Security protocols for satellite communications

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Cyber Security
June 2024
Bellot Adrien

Supervisors:
Dr. Mohammed B. M. Kamel (ELTE University - Hungary)
Alfonso Iacovazzi (RISE Research Institutes of Sweden - Sweden)
Examiners:
Prof. Seppo Virtanen (University of Turku)
Prof. Jouni Isoaho (University of Turku)

UNIVERSITY OF TURKU
Department of Computing

Bellot Adrien: Security protocols for satellite communications

Master of Science (Tech) Thesis, 60 p., 17 app. p.
Cyber Security
June 2024

The work explores the application of the MASK (Multi-AS-Key) protocol for enhancing satellite communication security. MASK is integrated with IPv6 segment routing (SRv6) to provide robust data transmission security with minimal computational overhead. The thesis includes a detailed implementation of MASK in a simulated environment using Mininet and IPMininet, demonstrating its effectiveness. Formal verification using the Tamarin prover confirms MASK's security properties, ensuring data integrity, source authentication, and replay attack resistance. Future work suggests implementing multipath support and full control plane integration to further enhance the protocol's capabilities.

Keywords: MASK, Source routing, Segment routing,Tamarin prover, Satellite communications

Acknowledgements

I would like to thank my university supervisors, Dr. Mohammed B. M. KAMEL and Dr. Seppo Virtanen for their support, patience and availability to help and guide me in this master thesis.

I would also like to thank Alfonso Iacovazzi for taking the time to organize weekly meetings to monitor my progress, for his continuous help, and all the resources he provided that allowed me to quickly and swiftly understand all the new concepts introduced by this thesis.

I would also like to express my gratitude to Simon Bouget for his advice and help on the Tamarin Prover software.

Finally, thank you to Dr. Shahid Raza, director of the cybersecurity unit at RISE, as well as Alfonso Iacovazzi again for entrusting me and giving me the opportunity to work on this internship.

Contents

Appendices

1 Introduction

1.1 Overview

The goal of this thesis is the creation and application of security protocols for satellite communication networks. I have chosen to concentrate on the MASK (Multi-AS-Key) protocol, which improves data transmission security with little overhead. With minimal computational load, MASK maintains good security through probabilistic marking and dynamic key management. I compare MASK with the EPIC (Every Packet Is Checked) protocol, emphasizing the flexibility and lower communication overhead of MASK. In the implementation part, I describe how to simulate a network using Mininet and IPMininet and show how to integrate MASK with segment routing over IPv6. I use the Tamarin Prover, a formal security proof tool, to confirm that MASK is resilient to several types of attacks and prove its reliability for satellite communications. I go on implementation outcomes and possible enhancements, like support for multi-path routing and control plane improvements, in the sections on the conclusion and future work.

1.2 Problem statement

Using the space segment to transfer information introduces more limitations than the already well-established ground-based network infrastructure. Satellites face unique challenges, including higher latency, dynamic topological changes, limited computational resources, and increased exposure to potential security threats. These challenges necessitate innovative mechanisms to enhance network security effectively. Existing protocols like EPIC and SCION offer exciting tools to improve cybersecurity within the network but require massive architectural changes and introduce significant overhead. This makes them less practical for satellite applications, where flexibility and efficiency are crucial. This thesis addresses the need for a lightweight, flexible, and efficient security protocol specifically designed for satellite networks. The goal of the proposed MASK (Multi-AS-Key) protocol is to offer strong security guarantees without necessitating significant changes to the infrastructure. This research is centred on the topic of whether such a protocol is practical, functional, and capable of providing enough security to be taken into consideration for implementation in satellite communication systems. This involves assessing how well the protocol can manage the particular limitations of satellite networks while upholding strict guidelines for data integrity, authentication, and attack resistance.

1.3 Objectives

The objectives of this thesis can be summarized as follows:

- Understand the principles underlying segment routing over IPv6 (SRv6) and MASK in detail.
- Develop a software implementation of the MASK protocol integrated with SRv6 in a controlled and simulated environment as a proof of concept.
- Model the MASK protocol using the Tamarin prover to formally verify its security properties.

1.4 Contributions

At the end of this thesis, I will demonstrate several significant achievements in the field of satellite communication security. I'll demonstrate how the MASK protocol has been successfully integrated with IPv6 segment routing. This integration demonstrates both protocols' interoperability in a simulated network environment while conforming to their requirements. The thesis will also provide security analysis and proofs utilising the Tamarin prover. Through the process of modelling the MASK protocol, I will analyse its security aspects. This analysis will show that despite the MASK protocol being susceptible to replay attacks at on-path routers, it does, in fact, provide source authentication as the protocol's creators intended. In addition, this study will highlight important issues and difficulties that must be resolved in future studies in order to fully utilise the MASK protocol's potential in satellite communications. To extend the scope of this work and overcome the limitations identified in the research, recommendations for further studies will be given. These efforts seek to improve our knowledge of and ability to implement security procedures in satellite networks, paving the way for more effective and safe space segment communication infrastructures.

2 Related work

In this section, we will go through the research that has been done in order to have the necessary basis and understanding of what has been done in path aware networking protocols and the security related to them. We will start by introducing a new path aware internet architecture called SCION, then have a look at a family of protocols that extends it called EPIC. Followed by the protocol SwiftParade, an anti-burst multipath validation. FABRID which stands for Flexible Attestation-Based Routing for Inter-Domain Networks and finally segment routing over IPv6 proof of transit (SRv6 POT)

2.1 SCION

The SCION stands for SCALABILITY, CONTROL, AND ISOLATION ON NEXT-GENERATION NETWORKS is a new generation network architecture developed by researchers at ETH Zurich and aims to to offer explicit trust information, route control, and failure isolation for end-to-end communication. In order to enable global connection, SCION articulates the current ASes into isolation domains that are clusters of separate routing planes that connect to one another. Isolation domains offer endpoints robust control over incoming and outgoing traffic, naturally isolate routing errors and misconfigurations, foster real and enforceable trust, and facilitate scalable routing changes with high path freshness. Strong resilience and security features are thus an inherent result of the SCION architecture's design. In addition to offering great security, SCION also offers scalable routing and very efficient packet forwarding. Because SCION has a path-based architecture, its end hosts discover available parts of the network path and merge them into end-to-end paths that are transmitted in packet headers. Senders, recipients, and ISPs can all choose their preferred path because path building is limited by the route restrictions of ISPs and receivers thanks to inbuilt cryptographic methods. Path-aware communication is a new networking trend that is made possible by this method. Additionally, these qualities make multi-path communication possible, which is a crucial strategy for robustness against DDoS attacks, higher end-to-end bandwidth, quick failover in the event of a network breakdown, and high availability. [1] [2]

2.2 EPIC protocols

While the EPIC (Every Packet Is Checked) protocols have not been chosen for this project (See section 3.5), it defines the fundamental concepts to use for this internship. These fundamentals are path validation, path authorization, packet and source authentication. These concepts are detailed and explained in Section 3.4. The EPIC protocols assure these properties in a Path Aware Network and is divided in 4 "levels", each of these levels are build on the previous ones and assure increasing level of security. The drawbacks is that the higher level you implement, the more computation and cryptographic operations you have to do on each on-path router as well as at the source and destination. EPIC is a family of cryptographic data-plane protocols designed to enhance the security of path-aware Internet architectures. The EPIC protocols are structured into four levels (L0 to L3), each offering progressively stronger security properties. Below is a detailed description of each level, their mechanisms, and the security guarantees they provide.

Level 0: Path Authorization EPIC Level 0 (L0) provides basic path authorization, ensuring that packets can only traverse paths authorized by all on-path autonomous systems (ASes). This level uses static Message Authentication Codes (MACs) calculated during the path exploration phase. Every AS determines a hop authenticator, $\sigma(0)$, as a MAC over the beacon timestamp, hop data, and the preceding hop authenticator, truncated to a certain length, during path exploration. There is no prior hop authenticator for the AS that is initiating the beacon; $\sigma(0)'$ is not present. In the data plane, this hop authenticator acts as the hop validation Field (HVF) directly. To construct a packet, the source host obtains a path and its associated hop authenticators from the path server. The source then includes the path timestamp, hop information, and the hop authenticator for each hop in the packet header. As the packet moves through the network, each AS recalculates the HVF and verifies it against the packet header. If the packet's interfaces and expiration checks are correct, it is forwarded; otherwise, it is dropped.

Level 1: Improved Path Authorization EPIC Level 1 (L1) builds on L0 by introducing per-packet HVFs to prevent reuse by attackers. The static hop authenticators of L0 can be exploited through brute-force attacks. In L1, these are replaced with dynamic HVFs calculated for each packet, ensuring they cannot be reused. During path exploration, each AS calculates its hop authenticator and segment identifier, which is a truncated version of the hop authenticator. For each packet, the source calculates the HVFs using the packet's timestamp and source identifier. The source constructs the packet by including the path timestamp, segment identifiers, and HVFs. Each AS, upon receiving a packet, recalculates the hop authenticator and segment identifier, verifies the HVF, and checks the packet's freshness using a replay-suppression system. This system uses Bloom filters to track and identify duplicates based on unique packet origins.The replay-suppression system also helps mitigate Denial of Service (DoS) attacks .

Level 2: Authentication EPIC Level 2 (L2) extends L1 by adding source authentication for routers and payload authentication for the destination. This level ensures that packets originate from valid sources and that the payload remains unaltered during transit. L2 defines host keys for each on-path AS and an additional key shared between the source and destination. HVFs in L2 include source authentication, combining the packet timestamp, source identifier, and hop authenticator. When the source constructs a packet, it fetches the necessary host keys, calculates the HVFs, and adds a destination validation field (VSD) for payload authentication. Each AS recalculates the host keys and verifies the HVFs, forwarding or dropping the packet based on these checks. The destination validates the VSD to ensure the payload's integrity.

