down- and core-segments. Prior work covered only up-segments.

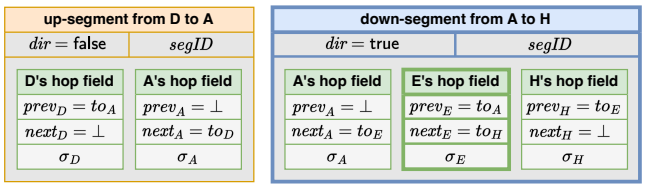


Figure 3: A two-segment packet at hop E of the path D-A-E-H in Fig. 1 (payload not shown). For up-segments, where *dir* = false, the meaning of *prev* and *next* is reversed. The first hop field's *prev* and the last hop field's *next* are empty (⊥). The current segment and hop field have bold frames.

Attacker model. Recall that the verified properties protect honest ASes against malicious senders. We assume that

- a subset of ASes may be compromised, that is, their secret keys are revealed to the attacker, but all routers follow the protocol;
- end-hosts in any AS may be compromised, regardless of whether their AS is compromised or not.

Instead of explicitly modeling compromised end hosts and routers deviating from the protocol, our model accounts for them by allowing the attacker to inject packets anywhere (*int* and *ext*) in the network, not only at compromised ASes.

In our abstract model, which does not (yet) use cryptography, the attacker can freely combine path segments to form packets, but the segments must be authorized. In the concrete model, we lift this limitation and introduce a full-fledged Dolev-Yao attacker [20], who manipulates symbolically represented messages and can eavesdrop on and inject new packets globally. The attacker's knowledge K consists of eavesdropped messages, all authorized segments, and the secret keys of a set of compromised ASes. The attacker's message forging capabilities are modeled as a closure operator $\mathrm{DY}(K)$, which closes the set K under the messages the attacker can derive (e.g., by constructing MACs with known keys) and hence inject. As is common in symbolic models, we assume perfect cryptography. Hence, only protocol flaws relating to the *use* of cryptographic primitives are considered, not flaws in the primitives themselves.

4.2 Protocol models

The abstract and concrete models describe how packets are created, processed, and forwarded.

4.2.1 Packet structure. A packet consists of up to three segments. Each segment consists of a direction flag (*dir*), a *segment identifier* (*segID*), and a sequence of hop fields (cf. Sec. 2). Counters keep track of the current segment and hop field. Fig. 3 shows an example two-segment packet for the path D-A-E-H in Fig. 1.

Packets in the abstract and concrete model differ in their use of the segment identifier *segID* and their hop field's authenticator σ_X.

In the abstract model, where the attacker cannot modify segments, the AS's name suffices as an authenticator, and segment identifiers are not used. In the concrete model, these fields are instantiated with cryptographic values.

Concrete cryptographic authenticators. During segment construction, each AS X on a segment creates a hop field with its local forwarding information (the interfaces $prev_X$ and $next_X$) and authorizes the segment's use by embedding a cryptographic authenticator σ_X in the hop field. This authenticator is a MAC constructed using a local MAC key K_X that is shared between the border routers within AS X:

$$\sigma_X = \mathsf{MAC}_{K_X}(prev_X,\ next_X,\ segID). \tag{1}$$

Here, *segID* is the (mutable) segment identifier, which the first AS on a segment initializes with a random value RND. Each subsequent AS X, after computing its σ_X, updates

$$segID := \sigma_X \oplus segID, \tag{2}$$

where $\oplus$ is exclusive-or (XOR). Each hop field's MAC thus protects the hop's *prev* and *next* fields and also, via nested MACs, all preceding hop fields. During forwarding, AS X reconstructs σ_X using Eq. (1), checks that it matches the hop field's authenticator, and forwards the packet with *segID* updated as in Eq. (2).

In an earlier version of SCION, the MAC directly included the MAC of the previous hop field. This simpler protocol was easier to verify, but it made segment truncation more difficult. In a truncated up-segment, the final router would lack the parent MAC of the last hop field required for authenticator verification, unless it was explicitly included in the segment. The XOR-based solution eliminates the need to include extra MACs for verification, as it folds the accumulated MACs into the *segID* field.

4.2.2 Events. Our models each have a *router event*, which forwards packets, and an *attacker event*, which injects attacker-fabricated packets into the network.

The router event for a router in AS A receives a packet *pkt* either from an external interface i or from the internal network, checks its validity, updates it, and sends it either to the next hop on the path over interface j or the internal network to be processed by a different router. The router uses two validity checks: $ifs\_valid(pkt, A, i, j)$, which ensures the validity of interfaces, in particular that the packet is received from the interface specified in the hop field, and $crypto\_valid(pkt, A)$, which checks the validity of the segment identifier and authenticator (only in the concrete model). When a packet moves to a new segment, the router also enforces the segment combination rules (cf. Sec. 2). In the updated packet, denoted by $update(pkt)$, the counters for the current hop field and, if needed, segment are incremented and, in the concrete model, the segment identifier is updated.

Overall, the checks and updates by the router event depend on: (i) whether a packet is leaving or entering an AS, (ii) whether forwarding to the next AS happens on the same or a different router in the same AS, (iii) the forwarding direction, and (iv) whether segment switching occurs. We must account for all combinations of these factors. For instance, whether the segment identifier must be updated depends on (i)–(iii), whether it is updated before or after computing the authenticator depends on (i) and (iii), and depending on (iv) more than one segment identifier may need to be updated. These case distinctions adds substantial complexity to our models over the prior work, which only considered a single factor, namely (i). All of the protocol attacks discovered in this work relate to missing checks in subtle combinations of (i)–(iv) (see Sec. 4.5).

In the abstract attacker event, the adversary can create packets consisting of only authorized segments, but combined arbitrarily. In the concrete event, we instead allow the attacker to inject arbitrary derivable packets from the set $\mathsf{DY}(K)$ (anywhere in the network). These events also cover the case of honest senders, as all authorized segments are known to the attacker.

4.3 Protocol verification

We express the security properties from Sec. 3.1 as invariants of the packets' network traversal history, which we record in the packets themselves as a separate *history* field for each segment. This history is just for specification purposes and cannot be manipulated by the attacker. More precisely, the path-authorization state invariant expresses that for all packets in a state, each of their segment's history belongs to the set of authorized segments. We close this set under prefixing and suffixing to allow for partial traversals. The loop-freedom invariant expresses that each inter-AS link $ext(A, i, B, j)$ traversed in the concatenation of the packet segments' histories is unique, while valley-freedom expresses that these links follow an up-core-down order. These properties hold for honest ASes, protecting them from malicious end hosts colluding with on-path ASes. As in prior work [32], they do not hold for compromised ASes. For example, for loop freedom to hold, there must be at least one honest AS in the loop.