Level 3: End-Host Path Validation EPIC Level 3 (L3) is the most advanced protocol in the EPIC family, enabling end-to-end path validation for both the source and destination. This level ensures that packets follow the intended path, enhancing security and compliance. In L3, on-path ASes overwrite their HVFs with proofs of packet processing, allowing the destination to validate the entire path upon receiving the packet. The destination sends a confirmation message back to the source containing the updated HVFs, enabling the source to verify the packet's path. The source initializes the packet by calculating HVFs that include proofs of processing by on-path ASes. Each AS updates the HVFs with proofs, and the destination validates the path and payload using the VSD, sending validation results back to the source.

Every EPIC tier preserves efficiency while improving security features. EPIC protocols help create a more secure and effective future Internet by addressing crucial security requirements such as path authorization, source authentication, payload integrity, and path validation. [3]

2.3 SwiftParade

SwiftParade is a solution that addresses the issue of bursty traffic and validates packet forwarding in multipath routing systems. The parallel transmissions that occur with multipath routing pose a challenge for traditional validation techniques. However, SwiftParade's composite validation technique enables the simultaneous validation of many packets, improving scalability and efficiency. It uses composite validation, which enables routers to validate packet groups as a whole. This method drastically lowers the computational load by dividing the validation overhead among several packets. In order to enable composite validation, the technique takes advantage of noncommutative homomorphic asymmetric encryption, assuring that proofs may be effectively verified regardless of the quantity of packets or complexity of the network. Routers create and update proofs for packet groups when it comes to proof construction and verification. The paths of the packets remain valid and intact thanks to these certifications. Composite proofs are checked during verification to make sure packets have taken legitimate paths without being tampered with. Using composite numbers in its encryption system, SwiftParade is built with strong security measures to ward off brute-force, selective-forging, and quadratic-residue attacks. In terms of performance, SwiftParade shows high efficiency in the creation and verification of proofs. It offers significant gains over conventional approaches and is well-suited for complex multipath routing scenarios due to its constant proof size of 406 bytes and constructs proofs in about 9.13 microseconds per packet, regardless of network complexity. [4]

2.4 FABRID

Flexible Attestation-Based Routing for Inter-Domain Networks, or FABRID, is a network routing system that aims to improve inter-domain networks' forwarding channels' security, control, and transparency. The protocol is built on the SCION (Scalability, Control, and Isolation on Next-Generation Networks) architecture. Path Construction Beacons (PCBs) are used by ASes in the control plane to propagate path policies. For internal paths that adhere to these policies, these beacons carry AS-local IDs, or policy indices, as well as details regarding router attributes. By using these indexes, the system can more effectively manage space and cut down on redundancy by accessing comprehensive policy descriptions kept in local or global repositories. To choose the best paths, endpoints retrieve these path policies from the local control service and compare them with their local preferences. By doing this, it is ensured that the required router attributes are met without divulging private network information. Endpoints encode specific policies into the headers of their data packets on the data plane. These encoded policies are used by on-path ASes to route packets via internal paths that correspond to the designated policies. To defend against manipulation and maintain privacy, the policies are encrypted and validated. ASes' border routers verify the policy indices to make sure packets are routed in accordance with the endpoints' preferred policies. By preserving a certain degree of path control, the system offers advantages even in the event that some ASes on the forwarding path do not support FABRID. FABRID offers a number of security mechanisms. It uses EPIC (Every Packet Is Checked) and SCION's public-key infrastructure (CP-PKI) to validate packet forwarding and authenticate routing data. By doing this, the policy indices that are encoded in the packet headers are guaranteed to be authentic and confidential, protected from manipulation and unauthorised access. Furthermore, by utilising Trusted Platform Modules (TPMs) for remote attestation on their routers, ASes can increase the credibility of the policies they advertise. This further secures the data forwarding path by confirming that the routers are operating legitimate, uncompromised software. [5]

2.5 SRv6 POT

An approach to cryptography called the SRv6 Proof of Transit (POT) was created to verify that packets in a network have followed the precise path that the Segment Routing over IPv6 (SRv6) protocol has defined. The efficiency, security, and compliance of traffic engineering and service function chaining in a network are all dependent on this technique. The process starts at the ingress node, where a keyed Hashed Message Authentication Code (HMAC) is used to construct a Path Verification Factor (PVF). By confirming that the Segment Routing Header (SRH) was created by a reliable source, this PVF guarantees the SRH's integrity. A random Nonce is supplied to ensure that each packet's evidence is distinct and prevent replay attacks. Using keys that are shared with every segment in the packet's path, the ingress node encrypts the PVF. The sequence in which this encryption is carried out needs to be reversed when the packet moves across the network. Using its shared key, each intermediate segment decrypts the PVF and updates the packet to reflect its journey through the segment. This phase confirms that each designated segment has been traversed by the packet. The initial HMAC can be seen at the egress node when the PVF has been fully decrypted. Next, to make sure the packet arrived in the correct order of segments, the egress node verifies this HMAC. If the packet did not pass through all the specified segments or if the order was incorrect, the HMAC validation would fail, indicating a deviation from the intended path. [6]

3 Background

This chapter is designated to introduce all the concepts needed to know before properly talking about the implementation of the MASK protocol. It will start with by a brief overview on how to predict the satellite link availability of such peculiar network that is the space segment, followed by an introduction to segment routing over IPv6 (SRv6), defining what a path aware network is, the different security properties we would like to have in it, the reasons for choosing MASK over EPIC and finally a detailed description of the workflow of the MASK protocol.

3.1 Predicting satellite link availability

One might think this implicates a lot of complications compared to a regular network. In reality, yes but also no. This project considers the satellite network but does not exclude the network on the ground. If the path selected by the source is a ground path for a multitude of reasons (latency, numbers of hops, user preferences, etc.); it will take this path. If the path is purely in the satellite network or a mix of both, same scenarios. The biggest question mark when it comes to the space segment is its availability. What appears to be a tricky question does have a simple answer. Satellites are tracked and monitored by ground stations that receive their position in space. Even in the case of the satellite being out of range of every ground station, we still know its velocity and its orbital trajectory. It is then easy to predict where the satellite will be in the near and distant future as well as how long it will stay available for data transmission and which other satellites are in range of it for relaying that data. Assuming this, we can extrapolate that the ground stations will act as a core actor of the control plane of the PAN and disseminate to the network the paths available for data to transit with the time frame of the availability of these paths. [7] [8] Of course, predicting the availability of those links does not prevent us from countering link failures in case of a communication problem or even a satellite being unable to transmit further information if it does not have the necessary energy left to do so and needs to recharge its batteries. Link prediction is one crucial step but further parameters must be taken into account.

3.2 Segment routing Over IPv6

Segment Routing Over IPv6 (SRv6) is being used in this thesis for the implementation of the MASK Protocols. The reason to use SRv6 with the MASK implementation is because, aside from already being supported natively on multiple routers, the path that the packet will take is a header in it thus reinforcing transparency as well as source routing in Path Awareness Networking. Now that the reason is stated, what is segment routing (SR)?

Segment routing (SR) is an elaborate network routing technique that indicates the exact path packets take via a network by applying the source routing paradigm. Segment routing permits the packet sender to specify the complete path, in contrast to conventional routing techniques where each router along the route determines the next hop. Segment identifiers (SIDs) and segments are concepts used in segment routing. A segment is basically a routing order, like processing a packet by a specified service or forwarding it through a particular interface. These sections may be service-based, associated with particular network services, or topological, associated with the architecture of the network. A segment identifier (SID), which refers to various routing instructions or services, identifies each segment. Sets of ordered segments called SR policies define the paths that packets take. Usually, these restrictions are implemented at the point of entrance into the SR domain, known as the ingress node. A packet processes each segment sequentially as it moves across the network according to the route specified by the SR Policy. A node processes the active segment that is currently in progress when a packet reaches it, and updates the segment list to point to the next segment.It supports distributed centralized or hybrid control planes as well as two data plane implementations: SR-MPLS and SRv6. [9] Segment routing over IPv6 simply applies the concept of segment routing to the IPv6 dataplane.

The segment routing header (SRH) is defined as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Next Header   |  Hdr Ext Len  | Routing Type  | Segments Left |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Last Entry   |     Flags     |              Tag              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|            Segment List[0] (128 bits IPv6 address)            |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
                              ...
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
|                                                               |
|            Segment List[n] (128 bits IPv6 address)            |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                                                             //
//         Optional Type Length Value objects (variable)       //
//                                                             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

At every hop, the router will check the segments left, last entry and the segment list[n]. Each Segments List[n] is an IPv6 address on the path. segments left is simply the number of IP addresses (hops) left until the destination. The Last Entry is the index of the list in which we are supposed to be (i.e. Segment List[1]). At each hop, the node associated with the packet's destination's address (DA) is the only one permitted to view the Routing Extension Header and, by extension, the SRH. Any other transit node that receives the packet must forward it towards the DA in accordance with its IPv6 routing table and must not examine the routing header beneath. If the node has the specified address, then it starts the process to ensure proper handling and forwarding of the packet. The node first inspects the Segments Left field. If this field reads zero, the router moves on to process the next header in the packet, guided by the Next Header field in the routing header.

If the Segments Left field is not zero, the router knows additional actions are required. If the local configuration requires type-length-value (TLV) processing, the router performs this task. Next, it calculates the maximum valid last entry using the formula:

$$\text{max\_last\_entry} = \left(\frac{\text{Hdr Ext Len}}{2}\right) - 1$$

If the Last Entry exceeds this maximum or if the Segments Left value is greater than Last Entry plus one, the router sends an ICMP Parameter Problem, Code 0 message to the Source Address, pointing to the Segments Left field, and discards the packet.

In the absence of these issues, the router decrements the Segments Left field by one. It then updates the destination address in the IPv6 header with the value from the Segment List at the new Segments Left index. At this point, the router verifies the IPv6 hop limit. If this limit is less than or equal to one, the router sends an ICMP Time Exceeded message, indicating Hop Limit Exceeded in Transit, to the source address and discards the packet. If the hop limit is greater than one, the router decrements it by one and resubmits the packet to the IPv6 module, ready to be transmitted to its newly assigned destination. [10]

Another field in this header that is worth looking deeper into is the "Optional Type Length Value objects" or TLV in short. This TLV will be used for the implementation of the MASK protocol. We will use the HMAC TLV because it contains a HMAC (Keyed-Hashing for Message Authentication) and will be used as a cryptographic value necessary in the computation of other cryptographic values in MASK.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Type     |     Length    |D|        RESERVED             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      HMAC Key ID (4 octets)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                              //
|                      HMAC (variable)                         //
|                                                              //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The HMAC TLV is used to make sure the segment list remains unchanged after it's generated and to confirm that the SRH applied to a packet was chosen by an authorised party. This makes it possible to confirm that the present segment is permitted for use because it is listed in the approved segment list. The SR domain uses ingress filtering techniques as outlined in BCP 84 [RFC3704] or other tactics as necessary to make sure the source node is allowed to utilise the source address in the packet.[10] In MASK, the HMAC will be used as a parameter to calculate the $mark_i$ but more on that later.

3.3 Path awareness networking (PAN)

A path-aware internet architecture differentiates itself from traditional internet protocols by having two characteristics: it allows endpoints to access the attributes of accessible Internet paths as well as permitting applications and endpoints to choose routes for their traffic across the Internet based on these attributes. Although the property of "path awareness" is already present in many Internet-connected networks within single domains and via administrative interfaces to the network layer, a true path-aware inter network extends these principles across layers and the Internet. [11]

PAN systems can identify and utilize multiple paths between a source and destination. This is in contrast to traditional single-path routing protocols like OSPF (Open Shortest Path First) or BGP (Border Gateway Protocol). It allows dynamic selection of paths based on various criteria such as latency, bandwidth, congestion, and reliability. This adaptability helps optimizing performances and recover quicker when a route is down or becomes congested. Using such a network is making it more difficult for attackers to intercept or tamper with the data. This approach can also mitigate the risk of data breaches on a single compromised path. PAN is ideal for segment routing. For example, Google developed the QUIC protocol (Quick UDP

Internet Connections) that includes path-aware features to improve performances, reliability, and even policies between different autonomous systems.

For this project, it was critical to use a path aware network environment to allow packet route traceability. It ensures that packets are travelling by trusted hosts only in space and on the ground. If it appears that an on-path router has been compromised. We can quickly see what packets went through it and what data has been potentially compromised.

3.4 Security properties

For this project we need protocols that validate specific security properties. This section will be used to define properly each one of them.

- Path validation: The process of confirming that the paths being utilised for data transfer are trustworthy and legitimate. This means making sure the paths meet specific requirements and are devoid of compromised or malicious segments. This is done using multiple cryptographic techniques and protocols like hashes and digital signatures to check the path's validity.[12]
- Path authorisation: It consists of allowing selected network paths to be used for data transmission. It helps to maintain network policies and security by making sure that only authorised entities can choose and use specific paths. In practice it is achieved by defining rules that restrict path usage according to parameters such as user roles, data sensitivity, or network settings. Path permissions can be managed and enforced through the use of certificates, tokens, and access control lists (ACLs).[13]
- Packet authentication: Is the process of confirming each data packet's authenticity and integrity as it passes through the network. By using this procedure, it is certain that the packets were not changed or tampered with while

being transmitted. It can be verified by using cryptographic checksums or hash functions. The authentication can for example be achieved by using HMAC (Hash-based Message Authentication Code) or authentication headers in IPsec.

- Source authentication: Is the guarantee that a packet or data stream is authentic and that its authentic origin may be identified. This protects against impersonation and insures that data is coming from reliable sources. This is defined in reality by verifying the identity of the sender using credentials, such as digital certificates, to verify the sender's identity and making sure the message's sender is unable to retract its transmission. Examples of this are protocols such as TLS, IPsec, or digital signatures that verify the authenticity of the source.[14]

3.5 Why choosing MASK over EPIC?

While SCION and the EPIC protocols present some robust characteristics and are undeniably innovative, the reason for choosing MASK over EPIC is simple: Choosing EPIC meant that an extensive redesign of the network architecture was necessary in order to make it work. SCION does indeed provide features to make this transition as smooth as possible by allowing SCION packets to be embedded into legacy IP packets to allow an autonomous system running with SCION to communicate with legacy internet architecture. However, it was necessary to fully implement SCION on the desired autonomous system and doing so to communicate with legacy internet meant that some SCION features could not be used properly such as the proper propagation and use of the path-segment construction beacons (PCBs) that allow for source routing and multipath supports.

That is why a more flexible sets of protocols have been chosen such as the MASK protocols. It is also easier to implement in an already existing architecture as the authors explore the possibility of implementing the MASK protocols with segment routing over IPv6 (SRv6) and using the HMAC supported by it as a cryptographic value that will be used in MASK. Since IPv6 and SRv6 are nowadays natively supported on Linux systems, it is easier, cheaper and less tedious to incorporate MASK into any network infrastructure provided it supports IPv6 and SRv6. The authors of the paper also claim that MASK is faster, lighter and performs better on long paths compared to the EPIC protocols. The reason for this is that MASK uses a probabilistic marking mechanism to keep the additional packet header size short and constant, reducing communication overhead while EPIC grows larger the more hops it goes through. The authors of the EPIC paper mention that it is not a problem and will likely not happen since the average hops taken by a packet according to statistics is less than four [3]. However, since we are exploring the possibilities of using the satellite network for sending packets, we can't predict if this affirmation is still true for the space segment.For typical network paths (with realistic path lengths), the overhead introduced by MASK is between 1/2 to 1/10 of the overhead introduced by EPIC.[15] This means that if EPIC introduces, for example, 100 bytes of overhead for a given path length, MASK would introduce only 10 to 50 bytes of overhead for the same path length. MASK also has a mechanism to detect replay attacks. So having a protocol that, no matter the path length, can keep a constant header size is an important consideration in this project.

A drawback of using MASK instead of EPIC is that the MASK protocols do not support path authorization, but the authors of the paper claim it could be achieved by implementing MASK in SCION and using some of the EPIC cryptographic properties and the SCION control plane to fully implement it. MASK also has weaker source authentication since it uses a probabilistic way of doing it, which is weaker compared to EPIC's deterministic approach.

3.6 MASK protocols

3.6.1 What is MASK

Multi-AS-Key (MASK) is presenting itself as a source and path verification technique that has minimal overhead in the router compared to existing mechanisms and a streamlined additional packet header. Two ideas served as the foundation for MASK design: One is the finding that, in order to demonstrate that it has processed a packet, a router needs to compute at least one *mark*. The other is that since the mark is temporary (lasting for thousands of packets), MAC operations with robust security guarantees (like AES) are not required on the data plane.

The security and performance of the network are enhanced through a probabilistic marking mechanism that maintains a short, constant additional packet header while ensuring robust security. This approach supports the detection of packet drops and replay attacks at the destination, providing a higher level of network integrity.

MASK allow user to instruct stateless routers on processing packets according to user-specific policies, meeting dynamic security requirements. Its design ensures compatibility with existing network infrastructures, facilitating easy deployment without necessitating significant changes to current systems.

3.6.2 How MASK works

Source host processing

The source host starts by requesting dynamic keys from the local Key Distribution Server (KDS). These keys are crucial for generating cryptographic marks that are used throughout the packet's journey. There are two types of keys: the keys shared between the source and destination AS (AK_{sd}) and the keys shared between the AS of the source and the router i (AK_{si}). These keys will be used to derive another set of keys: The host dynamic keys (HDK) that will be given to the source and the destination and the router dynamic keys (RDK) that will be given to the source and the corresponding on-path router (each router has a different RDK that is shared with the source, meaning the routers on path only know their RDKs while the source knows them all). To be fully generated, HDK needs the AK_{sd} as mentioned above, but also a session. A session is defined like this:

$$\text{session} = H(\text{IP}_{\text{src}}||\text{IP}_{\text{dst}}||\text{id}_1||\cdots||\text{id}_n||\text{TS})$$

where IP_{src} and IP_{dst} are the source and destination IPs, $id_1...id_n$ are the on path router ids and TS is a timestamp corresponding to the beginning of the session. The HDK is then the Message Authentication Code (MAC) of the session, with AK_{sd} as keys

$$\text{HDK } = \text{MAC}_{\text{AK}_{\text{sd}}}(\text{session})$$

For the RDK, it is a bit different and is derived by concatenating the id_i of the router i and an indicator which is a 32-bit value composed of a 16-bit time slot ts and a 16-bit user identifier uid. The RDK is then the MAC with the AK_{si} of the previous value:

$$\text{RDK}_{\text{si}} = \text{MAC}_{\text{AK}_{\text{si}}}(\text{id}_i||\text{indicator})$$

Once the keys have been negotiated, the source can start constructing the packet. We have seen the SRv6 header earlier that will be in the packet as well in section 3.2, but now we must look at what the MASK header looks like.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     flag      |     epoch     |              seq              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           indicator                           |
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              id                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            marksrc                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             markr                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            markpkt                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Where the flag is an 8-bit value indicating specific packet features, epoch and seq fields are both 16-bit sized and are respectively the epoch number and the sequence number of the packet, an 32-bit indicator field that is composed of a 16-bit time slot and 16-bit user identifier information to select the appropriate RDK, and an id field representing the on-path router selected for verification of the $mark_{src}$.

Additionally, the MASK header includes three critical marks: $mark_{src}$, calculated by the source and verified by the selected on path router with the same id as the one embedded in the header; $mark_r$ which is updated by each on-path router with a XOR operation and finally, the $mark_{pkt}$, calculated by the source and verified by the destination.