We prove the above invariants for the abstract model, where they are easiest to prove. Since the abstract attacker can only combine authorized segments, path authorization is trivial. This allows us to focus on valley and loop freedom, proving that each abstract event preserves these two invariants. Throughout the proof, we use technical auxiliary invariants regarding the format of packets and hop fields: for example we prove that the interfaces of adjacent hop field match, unless the ASes at both ends of the link are compromised.

For the refinement, we must relate the abstract and concrete states and events, and show that all concrete events have an abstract counterpart. The challenge here is to show that the refinement preserves intra-segment path authorization, since the concrete attacker can seemingly do much more by sending arbitrary derivable packets instead of just authorized segments. Once established, the refinement preserves all three main properties (actually, all trace properties). Moreover, the protocol uses XOR, which is notoriously difficult to reason about [3, 22]. Below, we first explain how we handle XOR and then describe our refinement proof in more detail.

XOR model. Rather than representing XOR using equational theories, we adopt a normal form representation of symbolic terms [32], where two terms equal modulo the XOR equational theory have identical normal forms. The normal form of the XOR of different terms is represented as a finite set, and the XOR of such sets corresponds to their symmetric set difference. By ensuring that all terms are in normal form, it becomes sufficient to check for equality of finite sets rather than reasoning over equational theories. In line

with prior work [32], we over-approximate the attacker's XOR capabilities by allowing the attacker to extract x and y from $x \oplus y$, which greatly simplifies reasoning. This is a sound approximation that suffices for our setting, since the dataplane protocol uses XOR only as an accumulator function, but not for secrecy.

Refinement proof. Our refinement proof proceeds in two steps. First, we observe that while in the concrete model, the attacker can send arbitrary derivable paths, forwarding only occurs if the hop fields are valid. We hence show that we can shorten concrete paths to their longest valid prefix in the packet abstraction function. Second, we show that this shortened cryptographically valid path is authorized for any packet that the adversary can derive. For this, we extract the path from a single hop field's MAC, following its nested structure, and show it is authorized. Our proof involves customized induction schemes, tailored to sequences of hop fields and further auxiliary invariants, e.g., that the intruder's knowledge remains constant across all reachable states of our concrete event system.

4.4 Linking protocol to code verification

To transition from protocol to code verification, we decompose the concrete protocol model into a model for each router and an environment model. We show that the composition of these models refines the original concrete model. In this refinement, we introduce buffering to separate the router's I/O operations from its internal message processing, in order to subsequently map these operations to the implementation's I/O library calls.

We automatically generate an I/O specification from the router model, along with a correctness proof showing its trace equivalence with the router model. We then manually translate this I/O specification into Gobra's specification language, which is straightforward. Our code-level verification includes proving that the router implementation conforms to this specification, see Sec. 5.

To relate the messages at the protocol level with those in the implementation, we assume a function α mapping bitstrings to terms, similarly to [56, Sec. 4.2]. The existence of such a function is a standard assumption in symbolic protocol verification. It represents the common abstraction of treating bitstring messages as well-typed terms under a perfect cryptography assumption, which precludes, for instance, collisions between different message types and cryptographic collisions (e.g. MACs). We also assume that the cryptographic operations at the code level correctly implement the corresponding abstract operations by relating them using α.

In SCION, MACs are required to use pseudorandom functions (PRFs), making collisions unlikely. The situation is more complex for XOR, as it is trivial for the attacker to craft XOR collisions. However, in SCION, XOR is used only in the segment identifiers, which have the form $MAC_k(a, X) \oplus X$. Since XORing with a random value yields a random result, the problem of finding a collision for such terms again reduces to the PRF security of the underlying MAC scheme. Furthermore, an attacker would need to find a collision such that the resulting hop fields have valid interfaces that can be used for forwarding. Nonetheless, the inability of symbolic models to reason about collisions reflects their inherent limitations. Cryptographic proofs avoid this issue, but they are often difficult to formalize and scale for complex protocols. Computational soundness could bridge symbolic and cryptographic models, but this is impossible for most symbolic models that include XOR, including ours [58].

ID	Vulnerability description	Fixed
V1	Path segment limit not enforced.	✓
V2	Routers can switch from a segment in the down direction to a segment in the up direction, thus creating a valley.	✓
V3	Non-core ASes can switch between segments in the same direction.	✓
V4	Core ASes fail to enforce segment constraints when switching, allowing multiple *Core*-type links.	✓
V5	No mechanism to stop forwarding after hop fields marked for verification only.	✓

Table 1: List of protocol vulnerabilities.

Attack	V1	V2	V3	V4	V5	P1	P2	P3
1. Inter-segment path splicing: attacker appends an up-segment to another up-segment (or down-down).			•			X		
2. Loop / traffic reflection: attacker sends packets in a loop among core ASes, causing congestion.	•		•	•		X		X
3. Arbitrary source routing: attacker picks arbitrary forwarding paths.	•	•	•	•		X	X	X
4. Splicing: attacker combines hop fields from different segments.					•	X		

Table 2: Protocol attacks and their required vulnerabilities (V1–V5) and violated properties: P1—Path authorization, P2—Valley freedom, P3—Loop freedom.

4.5 Results

Artifacts. Overall, we produced several formal protocol models of packet forwarding in SCION, together with their execution environment and attacker model. This Isabelle/HOL development consists of 16,100 LoC, containing over 1,000 lemmas, and substantially extends the existing SCION formalization which consisted of 5,500 LoC [32], with around 450 lemmas. Overall, our Isabelle formalization takes five minutes to verify on a standard laptop. We estimate that this development took two to three person years.

The protocol formalization and proofs guarantee that packet forwarding in SCION as deployed today is secure, even in the presence of a strong attacker. We expect them to be useful also to verify SCION's control plane protocols and during SCION's future evolution, for instance, to assess the impact of protocol changes on the intended security properties.

Attacks found and protocol improvements. During our verification effort, we found five previously unknown protocol vulnerabilities, giving rise to four novel attacks, see Tables 1 and 2. Three of the vulnerabilities are related to the subtle edge cases in the segment switching logic. All vulnerabilities were confirmed and resolved. The most severe attack (Attack 3 in Table 2) allowed a malicious sender to fabricate arbitrary forwarding paths, thus violating all three security properties. This attack exploited missing checks in

the way that segments are combined and in the limit of the number of segments. An attacker could hence craft a packet that follows an arbitrary path by using a dedicated segment for each individual hop. This attack could be used to send packets in a loop for a bounded number of iterations, causing congestion and network failures.