Now that the keys have been negotiated and the header defined, we can go deeper into the construction of the packet. In PAN and source routing, the source gets to choose the preferred path and get the following information:

$$PATH = (id_1, id_2, ..., id_n)$$

$$RDKpath = (RDK_{s1}, RDK_{s2}, ..., RDK_{sn})$$

Then, it chooses the on path router who will have to perform the verification of the $mark_{src}$, each on-path router has 1/n chances to verify it (n is the path length). When considering the number of packets in a specific epoch, the source can designate the default number (e.g., 4096) and then use the HDK and the epoch to calculate the adjustment factor:

$$\lambda = \text{MACHDK(epoch)} \mod K$$

Where K is an integer, if we assume K is 4096, the value range of λ is from 0 to 4095. Therefore, an epoch will contain an amount of packets that ranges from 4096 to 8192. The number of packets in different epochs forms the Checkpoint Sequence Hopping (CSH). The CSH[M] is an array that includes M sequences for M epochs, e.g., $\text{CSH}[M] = \{4500, 5001, \ldots, \text{CSH}[m], \ldots\}$, meaning there are 4500 packets in epoch 0, 5001 packets in epoch 1, and $\text{CSH}[m]$ packets in epoch m.

It's worth noting that there is an 8-bit epoch in the packet header, but any epoch length can be used by the source and destination. Regarding the aforementioned destination, it can also calculate λ using the equation above.The source and the destination share the same CSH with this design and do not require inter-domain negotiation.

The source will then calculate the CST_{pkt} which is a concatenation of the source IP address, the path, flag, epoch, seq and id. This value is used to calculate the marks.

$$\text{CST}_{pkt} = (\text{IP}_{src}||\text{PATH}||\text{flag}||\text{epoch}||\text{seq}||\text{id})$$

The first mark to be calculated for the header is the $mark_{src_i}$ which the truncation of the 32 first bits of $mark_i$.

$$\text{mark}_i = \text{MAC}_{\text{RDK}_{\text{si}}}(\text{CSTpkt}||\text{id}_{i-1}||\text{id}_i||\text{AUTH})$$

$$\text{mark}_{src_i} = \text{mark}_i[0:32]$$

Where i is the value of the on-path router that will verify the $mark_{src}$. and $AUTH$ is the SRv6 HMAC.

Then the source is proceeding to calculate the next $mark$, which is the $mark_{pkt}$. In order to do so, the source will use the HDK, the CST_{pkt} previously computed and will generate a hash. This hash and the CST_{pkt} will be concatenated together and will be used with the HDK to generate a MAC. This MAC will then be truncated to the 32 lowest bit which is the $mark_{pkt}$.

$$\mathrm{mark}_{pkt} = \mathrm{MAC}_{\mathrm{HDK}}(\mathrm{CST}_{pkt}||\mathrm{hash})[0:32]$$

Finally, the source is calculating the next $mark$: the $mark_r$. This $mark_r$ is a XOR operation with the highest 32-bit string of each $mark_i$ and the hash used to generate the $mark_{pkt}$. Each on-path router will perform the XOR operation hop by hop and the $mark_r$ received at the destination, if all routers have performed as expected will be $\mathrm{mark}_r = \mathrm{hash}$. Thus,

$$\mathrm{mark}_r = \mathrm{hash} \oplus \mathrm{mark}_1[32:64] \oplus \ldots \oplus \mathrm{mark}_n[32:64]$$

The goal of this mark is to prevent adversaries from evading verification.

Router processing

As the packet traverses the network, each router it encounters processes it according to the MASK protocol. Upon receiving a packet, the router first checks that it entered the correct interface by checking expected path information (idi-1), filtering out any incorrect paths.

After this first check, the router then proceeds to calculate the $mark_i$ using its router dynamic key shared between the source and this router i (RDK_{si}).

In order to do so, it needs to calculate the CST_{pkt}. The process of getting this value as well as the computation of the $mark_i$ have already been explained above.

From the $mark_i$ calculated locally, if the router id is equal to the id present in the packet header, the router derives the $mark_{src_i}$ and compares it to the $mark_{src}$ present in the packet header. If they match, the packet is processed further and validate source authentication, dropped otherwise.

If the $mark_{src}$ has not been dropped or if the router id is not equal to the one present in the MASK header, the router will update the $mark_r$ by truncating the $mark_i$ previously calculated to the 32^{th} bit to the 64^{th} bit and perform the following operation:

$$\text{markr}_i = \text{markr}_{i-1} \oplus \text{mark}_i[32:64]$$

Once it is done, the router will forward the packet to the next hop where this whole process can be repeated until the packet is dropped or arrives at the destination. Overall, the whole processing of the packet by each on-path router can be summarized by this flowchart:

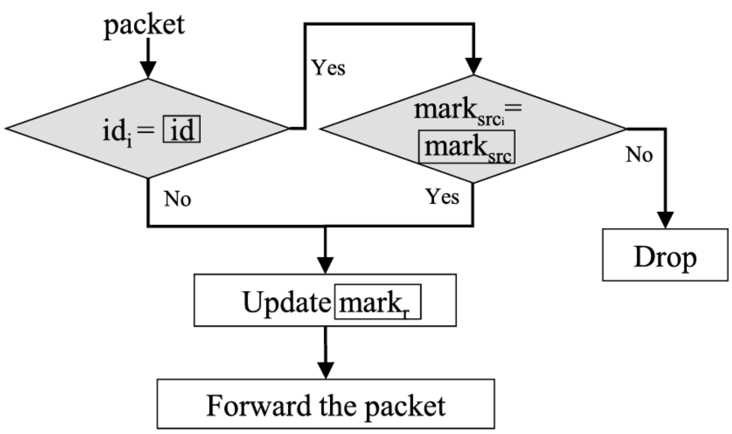

Figure 3.1: Packet processing on path router

Destination processing

When a packet reaches the destination host, it undergoes a series of verification steps to ensure both source and path integrity. The process starts with the destination initializing counters to record the number of packets that pass verification for each epoch and each on-path router. These counters are organized in a two-dimensional array, with $sCounter[m][i]$ representing the counter for on-path router i in epoch m.

Upon receiving a packet, the destination calculates the expected $mark_{pkt}$ using the host dynamic key (HDK) and the constant packet header CST_{pkt} concatenated with $mark_r$:

$$mark_{pkt} = \mathrm{MAC}_{HDK}(CST_{pkt} \| mark_r)[0:32]$$

The calculated $mark'_{pkt}$ is then compared with the $mark_{pkt}$ included in the packet. If they match, it indicates that the packet is valid, has not been tampered with and has been processed by the routers it was supposed to go through. Consequently, the counter $sCounter[m][i]$ for the corresponding router and epoch is incremented. If the marks do not match, the packet is filtered out and dropped.

As packets continue to arrive, the destination keeps verifying them until a predefined number (e.g., 200 packets) is reached for the current epoch. At the end of each epoch, the destination assesses the counters to ensure they align with the expected policy. The policy's compliance is checked using the formula:

$$N \times (1 - \beta_1) < sCounter[m][i] < N \times (1 + \beta_2)$$

where N is the expected number of packets, and β_1 and β_2 are thresholds to allow for normal packet loss and minor discrepancies. If any counter $sCounter[m-1][i]$ for the previous epoch $m-1$ violates this policy, it indicates a potential issue with the router i.

After validating the epoch, the destination could send an acknowledgment (ACK) packet back to the source. This ACK packet would include the verification results, which would be encrypted with the HDK to ensure secure communication. The destination can set $mark_{pkt}$ with $mark_{pkt} + 1$ to indicate it is an acknowledgment, preventing adversaries from dropping the ACK.

To further prevent replay attacks, MASK utilizes unique sequence numbers in each packet. The destination compares the received packets with the threshold and reports any anomalies, such as replay attacks, to the source. This mechanism ensures that even if an attacker tries to resend a legitimate packet, it will be detected and filtered out.

Since we know how many packets are within an epoch, this can be used by the destination host to ask for retransmission of the packet in case of a packet loss. A very important feature to consider since we established earlier that the space segment, while predictable, still has restrictions when it comes to transferring information efficiently. This retransmission possibility, combined with an implementation of multipath support could prove very efficient in countering packet loss and links failure in the space segment.

3.6.3 Security features and attack resilience

The MASK protocol integrates a comprehensive suite of security features designed to protect network communications from various threats. One of the primary security mechanisms it employs is source authentication. Each packet includes a source mark ($mark_{src}$) calculated by the source host using a dynamic key. This mark is verified by on-path routers and the destination to ensure that the packet originates from a legitimate source, effectively preventing spoofing attacks where an attacker could masquerade as a trusted source.

Path verification is another critical feature of MASK. The protocol uses a probabilistic marking scheme, where each on-path router updates a mark ($mark_r$) that is later verified by the destination. This mechanism ensures that packets follow the intended path specified by the source, protecting against path alteration attacks that might reroute packets through unintended paths, such as detours or reorderings.

Ensuring packet integrity is also a key focus of MASK. The destination recalculates a packet mark ($mark_{pkt}$) using its Host Dynamic Key (HDK) and compares it with the received $mark_{pkt}$. This comparison ensures that the packet's content has not been tampered with during transit, safeguarding against man-in-the-middle attacks that attempt to alter the packet's payload.

MASK also includes mechanisms to counter replay attacks. Each packet features a unique sequence number, which the destination tracks and verifies. By checking these sequence numbers, the destination can detect and discard duplicate packets, preventing attackers from disrupting communication by resending legitimate packets.

The protocol addresses packet drop attacks by using counters at the destination to track the number of verified packets for each router and epoch. These counters are compared against expected values based on a shared policy, helping to detect when packets are dropped by malicious routers and ensuring that packets are not silently discarded.

A distinctive feature of MASK is its use of probabilistic marking. Instead of performing full cryptographic operations on every packet, routers probabilistically update marks, reducing the computational burden. This approach helps prevent computational overload attacks that aim to exhaust router resources.

Dynamic key management is another integral aspect of MASK's security framework. Dynamic keys are derived for each session and periodically refreshed, preventing long-term key usage. This regular update ensures that even if an attacker gains access to a key, it quickly becomes obsolete, limiting potential damage from key compromise attacks.

Moreover, MASK is designed to maintain low communication and computation overhead. Its minimal additional packet headers and efficient cryptographic operations help sustain high performance, protecting against Denial of Service (DoS) attacks aimed at overwhelming network resources with excessive processing demands.

Additionally, MASK can detect and prevent packet damage, path changes, and colluding attacks by ensuring that each packet follows the specified path and is verified at multiple points. This comprehensive approach prevents tampering with the packet's source address or payload, ensures packets are not rerouted through unintended paths by dishonest routers, and detects when multiple routers collude to evade verification mechanisms.

However, while MASK provides robust protection against a range of attacks, it is not specifically designed to counter large-scale Distributed Denial of Service (DDoS) attacks, which typically require additional network-level strategies such as rate limiting and traffic analysis. Nevertheless, MASK's low overhead and efficient processing help mitigate some impact by not overwhelming the network infrastructure.

4 Implementation of MASK

This chapter will describe in detail the implementation of the MASK protocol as well as the tools used in order to achieve it.

4.1 Mininet & IPMininet

The network simulation is done with Mininet, an open source software that emulates "realistic virtual network, running real kernel, switch and application code" (https://mininet.org/). Process-based virtualization and network namespace features present in most recent Linux kernels are used by Mininet to build virtual networks. A Mininet "Host" can include any code that would typically run on a Linux server, such as a web server or client program. This is because Mininet hosts are emulated as bash processes operating in a network namespace. The Mininet "Host" will only be able to view its own processes and will have a private network interface. Software-based switches, such as Open vSwitch or the OpenFlow reference switch, are used in Mininet. Virtual Ethernet pairs, or links, are embedded in the Linux kernel and serve as a means of connecting our simulated switches to simulated hosts, or processes.

IPMininet is an open source extension of Mininet that allows Mininet to support more features and emulate complex IP networks. These features include but are not limited to: BGP, OSPF, SSHd and, more relevant to this project, ip6tables and Segment Routing over IPv6. [16]

4.2 Topology

Mininet combined with IPMininet allows us to define, configure and run complex topologies very quickly and very easily. Our network topology, since it is a proof of concept one, will be composed of 2 routers and 2 end hosts and will be arranged like the following:

```
++++++++++++++++++++++++++++++++
|H1|———|R1|———|R2|———|H2|
++++++++++++++++++++++++++++++++
```

This topology is simple but is enough for a proof of concept, it can be easily extended for further testing and implementation. The IP addresses of this topology are IPv6 addresses and static because the control plane is not implemented. MASK, Path Aware Networks and SRv6 rely on source routing meaning that the source must know the different paths from point A to B and choose one of them before sending any packets. Usually, an external trusted authority should negotiate, discover and provide these routes as well as most of the information related to this route to the source but since the control plane is not implemented, the source must already know the routes to take in order to send a packet. The IPv6 addresses of the hosts and routers present in the network are the following:

- H1: "fc00:a::1/64"
- R1: "fc00:a::2/64" and "fc00:c::1/64"
- R2: "fc00:c::2/64" and "fc00:b::3/64"
- H2: "fc00:b::4/64"

The routers R1, R2 and the host H2 have ip6tables activated with the following rules:

```
ip6_rules_r1 = [Rule('-I FORWARD -s fc00:a::1 -j NFQUEUE --queue-
    num 0 ')]
r1.addDaemon(IP6Tables, rules=ip6_rules_r1)

ip6_rules_r2 = [Rule('-I FORWARD -s fc00:a::1 -j NFQUEUE --queue-
    num 1 ')]
r2.addDaemon(IP6Tables, rules=ip6_rules_r2)

ip6_rules_h2 = [Rule('-I INPUT -s fc00:a::1 -j NFQUEUE --queue-num
    2 ')]
h2.addDaemon(IP6Tables, rules=ip6_rules_h2)
```

These rules allow incoming packets from H1 to be placed in a queue to be processed by the routers and the destination. More details about the mentioned queue and the processing of the packet in 4.4.1. The full code that defines the topology can be found here B.5.

4.3 Limitations of the implementation

This implementation is a proof of concept and has been carried out at the application layer of the ISO model using Python. Thus, only the data plane has been implemented as implementing the control plane is out of the scope of this thesis and would require a lot more time as well as manpower. As a result, some of the data needed from the control plane (i.e. the HDK) is present in the code. To provide a more concrete example, the source needs the RDK of the on-path routers in order to calculate markr

4.4 Implementation

The implementation will be divided into multiple subsections as the code has been divided into four sections:
The "MASK header" section containing the implementation of the MASK header. The "source" section where the packet is initialized and sent. The "on path router" section describing how the packet is being processed by on path routers and finally, the "end host" section containing the implementation of the packet processing once it reached its destination.

4.4.1 The MASK header

Before crafting a packet, a custom MASK header must be defined because it is not supported officially by any kernel or OS. To implement implement it and forge a packet with it later, we use "scapy", a python library designed to manipulate, code and decode packets following a wide variety of protocols. It also allows us to create our own protocols and packets. We define the header as it is described in the MASK paper. [15]

MASK Header:

```
class maskh(Packet):
name = "maskh"
fields_desc = [
    BitField("flag", 0, 8),
    BitField("epoch", 0, 8),
    BitField("seq", 0, 16),
    BitField("id", 0, 32),
    BitField("marksrc", 0, 32),
    BitField("markr", 0, 32),
    BitField("markpkt", 0, 32)
]
```

Where flag is the flag field, epoch and seq are respectively the epoch and sequence number, the indicator field is here omitted since the data plane is not implemented and the RDKs are known by the routers based on the id of the router present in the packet. id is the id of the on-path router that will have to perform the $mark_{src}$ verification. Regarding the marks present in the header: $mark_{src}$ is calculated by the source and verified by the on-path router with the same id as the one present in the header; $mark_r$ is a mark updated on each router and $mark_{pkt}$ is a mark calculated by the source and verified at the destination.

4.4.2 Source

At the source host, we execute a series of operations before crafting and sending a packet with the MASK header. We first define the Checkpoint Sequence Hopping (CSH), in this example, it is [5, 10]. That means for the first packet sent, epoch will be equal to 0 and seq will be equal to 5. Then the next one will have epoch=0 and seq=4, etc. Until seq reaches 0 then the epoch will be equal to 1 and seq will be equal to 10. We pick a random router id that will be on the packet path, this will be the value of the "id" in the header and define the router on the path that needs to calculate $mark'_{src}$ and compare it to $mark_{src}$. Next, it calculates the CSTpkt which is the concatenation of the PATH that the packet will take with the flag, the epoch, the seq and the id of the on-path router.

$$\text{CSTpkt=(PATH || flag || epoch || seq || id)}$$

Based in the router id, the source takes the id of the hop before it and use these ids, the CSTpkt as well as the HMAC in SRv6 to calculate with the function "calculate_marki(CSTpkt, idi_1, idi, HMAC)" the $mark_{src}$ with the RDK_{si}

$$mark_i = MAC_{RDK_{si}}(\text{CSTpkt} \mid\mid id_{i-1} \mid\mid id_i \mid\mid \text{HMAC})$$

$$mark_{src_i} = mark_i[0{:}32]$$

This is translated like this in Python:

```
for epoch in range ( len ( CSH_M ) ) :
    seq_count = CSH_M [ epoch ]
    while seq_count > 0:
    # Create constant packet header ( CSTpkt )
    id_router = random . randint (0 , len ( IDrouter_on_path ) -1)
    CSTPath = " "
    for i in range ( len ( PATH ) ) :
        CSTPath += PATH [ i ]
    CSTpkt = ( CSTPath + str ( epoch ) + str ( seq_count ) +
    IDrouter_on_path [ id_router ]) . encode ()

    # Determine the id and calculate the marks
    idi = IDrouter_on_path [ id_router ]
    if id_router == 0:
        idi_1 = " 0 "
    else :
        idi_1 = IDrouter_on_path [ id_router - 1]
```

Where the function "calculate_marki()" is defined in the following:

```
# Compute marksrc and marki
def calculate_marki(CSTpkt, idi_1, idi, AUTH):
    mac = hmac.new(RDKpath[int(idi)-1].encode(), digestmod=hashlib.
        sha256)
    mac.update(CSTpkt + idi_1.encode() + idi.encode() + AUTH)
    marki = mac.digest()
    return int.from_bytes(marki[:4], 'big'), int.from_bytes(marki
        [4:8], 'big')
```

We take as input the CSTpkt , idi_1 , idi ,and AUTH and use them to calculate the marki. Which will then be truncated to 4 bytes (0:32 bits) to generate the $mark_{src}$ and truncated from the 4th byte to the 8th (32:64 bits) to generate the $mark_i$. The CSTpkt is then used again to calculate the $mark_{pkt}$ with an initial hash.