The vulnerabilities were resolved in updates to the protocol. The updates added several validation checks in the segment switching logic (addressing **V2-V4**), removed the need for hop fields used solely for verification purposes (addressing **V5**), and enforced a limit of three segments (addressing **V1**). The updated protocol is the one presented and verified in this paper.

Furthermore, we discovered an additional check that routers could perform (checking link types *within* a segment), which allowed us to prove a stronger valley and loop freedom property. Originally, SCION routers checked for valleys only when switching between segments, assuming that the control plane would only construct valley-free segments. While carrying out our formal proofs, we noticed that two compromised ASes, one at each end of a segment, were enough for an attacker to extend the segment with a valley hop and craft a two-segment packet with a loop passing through the honest ASes in the middle of the segment. Our proposed intra-segment link type checks guarantee that a valley or loop can exist only if *all* on-path ASes are compromised, as opposed to *at least some* AS. The developers implemented our recommendation.

5 Code verification

In this section, we explain how we verify that the router implementation satisfies the properties listed in Sec. 3.1, namely safety, functional correctness, and protocol compliance.

We verify the intended properties by annotating all functions in the router implementation with pre- and postconditions and use Gobra to check that the function implementations satisfy their specifications for all possible executions. Annotations are expressed in special comments that are recognized by the verifier, but ignored by other tools like the compiler. Since verification in Gobra is modular, we can split the overall verification effort into small tasks, which can be tackled in parallel (modularity aspect 5 in Sec. 3.2).

We verify the code in three main steps (modularity aspect 4 in Sec. 3.2). Step 1 focuses on safety, that is, the implementation neither crashes nor causes data races. This step guarantees in particular the absence of common coding errors such as de-referencing a `nil` pointer or accessing an array out-of-bounds. Step 2 verifies functional correctness, that is, that all functions compute the intended results. This entails strengthening the pre- and postconditions from step 1 to express and prove the additional properties. Step 3 proves protocol compliance, that is, that the implementation satisfies the I/O specification obtained from the router model. This step relies on the properties proved in step 2, for instance, to show that each packet forwarded by the router was previously validated and its headers updated according to the SCION protocol.

We perform these steps by progressively strengthening the function specifications in a tight feedback loop with the verification tool. Gobra is integrated into the IDE. Verification errors are reported in terms of the original Go code and its annotations; the programmer is not exposed to the underlying verification logic. We also added Gobra to our continuous integration workflow to re-verify the implementation on every change.

```
1 //@ req PktMem(pkt) && DataPlaneMem(d)
2 //@ req ifs_valid(Abs(pkt),d,i,j) && crypto_valid(Abs(pkt),d)
3 //@ req Token(l0) && IOSpec(l0, s0)
4 //@ req pkt ∈ s0.inputBuffer[i]
5 //@ ens PktMem(pkt) && DataPlaneMem(d)
6 //@ ens update(old(Abs(pkt)), Abs(pkt))
7 //@ ens Token(l1) && IOSpec(l1, s1)
8 //@ ens pkt ∈ s1.outputBuffer[j]
9 func process(pkt,d,i,j /*@,l0,s0 @*/) /*@ (l1,s1) @*/
```

Figure 4: The specification of `process` expresses safety (Lines 1 and 5), functional properties (Lines 2 and 6), the I/O spec (Lines 3 and 7), and properties about the I/O abstract state (Lines 4 and 8). `req` and `ens` declare pre- and postconditions; `old` allows a postcondition to refer to a pre-state value.

The router implementation uses complex language features that are hard to reason about and therefore often avoided when code is written with verification in mind. To be able to verify the existing router implementation, we had to develop novel verification techniques for the features used in the original SCION code base and implement them in Gobra. In particular, we developed a novel automated solution for specifying and verifying closures with captured state and side effects. Additionally, for the first time in a Separation Logic verifier, we added support for modularly specifying invariants on global variables, which are established by the package initializers and maintained by the rest of the program. We also extended Gobra to support many features of Go that did not require fundamentally new solutions but did require substantial engineering efforts, e.g., support for the built-in types of Go and its unique control structures (e.g., `defer` statements and `for-range` loops), and we added support for (ghost) mathematical types to allow for cleaner specifications. For these extensions, we greatly benefited from the fact that Gobra is based on an intermediate verification language, Viper [42], such that additional Go features can be supported by devising an encoding into Viper and leveraging its existing proof automation. The extensions are now available in Gobra, substantially strengthening its practical applicability.

Below, we provide more details on the three verification steps outlined above, discuss how we mitigate tool performance challenges that we encountered in all three, and summarize our results. We illustrate these steps on the function `process` in Fig. 4, which is a simplified version of the router implementation's function `process`. It takes as arguments the packet `pkt` to process, a structure `d` representing the router's configuration, as well as the ids of the input and output buffers, `i` and `j`. It is called after the interface and cryptographic checks. The function implementation, which we omit for brevity, performs the update of the router model's packet processing event (see Sec. 4.4). We explain the function's specification below.

5.1 Verifying safety

Gobra uses implicit dynamic frames [55], a variant of separation logic [45, 51]. The expressiveness of this logic is crucial to tackle the concurrency and complex heap structures in the SCION code base and to obtain the modularity needed for a verification effort

of this scale. Using this logic, Gobra checks that programs do not crash and do not exhibit data races, as we explain next.

Permissions. Separation logic expresses ownership of memory by associating a *permission* with each location, which is created during allocation and can be transferred between different function executions. Gobra checks that a function may access a location only if it holds the corresponding permission; otherwise verification fails.

Permissions are a powerful reasoning principle that Gobra uses for several purposes. Since there is only one permission for each location, it is not possible for two threads to access a location simultaneously, which rules out data races and thus allows us to reason about code without having to consider interference by other threads. Gobra's *fractional permissions* [14] allow concurrent read accesses while still ensuring exclusive writes. Moreover, allocating an array creates one permission per array slot. Any out-of-bounds access is detected by Gobra since the permission for the (non-existent) slot is not available. Finally, permissions allow modular reasoning about side effects. As long as a function holds on to the permission for a location, no other function can possibly modify this location. This allows Gobra to preserve properties of locations across calls.

A function's pre- and postcondition express which permissions the function expects from its caller and which it returns upon termination (see Lines 1 and 5 in Fig. 4). Predicates such as `PktMem` group together the permissions of entire data structures.

In Go, threads are typically synchronized using channels and locks. To reason about thread interactions, Gobra allows both channels and locks to be associated with an invariant. A channel invariant expresses properties of the messages sent over the channel and may also include permissions, such that a message may transfer permission to a memory location. Locks are handled analogously.

Complex code patterns. A key difficulty in verifying an existing code base that was not written with verification in mind is the use of complex code patterns that improve performance, but complicate reasoning. We give three examples and explain how we tackle them.

First, packets have two data representations, as a bytestring and as a struct. Conceptually, bytestrings are unmarshaled into the struct, then processed, and then marshaled back to the bytestring. However, the details are more complicated because (1) these three steps are interleaved, such that the implementation effectively uses both representations, and (2) struct fields store views of portions of the bytestring (so-called *slices* in Go), such that both representations share memory. As a result, it is difficult to specify permissions and the relevant invariants in a purely recursive manner as is standard in automated verification tools. Our solution instead uses iterated separating conjunctions (ISCs) [41, 51] and magic wands [17, 51, 53], two separation logic constructs that enable non-recursive specifications. ISCs represent permissions for every value of a quantified variable and are commonly used to specify permissions to random access data structures like arrays (e.g., the ISC `forall i :: 0 <= i < len(a) ==> acc(a[i])` gives permission to every element of the array `a`). Magic wands are separating implications commonly used to express permissions to partial data structures (e.g., the wand `tree(x) --* tree(root)` represents the permissions to the tree starting at `root` with the exception of the subtree starting at `x`). ISCs allow us to describe how the struct's fields overlap with the bytestring, whereas magic wands enable us to preserve information when there is a short-lived overlap between bytestrings and structs. Both of these features, while standard in the theory of separation logic, are very rarely supported in *automated* separation logic verifiers. We have found them (ISCs in particular) indispensable not only for their traditional use cases like specifying arrays and partial data structures, but also due to the additional flexibility they gave us for specifying other code patterns like these overlapping struct representations.

Second, to avoid memory allocation and reduce the time spent on garbage collection, the router maintains a pool of allocated structs, for instance, to represent SCION paths. Whenever a path is created, a struct is retrieved from the pool, and returned afterward. Verification must ensure that structs do not continue to be used after being returned to the pool, and that they are properly initialized before they are re-used. We re-use Gobra's existing permission system to prove this property: When a struct instance is returned to the pool, we prove that all permissions to it are returned to the pool. Thus, it can no longer be used by other code, since that would require those same permissions that are now no longer available.

Third, the router implementation makes extensive use of interfaces for subtype polymorphism. To verify code that uses these interfaces, we have to annotate each interface with specifications that must be satisfied by each implementation of the interface. Due to Go's structural subtyping, one cannot easily determine all implementations of an interface. This makes devising interface specifications challenging, as one must anticipate the behavior of each interface implementation in the entire code base. As a result, we frequently adjusted interface specifications in the SCION libraries as we verified new implementations that were not compatible with the previous version of the specification. Each such change required re-verifying the already-verified interface implementations against the adjusted specification. This highlights a limitation of modular verification when applied to existing code.

5.2 Verifying functional correctness

Functional properties describe the result and the state updates performed by a function. Even if a function's *implementation* operates on low-level data structures such as bytestrings, its *specification* needs to express the functional properties in terms of the logical values that the low-level data structures represent. This *data abstraction* is essential to making specifications human-readable and connecting them to the state of the protocol model.

The implicit dynamic frames logic used by Gobra allows programmers to express data abstraction via mathematical functions, written as side-effect-free Go functions, that map a concrete data structure to its logical representation. In contrast to standard separation logic, these *pure* functions decouple permission reasoning from reasoning about the values stored in memory. This allowed us to incrementally *extend* the specifications for safety properties discussed earlier with functional properties. The SCION implementation already contained numerous pure functions; using those in specifications substantially reduced the annotation overhead compared to standard separation logic.

In our example, the pure function `Abs` abstracts the concrete representation of a packet `pkt` to a message in the protocol model. This function is used in the precondition of `process` (Line 2 in

```
1 pred IOSpec(l,s) {
2   (forall i, pkt :: pkt in s.outputBuffer[i] ==>
3     Send(l, i, pkt, ?l2) &&
4     IOSpec(l2, s[outputBuffer[i] -= pkt])) &&
5   ... // other I/O permissions
6 }
```

Figure 5: The core predicate of the I/O specification generated from our router model. The predicate includes permissions for all I/O operations prescribed by the model (we show only the Send permission here). The recursive predicate application shows how the position in the protocol (l) and the model state (s) are updated when an I/O operation is performed. Variable $?l_2$ is existentially quantified.

Fig. 4) to express that the packet header has been validated. The postcondition at Line 6 states that `process` modifies the packet as prescribed by the router model's *process* event. Crucially, specifying these functional properties in the contract of `process` does not require changing the safety specifications previously added.

This step specified and proved critical properties of functions that parse, serialize, and manipulate the headers of the SCION packets. Many of these functions are part of the API of SCION. As such, our security guarantees apply not only to the router implementation, but also to third-party clients of the SCION libraries.

5.3 Verifying protocol compliance

To verify that the router code implements the protocol correctly, we prove that each I/O operation performed by the router is permitted by the abstract router model. To reason about I/O behavior, we associate a separation-logic permission with each I/O operation that may be performed by the implementation. We annotate the relevant I/O operations of Go's I/O library to require and consume the corresponding permission. Consequently, a caller may perform the I/O operation only if it holds the permission; otherwise verification fails. This approach was originally proposed by Penninckx et al. [46]; we handle concurrency differently to make the router model and I/O specification independent of the concurrency used in the implementation (see below).

I/O specification. The I/O specification generated from the router model (see Sec. 4.4) is a separation logic formula that provides the I/O permissions for the entire router, that is, expresses which I/O operations the router may perform, their arguments, and in which order they may occur. Fig. 5 shows the definition of the predicate `IOSpec`, the core of the I/O specification. The predicate's parameter `s` represents the router model's state; its relation to the router's concrete data structures is expressed via abstraction functions (Sec. 5.2). `IOSpec`'s other parameter, `l`, is a *protocol location* that indicates the execution's current position in the protocol; it is used to specify valid sequences of I/O operations and is advanced whenever the router performs a protocol step, such as sending a packet.