This mark is calculated by the function "init_markpkt(CSTpkt)" $mark_r$=hash XOR $mark_1$[32:64]··· XOR $mark_n$[32:64] The function returns the newly generated $mark_{pkt}$ to be embedded in the MASK header as well as the initial hash which will be used to calculate the full $mark_r$. Regarding the full $mark_r$, it will be generated by the function "calculate_full_markr(init_hash, CSTpkt, IDrouter_on_path, HMAC)". The IDrouter_on_path is a list that contains the ids of the on path router. The function then returns the $mark_r$ that will be embedded in the MASK header.

```
    markpkt , init_hash = init_markpkt ( CSTpkt )
    markr = full_markr (init_hash , CSTpkt , IDrouter_on_path ,
        AUTH)
```

Here is the definition of the "init_markpkt()" function:

```
def init_markpkt(CSTpkt):
    mac = hmac.new(HDK.encode(), digestmod=hashlib.sha256)
    init_hash = (uuid.uuid4().bytes)[:4]
    mac.update(CSTpkt + init_hash)
    init_markpkt = mac.digest()[:4]
    return int.from_bytes(init_markpkt, 'big'), int.from_bytes(
        init_hash, 'big')
```

and "full_markr()":

```
#Compute the full markr (markr = init_hash XOR markr1[32:64] XOR
    markr2[32:64]...markri[32:64])
def full_markr(init_hash, CSTpkt, IDrouter_on_path, AUTH):
    markr = init_hash
    for i in range (len(IDrouter_on_path)):
        if i == 0:
            _,markri = calculate_marki (CSTpkt, "0",
                IDrouter_on_path[i], AUTH)
        else:
```

```
            _,markri = calculate_marki (CSTpkt, IDrouter_on_path[i
                -1], IDrouter_on_path[i], AUTH)
        markr = markr ^ markri
    return markr
```

The goal of this mark is that every on-path router will update this mark and at the end host, the $mark_r$ will be equal to the init_hash so the destination can use it to compute $mark'_{pkt}$ and check it against $mark_{pkt}$ to validate that the packet has been processed by every router on-path. Now that every cryptographic element has been computed, we can create and send the packet with the function "create_packet(seq_count, epoch % 256, int(IDrouter_on_path[id_router]), $mark_{src}$, $mark_r$, $mark_{pkt}$)".

```
    packets_sent.append(packet)
        send(packet)
        packet.show()
```

The full code for sending packets (send.py) and the maskHeader.py that contains the definition of the MASK Header as well as the different functions to calculate the multiple *mark* (maskHeader.py) can be found in B.2(send.py) and B.1(maskHeader.py)

4.4.3 On path router

Once the packet is generated and sent, it will travel across the network and be processed through the designated routers. For an intermediate router to process the packet, we use NETFFILTER NFQUEUE, a Python library that allows incoming packets in the firewall to be placed in a queue and handled, modified, accepted or dropped by a user generated code [17]. For this, we need to establish a rule in the firewall to redirect incoming packets to this queue. To see this rule, refer to the subsection 4.2 to see the details of it.

As mentioned earlier, every router has an id and when a packet arrives, the router checks if this packet contains the MASK header (maskh). If yes, then we will retrieve this header as well as the path contained in the SRv6 header. The router then computes the CSTpkt with the retrieved information from the header (reminder that CSTpkt=(PATH || flag || epoch || seq || id) and that all of these parameters are present in the MASK Header).

Next, the router check if the id present in the MASK Header is equal to its on id. If yes, then the router calculate $mark'_{src}$ with the function "calculate_marki(CSTpkt, idi_1, idi, HMAC)". If this $mark'_{src}$ is equal to $mark_{src}$ present in the packet header, then the router has proof that this packet is coming from the source and will continue processing the packet, otherwise it drops it.

```
if idi == router_id:
    # Calculate marksrc
    expected_marksrc, _ = calculate_marki(CSTpkt, idi_1, idi, hmac)
    # Verify the marksrc
    if mask_header.marksrc != expected_marksrc:
        print("Source verification failed, dropping packet.")
        pkt.drop()
        return
    elif (mask_header.marksrc == expected_marksrc):
        print ("marksrc validated")
```

If the $mark_{src}$ has been validated or if it wasn't for this router to validate the $mark_{src}$, then the $mark_r$ will be updated using the same function but this time taking the returned $mark_r$ instead of the returned $mark_{src}$. It then updates the $mark_r$ present in the MASK header with the returned value of the function and forwards the packet to the next hop.

```
    _, expected_markri = calculate_marki(CSTpkt, idi_1, router_id,
        bytes(AUTH))
        # Update markr
        new_markr = mask_header.markr ^ expected_markri
```

```
        # Update the packet with new markr
        scapy_pkt[maskh].markr = new_markr
        pkt.set_payload(bytes(scapy_pkt))
        pkt.accept()
    else:
        pkt.drop()
```

You can find the full code of the router on path in the Appendix B.3

4.5 End host

Finally, when the packet goes through the intermediate router and reaches its destination, it still needs to undergo some verification before it can be validated. Like the processing at the router, we capture the incoming packet using NETFILTER NFQUEUE. Since it is a proof of concept we will assume that the receiveds packet that are not validated are dropped by the destination. Like at the intermediate routers, the end host will retrieve the information from MASK header to calculate the CSTpkt. With this CSTpkt and the $mark_r$ present in the header, it will calculate $mark'_{pkt}$. If every router on the path processed the packet correctly and updated the $mark_r$ correctly then the $mark_r$ at the destination should be the init_hash. It will then take the CSTpkt and the $mark_r$ and use it as arguments for this function "calculate_markpkt(CSTpkt, mask_header.markr)" to get $mark'_{pkt}$. This $mark'_{pkt}$ is compared to the $mark_{pkt}$ present in the header. If they are the same then the packet is validated, dropped otherwise.

```
# Calculate markpkt
expected_markpkt = calculate_markpkt(CSTpkt, mask_header.markr)
# Verify markpkt
if mask_header.markpkt != expected_markpkt:
    pkt.drop()
```

If the packet hasn't been dropped, the destination will check if the epoch counter has reached the threshold defined in the CSH_M. If yes then it will check if the epoch violates the policy. If yes it doesn't mean the packet will be dropped but it might indicate a potential issue with the router i.

```
    # Initialize the sCounter if necessary
        if epoch not in sCounter:
            sCounter[epoch] = {}
            epoch_counter[epoch] = 0  # Initialize the epoch counter

        if router_id not in sCounter[epoch]:
            sCounter[epoch][router_id] = 0

        # Increment the router-specific counter and the epoch counter
        sCounter[epoch][router_id] += 1
        epoch_counter[epoch] += 1

        # Check if the epoch counter has reached the threshold defined in CSH_M
        if epoch_counter[epoch] >= CSH_M[epoch]:
            for router in sCounter[epoch]:
                # Calculate the expected number of packets for this router
                N = CSH_M[epoch] / len(PATH)  # Assumed default policy
                print("N is:"+ str(N)+ " sCounter" + str(sCounter))
                # Check if the actual count is within 5% of the expected count
                if not (N * 0.95 <= sCounter[epoch][router] <= N * 1.05):
                    print(f"Verification failed at router {router}
```

```
                        in epoch {epoch}")
                else:
                    print(f"Verification succeeded for router {
                        router} in epoch {epoch}")
            # Clear the counters for the epoch
            sCounter[epoch] = {}
            epoch_counter[epoch] = 0

        print("Packet processed successfully at destination.")
```

You can find the full code of the end-host processing of the packet in the Appendix B.4

4.6 Outputs

We can see the state of a packet when the implementation is run. The picture below shows the packet sent by the source h1. The MASK header is present along with the SRH and TLV HMAC header.

At router R1 and R2, we can see the incoming $markr$ as well as the updated $markr$ forwarded and if the router validated the $mark_{src}$

And finally, we can see the destination host validating the $mark_{pkt}$ and validates the policy for the current $epoch$.

```
.
Sent 1 packets.
###[ IPv6 ]###
  version   = 6
  tc        = 0
  fl        = 0
  plen      = None
  nh        = Routing Header
  hlim      = 64
  src       = fc00:a::1
  dst       = fc00:b::4
###[ IPv6 Option Header Segment Routing ]###
     nh        = No Next Header
     len       = None
     type      = 4
     segleft   = 1
     lastentry = None
     unused1   = 0
     protected = 0
     oam       = 0
     alert     = 0
     hmac      = 0
     unused2   = 0
     tag       = 0
     addresses = [ fc00:a::2, fc00:b::3 ]
     \tlv_objects\
      |###[ IPv6 Option Header Segment Routing - HMAC TLV ]###
      |  type      = HMAC TLV
      |  len       = None
      |  D         = 0
      |  reserved  = 0
      |  hmackeyid = 0
      |  hmac      = ".K\\xe8\\xc8f\\xfaK\\x89k\\xe7\\xe8)\\xd7w\x1bX(:\x02\\xff\\xc9fH%'\x17Jd"
###[ maskh ]###
        flag      = 0
        epoch     = 1
        seq       = 1
        id        = 2
        marksrc   = 970931242
        markr     = 2475827092
        markpkt   = 1254066725

Packets sent successfully.
root@Pantsat:/home/scion-dev-vm/mask/P4#
```

Figure 4.1: Packet sent by H1

```
-------------------------------------NEXT packet-----------------------------------------
marksrc validated
markr=121685339, new_markr=2557639431
Packet forwarded with updated markr: 2557639431
-------------------------------------NEXT packet-----------------------------------------
markr=3525371460, new_markr=2135050623
Packet forwarded with updated markr: 2135050623
-------------------------------------NEXT packet-----------------------------------------
markr=2475827092, new_markr=3234103490
Packet forwarded with updated markr: 3234103490
-------------------------------------NEXT packet-----------------------------------------
```

Figure 4.2: Packet processed by R1

```
-------------------------------------NEXT packet-----------------------------------------
markr=2557639431, new_markr=441788282
Packet forwarded with updated markr: 441788282
-------------------------------------NEXT packet-----------------------------------------
marksrc validated
markr=2135050623, new_markr=4279114927
Packet forwarded with updated markr: 4279114927
-------------------------------------NEXT packet-----------------------------------------
marksrc validated
markr=3234103490, new_markr=1121681386
Packet forwarded with updated markr: 1121681386
-------------------------------------NEXT packet-----------------------------------------
```

Figure 4.3: Packet processed by R2

```
Destination Host: CSTpkt=b'fc00:a::2fc00:b::3131', markr=441788282, expected_markpkt=3336438410, markpkt=3336438410
Packet processed successfully at destination.
Packet contain MASKHeader.
Destination Host: CSTpkt=b'fc00:a::2fc00:b::3122', markr=4279114927, expected_markpkt=40669888, markpkt=40669888
Packet processed successfully at destination.
Packet contain MASKHeader.
Destination Host: CSTpkt=b'fc00:a::2fc00:b::3112', markr=1121681386, expected_markpkt=1254066725, markpkt=1254066725
N is:5.0 sCounter{0: {}, 1: {1: 5, 2: 5}}
Verification succeeded for router 1 in epoch 1
```

Figure 4.4: Packet processed by H2

5 Formal security proof

In this chapter, we will go through the introduction of the Tamarin Prover, defining the necessary vocabulary and concepts related to it. Then we will define the attacker present in the Tamarin Prover and what it is capable of. We will see the translation of MASK into a Tamarin Prover code, the definition of the security properties that needs to hold against the attacker and if they do resist or not.

5.1 Tamarin

Tamarin is a verification tool for security protocols that supports both falsification and unbounded verification within the symbolic model, offering formal guarantees for security proofs. In Tamarin, security protocols are defined as multiset rewriting systems and analyzed against first-order properties, a message theory, and a user-defined rewriting theory. A protocol model includes specifications of the actions for each participating agent, the adversary, and the properties to be verified. In Tamarin, this model is referred to as a theory. Tamarin attempts to automatically construct a proof for the given properties, even when numerous instances of the protocol run in parallel. Its search algorithm for trace properties is both sound and complete, although it is not guaranteed to terminate. If it does terminate, the result is either a full proof of correctness for all possible traces or a trace that demonstrates an attack on the property. Tamarin's default attacker model is Dolev–Yao, but alternative attacker models can be specified.

5.2 Terms

In Tamarin, a *term* is a fundamental building block used to model cryptographic messages. It serves as a symbolic representation constructed using constants, variables, and function symbols. Terms can represent simple atomic values or more complex structures, depending on the functions applied to them.

Constants are fixed values such as public names, like the identities of agents, and special constants. They are usually denoted by strings in single quotes, for example, `'Alice'`. Variables, on the other hand, are placeholders that can represent any term. They can be of different sorts: $\sim x$ denotes a fresh variable, which is a unique, randomly generated value; `$x` denotes a public variable, which can represent any public value; `%x` denotes a natural number variable; and `#i` denotes a temporal variable used to represent time points.

Function symbols are used to construct terms from other terms. Tamarin includes a set of built-in functions for common cryptographic operations and allows users to define additional functions. For example, the hashing function `h(x)` represents a cryptographic hash of `x`, the binary function `aenc(m, pk)` represents the asymmetric encryption of message `m` with public key `pk`, and `adec(c, sk)` represents the asymmetric decryption of ciphertext `c` with secret key `sk`.

Terms are constructed by applying these function symbols to other terms. For instance, `h('message')` represents the hash of the string constant `'message'`, and `aenc(`$\sim k$`, pk($S))` represents the asymmetric encryption of a fresh key $\sim k$ with the public key of some server `$S`.

Tamarin supports equational theories to model the properties of cryptographic functions, defining how terms can be transformed. An example of such an equation is `adec(aenc(m, pk(sk)), sk) = m`, which specifies that decrypting an encrypted message with the correct key yields the original message.

5.3 Rules

In Tamarin, a *rule* is a construct used to describe how the state of a security protocol changes over time. Rules define the conditions under which certain actions can transform the current state into a new state. This transformation is articulated through the manipulation of *facts*, which are predicates that capture information about the protocol's state.

A Tamarin rule is composed of three main components: premises, actions, and conclusions. The premises are conditions that must be satisfied for the rule to be applicable. These are expressed as a set of facts that must be present in the current state. The actions are optional labels that indicate observable events or actions that occur when the rule is executed. These labels are crucial for specifying security properties and tracking events. The conclusions are the new set of facts that will hold true after the rule is applied, replacing the premises in the system's state.

For instance, consider a simple protocol where a client sends an encrypted message to a server, and the server replies with a hash of the decrypted message. This can be modeled in Tamarin with three rules. The first rule, `Client_Send`, represents the client sending an encrypted message. The premises involve generating a fresh key and having the server's public key. The conclusions include storing the server and key for the next step and sending the encrypted key.

The second rule, `Server_Receive`, depicts the server receiving the client's message. The premises are the server's private key and the incoming request. The actions log that the server has processed the request, and the conclusions involve sending back the hash of the decrypted request.

The third rule, `Client_Receive`, shows the client receiving the server's response. The premises include the stored server and session key as well as the received hash. The actions log that the session key was established with the server.

Here is an example of how these rules might look in Tamarin:

```
rule Client_Send:
    [ Fr(~k), !Pk($S, pkS) ]
  -->
    [ Client_1($S, ~k), Out(aenc(~k, pkS)) ]

rule Server_Receive:
    [ !Ltk($S, ~ltkS), In(request) ]
  --[ AnswerRequest($S, adec(request, ~ltkS)) ]->
    [ Out(h(adec(request, ~ltkS))) ]

rule Client_Receive:
    [ Client_1($S, k), In(h(k)) ]
  --[ SessKeyC($S, k) ]->
    []
```

In these rules, *facts* represent pieces of state information, such as `Out(x)` for a message sent on the public channel or `In(x)` for a message received. Fresh values, denoted by `Fr(`$\sim k$`)`, indicate newly generated values. Persistent facts, marked by `!Fact`, are facts that can be reused without being consumed when the rule is applied. Action facts, specified in the action part of the rule using `-[ Action ]->`, are used to log significant events or actions that have taken place.

5.4 Properties

In Tamarin, *properties* refer to the security requirements or goals that a protocol aims to achieve. These properties are expressed as logical statements about the traces of the protocol's execution. Properties in Tamarin are used to specify and verify the desired security guarantees of a protocol, such as confidentiality, authentication, and integrity.

Properties in Tamarin are expressed using lemmas. A lemma is a formal statement that describes a property or security requirement of the protocol. These lemmas are written in a formal language and are used to specify what should hold true for all possible executions (traces) of the protocol.

There are different types of properties that can be specified in Tamarin. Trace properties describe conditions that must hold over the entire execution trace of the protocol. They are used to specify requirements such as confidentiality, ensuring that certain information remains secret; authentication, ensuring that the identities of the communicating parties are verified; and integrity, ensuring that data has not been tampered with. Another type of property is observational equivalence, which is used to specify that two different processes or protocol executions are indistinguishable to an adversary. This type of property is often used to verify privacy properties.

To specify properties, Tamarin uses lemmas, which are written in a formal language supported by the tool. For example, a lemma specifying a confidentiality property might look like this:

```
lemma Client_session_key_secrecy:
" /* It cannot be that a */
 not(
  Ex S k #i #j.
    /* client has set up a session key 'k' with a server'S' */
    SessKeyC(S, k) @ #i
    /* and the adversary knows 'k' */
  & K(k) @ #j
    /* without having performed a long-term key reveal on 'S'. */
    & not(Ex #r. LtkReveal(S) @ r)
 )
```

"

In this example, the lemma `Client_session_key_secrecy` specifies that it should not be possible for an adversary to know the session key `k` set up between a client and a server unless the adversary has compromised the server's long-term key. This is expressed as a logical formula where `SessKeyC(S, k)` denotes the setup of the session key, and `K(k)` denotes the knowledge of the key by the adversary. The property also checks that there is no instance of the server's long-term key being revealed (`LtkReveal(S)`).

Tamarin uses automated and interactive proof techniques to verify these properties. If a lemma is provenn to be true, it means the protocol satisfies the specified property under all possible executions. If a lemma is false, Tamarin provides a counterexample, which is an execution trace that violates the property, thereby identifying potential vulnerabilities in the protocol. [18]

5.5 Dolev-Yao attacker model

The Dolev–Yao model refers to a symbolic attacker model in which the adversary has complete control over the network. They have the ability to drop, alter, eavesdrop and even inject new messages. They are able to carry out cryptographic functions. Additionally, we take for granted that we possess perfect cryptographic primitives. Dolev-Yao constitutes the default attacker model in Tamarin and by using In and *Out* facts to represent message exchange between participants, we provide the adversary with all the network capabilities that the attacker model predicts it should have. A rule that outputs an Out fact along with the message is used to describe the process by which a party sends a message through the adversary-controlled network. Anything that is contained In an *Out* fact becomes known to the enemy. All messages in the adversary knowledge base can be combined, and any function in the model can be applied to them by the adversary. The adversary can then create an *In* fact using any message of that kind. A rule consuming an *In* fact represents a party receiving a message across the network. The opponent is aware of the value of a term if it is initialised as a constant or public variable. The opponent does not know secret keys if they were not disclosed since the only terms they do not know by default are new values. Tamarin is providing by default all these rules to the adversary.

5.6 Modeling the protocol

Before going into the details of the modeling of the protocol, it is important to note that the model built here is designed to prove source authentication and thus some of its components are omitted. However, these components do not take part in source authentication in the protocol so omitting them or not should not interfere with the results. With that out of the way, let's dive into the modelisation of the MASK protocol in Tamarin. We first import the "hash" builtin function in Tamarin that we will use to generate our HMAC. We then are defining our own abstract function such as MAC for "Message Authentication Code", the "/2" specifies it takes 2 arguments as an input to generate one output and "truncate_32/1) that is the truncation to the lowest 32-bits of a value.