An individual I/O permission is parameterized with the protocol locations before and after the operation, as well as the parameters and results of the operation. For instance, Line 3 in Fig. 5 provides the permission to send packet `pkt` over the AS interface `i`, provided that the packet is in the corresponding output buffer (Line 2) and the protocol is at location `l`. Performing this operation advances

```
1 //@ req SktMem(conn) && PktMem(pkt)
2 //@ req Token(l1) && Send(l1,conn,pkt,?l2)
3 //@ ens SktMem(conn) && PktMem(pkt)
4 //@ ens Token(l2)
5 func (conn) Write(pkt /*@, l1 @*/)
```

Figure 6: A simplified specification of the `Write` function of Go's I/O library. The function sends packet `pkt`; `conn` is the network socket representing the AS interface that the packet is sent over. Lines 1 and 3 specify permissions for the packet and socket. Lines 2 and 4 specify I/O permissions.

the protocol to location l_2. The recursive instance of `IOSpec` at Line 4 lets the router perform the next I/O operation. Its parameters reflect the new protocol location as well as the updated router state, where `pkt` is removed from the output buffer. The condition under which a `Send` operation is permitted and the modification of the router state are extracted automatically from the guard and update, respectively, of the corresponding event in the router model.

The router's full I/O specification, extracted from the router model, is `Token(l) && IOSpec(l, s0)`, where s_0 is the initial abstract state and `Token(l)` indicates the initial protocol location. This specification, which is a precondition of the router's entry-point, describes the permitted I/O operations of the entire router.

Verifying I/O properties. Fig. 6 shows a specification of the `Write` function from Go's I/O library, which forwards messages to the network. In addition to the memory permissions specified at Line 1, calling `Write` also requires the current protocol location to be l_1 and a `Send`-permission to send the packet `pkt` over the AS interface's network socket `conn` (Line 2). To verify the following call to `Write`

```
/*@ l1, _ := @*/process(pkt,t,conn_in,conn_out /*@,l,s @*/)
conn_out.Write(pkt /*@,l1 @*/)
```

we use the postconditions at Lines 7 and 8 of `process` (Fig. 4), together with the definition of `IOSpec`, to obtain the I/O permission required by the `Write` operation.

An attempt to call `Write` *without* first calling `process` would fail for two reasons: First, we could not establish `Token(l1)`, i.e., the execution would be at the wrong protocol location. Second, we could not obtain the I/O permission from `IOSpec` because the condition `pkt in s.outputBuffer[i]` would not hold, i.e., the router would not be in the expected state. This illustrates that verification enforces that both the sequence of I/O operations and the evolution of the router state precisely follow the protocol model.

Concurrency. The router implementation spawns multiple threads, each of which receives packets, processes them, and forwards the result. The router model, from which we extract our I/O specification, is agnostic to the concurrency used in a concrete implementation. Hence, the I/O specification is parametrized with a *global* router state (`s` in Fig. 5) and describes how this state is affected by each operation. In the implementation, the receive, process, and send operations may occur concurrently, while operations *on the same* packet are sequenced via constraints on the state (e.g., Line 2 of Fig. 5 ensures that `pkt` is received and processed before it is sent).

Due to this global nature of the I/O specification, verification must ensure that each thread may obtain I/O permissions and that

all threads collectively maintain the global state used in the I/O specification. To this end, we treat the I/O specification like a shared data structure in concurrent separation logic. This data structure consists of (1) global ghost (that is, verification-only) variables `cL` and `cS` for the current protocol location and the current model state, and (2) an invariant `Token(cL) && IOSpec(cL, cS)` for the current token and I/O specification. We enforce that each protocol step performed by a thread executes atomically and preserves the invariant [27]. In this atomic step, a thread may use the token and `IOSpec` predicate from the invariant (e.g., to perform an I/O operation), update `cL` and `cS`, and then release the token and `IOSpec` predicate to make them available for the next thread.

This novel way of verifying I/O properties for concurrent programs does not restrict the concurrency used by the implementation and allows the protocol models and the extracted I/O specification to be completely agnostic of the implementation's concurrency structure, which further increases the modularity of our approach.

5.4 Taming verification performance

To our knowledge, this project contains one of the largest verification efforts ever carried out in a deductive verifier based on separation logic, and the largest ever carried out in Gobra. The size and complexity of both the verified code and its specifications repeatedly led to scalability issues with Gobra, resulting in verification times that were too long to make effective progress. This was the single biggest challenge for the code verification.

Most performance problems we ran into could be attributed to one or both of the following causes:

(1) *Long functions:* Gobra verifies code using symbolic execution, which verifies each path in the control flow and each case in the specification (e.g., for disjunctions) separately, leading to a potentially-large number of *branches*. The time required to verify a function depends mostly on the size of the function, the number of branches, and the complexity of the properties to be verified. The SCION router implementation contains several large functions (e.g., the function `Run` has 96 LoC and `prepareSCMP` has 101 LoC) with many branches, for which verification was too slow.

(2) *Large verification context:* While we verify each function separately, the effort for the verification tool is impacted by the entire program's *verification context*, which includes user-provided type declarations and invariants, as well as mathematical axioms generated by Gobra to model Go's built-in types such as slices, structs, and interfaces. Large verification contexts have a severe impact on verification performance because the resulting global declarations and (quantified) axioms in the proof obligations can overwhelm the SMT solver by causing too many quantifier instantiations.

To mitigate performance problems, we extended Gobra with three novel features to provide finer-grained control over the verification context and over the verification algorithms used by Gobra.

First, since our goal is to verify the existing SCION router as is, we cannot refactor long functions into several shorter ones. To achieve the same effect for verification without changing the code, we designed a new code annotation (called *outline statement*) that causes verification to handle a code segment as if it was extracted into a separate function. Analogously to functions, outline statements have pre- and postconditions and are verified separately (i.e., modularly), such that they split the verification of a function into smaller logical parts without actually changing the implementation. As a concrete (contrived) example, consider the following code:

```
x.f = 5;
y.f = 6;
assert x.f < y.f;
```

Pretending that the second statement was difficult to verify, we could extract it as follows:

```
x.f = 5;
// @ requires acc(y.f)
// @ ensures acc(y.f) && y.f == 6
// @ outline (
y.f = 6;
// @ )
assert x.f < y.f;
```

This feature allows for fine-grained control over the proof context, as (1) the preconditions of outline statements can be used to prune the proof context for the outlined statement, and (2) quantified assertions that are established and used only in an outline block are not included in the proof context of the entire function.

Second, we developed a pre-processing step for Gobra that conservatively identifies and prunes irrelevant definitions and axioms in the verification context before verifying a function.

Third, we modified Gobra to give users more fine-grained control to select or fine-tune the verification algorithms used on a per-function basis. It is therefore possible, for example, to use more efficient but less complete algorithms for reasoning about the heap by default (e.g., the SE-PS algorithm [21]), but fall back to slower but more precise algorithms if needed (e.g., the SE-PC algorithm [21]).

These three features improved Gobra's performance substantially and enabled the successful verification of the router.

We experimented with multiple additional optimizations that revealed interesting tradeoffs between performance, completeness and verification stability. An example is the support for verifying all branches of a function in parallel rather than sequentially. Despite improving verification performance, this feature caused verification instability (i.e., non-deterministic reports of spurious errors), likely because the verifier accumulated declarations and definitions across branches, thus enlarging the verification context, which is a known cause of verification instability [67]. We resolved such trade-offs generally in favor of verification stability and completeness because they are essential for making steady progress in the verification.

The performance issues we encountered are general problems that, given sufficiently complex code and specifications, are likely to occur with other automated verifiers as well. Since the overall strategies we used to mitigate them are not specific to Gobra or separation logic, we believe they will also apply in other contexts and may be useful to other verification projects in the future.

5.5 Results

We equipped the router's implementation with annotations that allow Gobra to prove safety, functional correctness, and compliance with the protocol model. We targeted the current, open-source implementation, which consists of 4,700 lines of Go code (ignoring comments and empty lines), including SCION-specific libraries, but excluding third-party libraries like the Go standard library or the library `gopacket`. We achieved our goal of verifying the deployed,

Issue description	Type
Missing bounds check when reading a host address from a bytestring	S
Latent race condition when processing packets with empty paths	S
`nil`-pointer dereference when reversing an empty SCION path	S
Improper error handling when forwarding to the internal network	F
Missing checks after decoding a SCION path	F
Incorrect error reporting when reversing paths	F
Incorrect accumulation of metrics about processed packets	F
Missing checks for the number of hop fields in a SCION path	F
Packets simultaneously originating from, and bound to the internal network are not rejected	P
Missing checks for the validity of the size of path segments	P

Table 3: Selected issues reported in the official SCION repository on GitHub.

performance-optimized implementation as is, except for three small and local changes to work around limitations of Gobra. First, we rewrote one type declaration that uses a specific combination of Go's interfaces and Go's delegation mechanism that is not supported by Gobra. Second, we split some compound expressions, which allows us to add necessary annotations about intermediate results. Third, we rewrote some `range`-loops into regular `for`-loops, which in some cases simplifies permission-based reasoning.

Altogether, we fully verified 345 functions across 12 packages. For three additional functions, we verified only the paths that are relevant here; the unverified paths deal with error reporting or SCION extensions that are not part of our project.

In total, we added 16,700 lines of specifications and annotations, including 1,000 for the I/O specification and the definitions it depends on. We wrote another 2,600 lines of trusted specifications for the Go standard library and third-party libraries. The overhead of 3.6 lines of annotation per line of code is typical for SMT-based deductive verification and would be substantially higher for verification using an interactive theorem prover. Annotating and verifying the code took roughly 2.7 person years (not including the time required to extend Gobra); running Gobra on the implementation takes 46 minutes on a commodity desktop. We expect that our specifications will also facilitate the safe evolution of the code base; after annotating any modified parts, Gobra can check whether the changed implementation is still correct and secure.

Discovered implementation bugs. We reported fourteen previously unknown issues in the router implementation, all of which were confirmed and fixed by the SCION developers. The detected issues affect the router's safety, functional correctness, and the I/O behavior. We list a subset of the errors we found in Table 3. While verifying safety (referred to as S in Table 3), we found missing bounds checks, a `nil`-pointer dereference, and data races. Verification of functional correctness (F) uncovered missing checks after decoding SCION paths, incorrect error handling, and the incorrect accumulation of metrics. Finally, while verifying protocol compliance (P), we discovered a critical bug where the router did not reject packets simultaneously originating from and destined to the internal network. As a consequence, an attacker could craft an ill-formed packet that keeps being redirected to the internal network of an AS, potentially jeopardizing the availability of the router. This behavior is explicitly ruled out by the SCION protocol and our models, but was allowed by the implementation. These findings demonstrate that all steps of the code verification were effective in uncovering bugs. Remarkably, all of these bugs escaped the extensive code reviews, testing, and fuzzing that are continuously performed on the code base, in parallel with formal verification.

6 Lessons learned

In this section, we reflect on our project and draw lessons learned for future large-scale verification efforts.

6.1 Overall approach

The most important success factor for our project was to separate protocol and code verification, which worked extremely well.

First, this separation proved to be very valuable, since it allowed us to use the right tools for each task. Performing both verification tasks within one framework would have been much harder. For example, to simplify reasoning about path segments, we derived tailor-made induction schemes. We also built protocol models that are parametrized on cryptographic schemes and assumptions about them. These features are not commonly available in SMT-based program verifiers. However, performing the entire verification within an interactive proof assistant would have meant forgoing the high degree of automation offered by SMT-base program verifiers and incurred a significant increase of the verification effort.

Second, our project demonstrated that the linked protocol and code verification processes are *mutually* beneficial, as opposed to the protocol models just serving as a specification of the code. By soundly linking the models to the code, we detected not only discrepancies between the models and the specified security properties, but also instances where the models did not faithfully capture the informal protocol specification. For example, initially our models performed checks on packets upon exiting an AS whereas the implementation correctly conducts checks on entry. In another example, described in Sec. 5.5, a discrepancy between models and code turned out to be an error in the implementation instead.

Using two frameworks, Isabelle and Gobra in our case, requires a semantic-preserving translation between them. The existing Igloo methodology soundly links the two formalisms using I/O specifications. We extended it with an automated, sound extraction of I/O specifications in Isabelle/HOL, which made the extraction of the router's I/O specification effortless and reduced its further translation to Gobra to simple syntactic transformations (see Sec. 3.6). The generated I/O specification is relatively simple and can be reused for different router implementations.

Overall, our success suggests that our verification approach can be used in other verification projects that target protocols and their pre-existing implementations. It is flexible enough to accommodate any protocol modeling approach based on transition systems. This includes, for example, Tamarin protocol models [7], and many program verifiers based on separation logic for different programming languages, including C, Go, Java, Python, and Rust.

6.2 Continuous development

Like most software, the SCION protocol and code are constantly evolving. In such projects, one cannot defer verification until the software reaches a stable state. Instead, we started our verification effort already when SCION was just a research project and evolved our formalizations and proofs along with the protocol and code.

Verifying the protocol and code already *during* their development has substantial benefits. As described in Sec. 4.5, we uncovered vulnerabilities and made suggestions to strengthen the obtained security guarantees, which were implemented early. In contrast, updating already widely-deployed protocols is hard and costly. Similarly, detecting implementation bugs early reduces the risk of their exploitation by attackers, which would cause economic and reputation damage. This is especially critical for a secure next-generation Internet architecture that is seeking widespread acceptance.

However, the early start also led to significant challenges. We faced multiple protocol changes that affected our models. Evaluating their effects on the security properties would have required laborious modifications to all models in the refinement sequence for each proposed change. To adapt to changes quickly, building on prior work [30, 32], we used parametrization to prove the security of an entire class of data plane protocols. The formalization defines several parameters (such as a cryptographic check function) and conditions that are sufficient for proving security in the parameterized refinement development. Since the proof effort for each instance is minimal, we could easily adapt to protocol changes.

The development of the router implementation was driven by strict performance and schedule constraints, such that extracting it from a formal model or co-developing code and proofs was not possible. Consequently, we verified an existing implementation that was frequently changing, as developers added features, optimized performance, and fixed bugs. We annotated a clone of the SCION repository, to which we ported changes from the original repository weekly to keep both repositories in sync. Porting changes to our code base required adding and often adapting annotations. In this process, we benefitted greatly from using modular verification, which confines the adaptations to a local scope and avoids adapting or re-verifying the unaffected parts of the code base.

We showed that conducting quality assurance *during* development is feasible, but requires strong modularity to limit the impact of changes. The reward is a higher chance of fixing found problems.

6.3 Limits of automated code verification

It was initially unclear whether SMT-based, automated verification would scale to the SCION router's Go implementation. While we hoped automation would make verifying a large, complex implementation a manageable task, SMT-based verification has two known limitations.

First, automated verification gives users less control over the proof, with limited means for debugging when the tool cannot automatically prove a property. We were always able to find strategies to work around this limitation, typically by phrasing annotations in a way that is beneficial for the SMT solver. In some cases, this required a good understanding of the tool's inner workings.

Second, proof automation may be slow, hindering the workflow of writing and incrementally improving specifications. As explained in Sec. 5.4, verification performance was a central challenge, and required developing strategies and improving the tools.

We conclude that automated verification of real-world code bases in complex languages is feasible. We are confident that automated verification greatly reduced the effort of verifying the code base compared to a more manual approach. We believe this also applies to similar verification projects, although some interaction with tool developers for debugging or adding features will likely be required.

7 Related work

We discussed operating system and compiler verification in the introduction. We focus here on the verification of networking systems, security protocols, and other distributed systems.

Networking-related verification. As noted in the introduction, our work differs from verifying network configurations [9, 28, 33, 59], which only considers the model level, and network functions [38, 47, 64–66], which focuses on the fully automated verification of local properties of network component implementations. In contrast, we combine protocol model and code verification to establish both network-wide security properties and local properties of the router. Moreover, the complexity of the existing implementation mandates deductive verification, requiring substantially more human effort.

As mentioned previously, we build on an existing protocol verification of a simplified version of SCION [30, 32]. We adopted their refinement-based approach, but extended the models substantially. Chen et al. [15] model S-BGP and an early version of SCION in a Prolog-style declarative language for networking protocols, and verify route authenticity (control plane) and data path authenticity (data plane, for SCION only). The latter is weaker than path authorization, as it considers each hop separately instead of relating successive hops. Hence, it does not rule out path splicing attacks.

Arnaud et al. [5, 6] model routing protocols in a process calculus and propose two decision procedures for their analysis. One analyzes a protocol for any topology and the other works for a given topology. All of these works are limited to protocol design verification, whereas we also provide guarantees for the implementation.

Verified security protocols. The Everest project has verified several security protocols, including TLS [11, 18] and QUIC [19] at the code level. They implement the TLS and QUIC protocols in F# [11] and F* [18, 19], respectively, and use the associated refinement type checker (F7 and F*, respectively) for the cryptographic security proofs. As they use a cryptographic attacker model, their proofs yield stronger security guarantees than are possible in a Dolev-Yao model. The resulting protocol implementations, written in F# or in OCaml extracted from F*, are reference implementations, which were designed for verification and achieve a significantly lower performance than OpenSSL. The authors also provide more efficient implementations written in language fragments that can be compiled to C (e.g., the Low* fragment of F* [49]), which achieve a performance similar to an optimized implementation in some cases, but require an additional proof of equivalence with the reference implementation [18, 19]. In contrast, we verified a pre-existing, optimized protocol implementation and soundly connect it to a protocol model using the Igloo methodology.

Both DY* [10] and Arquint et al. [7] verify security protocols at the code level by proving invariants over the protocol traces and showing that these invariants entail the desired security properties. Our refinement-based approach increases modularity, which is crucial for verifying systems of the size and complexity of SCION. Arquint et al. [8] extend the Tamarin prover with the automated generation of each protocol role's I/O specification and verify the official Go implementation of the WireGuard key exchange protocol. As said earlier, Tamarin is not expressive enough for our purposes.

Distributed system verification. IronFleet [26], Verdi [60, 63], and Velisarios [50] verify respectively the Paxos, Raft, and PBFT consensus protocols. Chapar [37] proves causal consistency of key-value stores. In IronFleet, models and implementation are written in Dafny [35], which does not allow for concurrent code, and are verified with an SMT solver. The other approaches model algorithms in Coq and rely on code extraction from Coq, which is unsuitable for verifying existing implementations in commonly-used programming languages. Anvil [57] is a framework for building and verifying Kubernetes controllers. It restricts the structure of the implementation, making it unsuitable for verifying pre-existing code.

8 Conclusion

We verified the SCION router from its high-level design down to its performance-optimized implementation. A key success factor was our highly modular verification approach, which allowed us to reduce the verification complexity, work in parallel on different aspects of the problem, and confine the impact of protocol and implementation changes. We discovered both design and implementation errors, critically affecting the router's security, that evaded all prior reviews and testing and have all been subsequently fixed.

Future work includes a tighter integration of code development and verification, incorporating upcoming features of the SCION router, and verifying the control plane.

Acknowledgments

This work was partially supported by the Zurich Information Security and Privacy Center (ZISC), by the EU-funded NGI Pointer and NGI0 Core projects, and by the Institute of Information & Communications Technology Planning & Evaluation grant funded by the Korea government (MSIT) (No. RS-2024-00440780). We thank the SCION developers for their helpful feedback on the reported issues and the proposed fixes.

References

[1] [n. d.]. https://doi.org/10.5281/zenodo.16891069.
[2] Prometheus Authors 2014-2024. [n. d.]. Prometheus Go client library. https://github.com/prometheus/client_golang/.
[3] Martín Abadi and Véronique Cortier. 2006. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science* 367, 1 (2006), 2–32.
[4] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991).
[5] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. 2011. Deciding Security for Protocols with Recursive Tests. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6803)*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer, 49–63.
[6] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. 2014. Modeling and verifying ad hoc routing protocols. *Inf. Comput.* 238 (2014), 30–67.
[7] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. 2023. A Generic Methodology for the Modular Verification of Security Protocol Implementations. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Denmark, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM.
[8] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. 2023. Sound Verification of Security Protocols: From Design to Interoperable Implementations. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1077–1093.
[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. ACM, 155–168.
[10] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 523–542.
[11] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with Verified Cryptographic Security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 445–459.
[12] Bruno Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 2001, Canada*. IEEE Computer Society, 82–96.
[13] Bruno Blanchet. 2013. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 54–87.
[14] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72.
[15] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. 2015. A Program Logic for Verifying Secure Routing Protocols. *Logical Methods in Computer Science* 11, 4 (2015).
[16] Laurent Chuat, Markus Legner, David Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. 2022. *The Complete Guide to SCION*. Springer.
[17] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 2022. Sound Automation of Magic Wands. In *Computer Aided Verification - 34th International Conference, CAV 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 130–151. doi:10.1007/978-3-031-13188-2_7
[18] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *2017 IEEE Symposium on Security and Privacy, SP 2017, CA, USA, 2017*. IEEE Computer Society.
[19] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. 2021. A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer. In *42nd IEEE Symposium on Security and Privacy, SP 2021, 2021*. IEEE, 1162–1178.
[20] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Information Theory* 29, 2 (1983).
[21] Marco Eilers, Malte Schwerhoff, and Peter Müller. 2024. Verification Algorithms for Automated Separation Logic Verifiers. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 362–386.
[22] Santiago Escobar, Ralf Sasse, and José Meseguer. 2012. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming* 81, 7 (2012), 898–928.
[23] Giacomo Giuliari, Dominik Roos, Marc Wyss, Juan Angel García-Pardo, Markus Legner, and Adrian Perrig. 2021. Colibri: a cooperative lightweight inter-domain bandwidth-reservation infrastructure. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, 104–118.
[24] Google. [n. d.]. GoPacket. https://github.com/google/gopacket.
[25] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 653–669.
[26] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM.

[27] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, 637–650.

[28] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI 2013*. 15–27.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP)*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM.

[30] Tobias Klenze and Christoph Sprenger. 2022. IsaNet: Formalization of a Verification Framework for Secure Data Plane Protocols. *Archive of Formal Proofs* (June 2022). https://isa-afp.org/entries/IsaNet.html, Formal proof development.

[31] Tobias Klenze, Christoph Sprenger, and David Basin. 2021. Formal Verification of Secure Forwarding Protocols. In *2021 IEEE 34rd Computer Security Foundations Symposium (CSF)*. IEEE.

[32] Tobias Klenze, Christoph Sprenger, and David Basin. 2022. IsaNet: A framework for verifying secure data plane protocols. *Journal of Computer Security* (2022).

[33] Dexter Kozen. 2014. NetKAT – A Formal System for the Verification of Networks. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014*. 1–18.

[34] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192.

[35] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer.

[36] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[37] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM.

[38] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Hungary 2018*, Sergey Gorinsky and János Tapolcai (Eds.). ACM.

[39] Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995).

[40] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 696–701.

[41] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*. Springer, 405–425.

[42] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016. Proceedings*.

[43] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer.

[44] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

[45] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (Lecture Notes in Computer Science, Vol. 2142)*. Springer, 1–19.

[46] Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, 2015. Proceedings*, Jan Vitek (Ed.), Vol. 9032. Springer.

[47] Solal Pirelli, Akvile Valentukonyte, Katerina J. Argyraki, and George Candea. 2022. Automated Verification of Network Function Binaries. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association.

[48] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy, SP 2020, 2020*. IEEE.

[49] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *PACMPL* 1, ICFP (2017).

[50] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings*, Amal Ahmed (Ed.), Vol. 10801. Springer.

[51] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

[52] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. 2012. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, Stephen Chong (Ed.). IEEE Computer Society, 78–94.

[53] Malte Schwerhoff and Alexander J. Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier. In *29th European Conference on Object-Oriented Programming, ECOOP 2015 (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 614–638. doi:10.4230/LIPICS.ECOOP.2015.614

[54] SIX. [n. d.]. Secure Swiss Finance Network. https://www.six-group.com/en/products-services/banking-services/ssfn.html

[55] Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP (Lecture Notes in Computer Science, Vol. 5653)*. Springer, 148–172.

[56] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. 2020. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 152:1–152:31.

[57] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 649–666.

[58] Dominique Unruh. 2010. The impossibility of computationally sound XOR. *IACR Cryptology ePrint Archive* 2010 (12 2010), 389.

[59] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proc. ACM Program. Lang. (OOPSLA 2016)*. Association for Computing Machinery, 765–780.

[60] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*, David Grove and Steve Blackburn (Eds.). ACM.

[61] François Wirz, Marten Gartner, Jelte van Bommel, Elham Ehsani Moghadam, Grace H. Cimaszewski, Anxiao He, Yizhe Zhang, Henry Birge-Lee, Flix Kottmann, Cyrill Krähenbühl, Jonghoon Kwon, Kyveli Mavromati, Liang Wang, Daniel Bertolo, Marco Canini, Buseung Cho, Ronaldo A. Ferreira, Simon. P. Green, David Hausheer, Junbeom Hur, Xiaohua Jia, Heejo Lee, Prateek Mittal, Omo Oaiya, Chanjin Park, Adrian Perrig, Jerry Sobieski, Yixin Sun, Cong Wang, and Klaas Wierenga. 2025. Scaling SCIERA: A Journey Through the Deployment of a Next-Generation Network. In *Proceedings of ACM SIGCOMM*.

[62] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *CAV (1) (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 367–379.

[63] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, 2016*. ACM.

[64] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, 2019*. ACM, 275–290.

[65] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. ACM.

[66] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. 2020. Automated Verification of Customizable Middlebox Properties with Gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, 2020*. USENIX Association, 221–239.

[67] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn Heule, and Bryan Parno. 2024. Context Pruning for More Robust SMT-based Program Verification. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*.

[68] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, 2017*. ACM, 1789–1806.