We then have our first rule "generateHMACKey" that generates a fresh HMAC key ∼hmac_key and associates it with a source $S and a receiver $R using the !HMAC_key($S, $R, ∼hmac_key) fact. The CreateHMACKeys($S, $R, ∼hmac_key) action fact is used to log the creation of the HMAC keys.

```
rule generateHMACkey:
 [Fr(~hmac_key)] --[CreateHMACKeys($S, $R, ~hmac_key)]->[!HMAC_key(
    $S, $R, ~hmac_key)]
```

To simuate the keys shared between the AS of the source and the AS of and on path router the "generateAK" rule generates a fresh AS-key (AK), represented as "∼AKsi," and associates it with a source $S and a receiver $R using the "!AK($S, $R, ∼AKsi)" fact. The "CreateAK($S, $R, ∼AKsi)" action fact serves to log the creation of the AK.

```
rule generateAK:
 [Fr(~AKsi)] --[CreateAK($S, $R, ~AKsi)]-> [!AK($S, $R, ~AKsi)]
```

We then need to model the generation and distributions of RDKs, that's where the "generateRDK" rule comes into play, generating a router dynamic key (RDK) derived from the AK. It takes the AK "AKsi" and combines it with the router identifier "$idi" and an "indicator" (consisting of a fresh timestamp "∼ts" and a fresh user identifier "∼uid") using the message authentication code function "MAC". The resulting RDK is then associated with the router "$R" and the source "$S" using the "!RDK_Router($R, $idi, RDK)" and "!RDK_Source($S, $idi, RDK)" facts, respectively. The "CreateRDK($S, $R, $idi, RDK)" action fact logs the creation of the RDK.

```
rule generateRDK:
 let
  indicator = <~ts, ~uid>
  RDK = MAC(AKsi, <$idi, indicator>)
 in
  [!AK($S, $R, AKsi), Fr(~ts), Fr(~uid)]
  --[CreateRDK($S, $R, $idi, RDK)]->
  [!RDK_Router($R, $idi, RDK), !RDK_Source($S, $idi, RDK)]
```

The "real_source" rule models the behavior of a source sending a packet in the MASK protocol. It starts by computing the HMAC of the source and destination IP addresses using the HMAC key "hmac_key." It then constructs the constant part of the packet, "CSTpkt," using various parameters such as the source IP "$IPsrc," path information "$PATH," flag "$flag," sequence number "$seq," epoch "$epoch," and router identifier "$idi" as specified in the MASK documentation. The mark "marki" is computed using the MAC function, taking the RDK, "CSTpkt," the previous router identifier "$idi_1," and the current router identifier "$idi" as input. The "marksrc" is obtained by truncating "marki" to 32 bits with the "truncate_32" function defined earlier. The source then sends out into the network the "HMAC," "$idi_1," "$idi," and "marksrc" using the "Out" fact. The "Source_create_marksrc($S, $idi, RDK, marksrc)" action defines the creation of "marksrc" by the source, while the "Unique_RDK_use($S, $idi, RDK)" action fact ensures that each RDK is used only once by a source. The "Running($S, $R, marksrc)" action fact indicates that the source "$S" is running the protocol with the receiver "$R" and the "marksrc."

```
rule real_source:
 let
  HMAC = h(<$IPsrc, $IPdst,hmac_key>)
  CSTpkt = <$IPsrc, $PATH, $flag, $seq, $epoch, $idi>
  marki = MAC(RDK, <CSTpkt, $idi_1, $idi, HMAC>)
  marksrc = truncate_32(marki)
 in
  [!RDK_Source($S, $idi, RDK), !HMAC_key($S, $R, hmac_key)]
  --[Source_create_marksrc($S, $idi, RDK, marksrc),
     Unique_RDK_use($S, $idi, RDK),
     Running($S, $R, marksrc)]->
     [Out(<HMAC, $idi_1, $idi, marksrc>)]
```

Moving on to the "router" rule, which models the behavior of the router in the MASK protocol, it takes as an input a message containing the "HMAC," "$idi_1," "$idi," and "marksrc" using the "In" fact. The router then computes the "marksrc_prime" using the same process as the source, utilizing its own RDK. If the received "marksrc" matches the computed "marksrc_prime," the router considers the source as authenticated, as indicated by the "Router_validates_source($idi, RDK, marksrc, marksrc_prime)" action fact. The "Eq(marksrc, marksrc_prime)" action fact represents the equality check between "marksrc" and "marksrc_prime." Finally, the "Commit($R, $S, marksrc)" action fact signifies that the router "$R" has committed to the source "$S" with the "marksrc."

```
rule router:
 let
  HMAC = h(<$IPsrc, $IPdst,hmac_key>)
  CSTpkt = <$IPsrc, $PATH, $flag, $seq, $epoch, $idi>
  marki = MAC(RDK, <CSTpkt, $idi_1, $idi, HMAC>)
  marksrc_prime = truncate_32(marki)
 in
  [!RDK_Router($R, $idi, RDK), In(<HMAC, $idi_1, $idi, marksrc>), !
     HMAC_key($S, $R, hmac_key)]
  --[Router_validates_source($idi, RDK, marksrc, marksrc_prime),
     Eq(marksrc, marksrc_prime),
     Commit($R, $S, marksrc)]-> []
```

We also need to specify two important restrictions. The "Unique_RDK_usage" restriction ensures that each RDK is used only once by a source, thereby preventing replay attacks. The "Equality" restriction that is quite self explanatory but needs to be defined manually in Tamarin.

```
// Restriction to ensure each RDK is only used once by a source
restriction Unique_RDK_usage:
 "All S idi RDK #i #j. Unique_RDK_use(S, idi, RDK)@i &
    Unique_RDK_use(S, idi, RDK)@j ==> #i = #j"
restriction Equality:
 "All x y #i. Eq(x,y) @i ==> x = y"
```

5.7 Correctness proof

The correctness proof in Tamarin is simply a set of instructions defining which actions should be true at a given time. The "exist-trace" means that there is at least a trace for which this property holds. In our case, we first want our model to create HMAC keys at time l, then RDKs at time i, and then a source needs to create a *marksrc* at time j. Finally, a router needs to validate the source at time k AND the *marksrc* at this time must be equal to *marksrc_prime*

```
lemma executable:
exists-trace
  "Ex S R idi RDK marksrc hmac_key marksrc_prime #i #j #k #l.
   CreateHMACKeys(S, R, hmac_key) @l &
   CreateRDK(S, R, idi, RDK) @ i &
   Source_create_marksrc(S, idi, RDK, marksrc) @ j &
   Router_validates_source(idi, RDK, marksrc, marksrc_prime) @ k &
   Eq(marksrc, marksrc_prime) @ k &
   #l < #i & #i < #j & #j < #k"
```

When run, Tamarin gives us the following outcome meaning the proof succeeded.

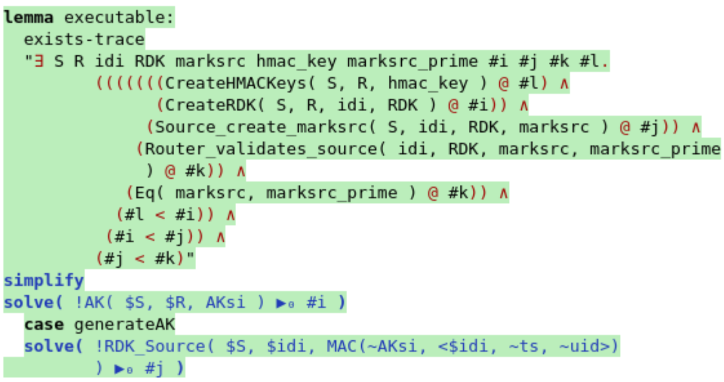

Figure 5.1: lemma executable results

5.8 Replay attacks proof

The MASK paper states that replay attack protection cannot be detected since the on-path router is stateless, this lemma validates that theory.

The lemma states that for all routers R, sources S, and marksrc values, if there is a Commit event at time i, then:

There exists a Running event at time j, where the source S is running the protocol with the router R and the same marksrc value. The Running event occurs before the Commit event (j < i). There does not exist another Commit event at time i2 with the same marksrc value but with different participants (R2 and S2) than the original Commit event at time i.

```
lemma injectiveagreement: //prove that it's not resilient to replay
    attack
 "All R S marksrc #i.
 Commit(R, S,marksrc) @i
 ==> (Ex #j. Running(S, R, marksrc) @j
 & j < i
 & not (Ex R2 S2 marksrc #i2. Commit(R2, S2, marksrc) @i2
 & not (#i2 = #i)))"
```

When run, we get the following meaning the proof failed and the lemma is not true and MASK is vulnerable to replay attacks at on-path routers.

```
lemma injectiveagreement:
  all-traces
  "∀ R S marksrc #i.
         (Commit( R, S, marksrc ) @ #i) ⇒
         (∃ #j.
           ((Running( S, R, marksrc ) @ #j) ∧ (#j < #i)) ∧
           (¬(∃ R2 S2 marksrc.1 #i2.
               (Commit( R2, S2, marksrc.1 ) @ #i2) ∧ (¬(#i2 = #i)))))"
simplify
solve( !RDK_Router( $R, $idi, RDK ) ▶₀ #i )
  case generateRDK
  solve( !HMAC_key( $S, $R, hmac_key ) ▶₂ #i )
    case generateHMACkey
    solve( !KU( h(<$IPsrc, $IPdst, ~hmac_key>) ) @ #vk.1 )
      case c_h
      by solve( !KU( ~hmac_key ) @ #vk.11 )
    next
      case real_source
```

Figure 5.2: lemma injective results

5.9 Source Authentication proof

This lemma states that for every Commit event by a router R with a source S and a marksrc, two conditions must hold: There exists a corresponding Running event by the source S with the same marksrc, and this Running event occurs before the Commit event. For all Commit events by the same router R with any source S2 and the same marksrc, the source S2 must be the same as the original source S.

The first condition ensures that the source has indeed run the protocol with the same marksrc before the router commits to it. The second condition guarantees that for a given router and marksrc, there is only one unique source associated with it across all Commit events. This prevents multiple sources from using the same marksrc with the same router asserting that every marksrc are unique and coming from the same source thus enforcing the source authentication property.

```
lemma source_authentication_unique:
  "All R S marksrc #i.
    Commit(R, S, marksrc) @ i //Commit(R2, S2, <'source', 'router',
        marksrc>)
    ==> (Ex #j. Running(S, R, marksrc) @ j & j < i)
        & (All S2 #i2. Commit(R, S2, marksrc) @ i2 ==> S = S2)"
```

When run, we get the following result. The proof succeeded meaning a marksrc

```
lemma source_authentication_unique:
  all-traces
  "∀ R S marksrc #i.
         (Commit( R, S, marksrc ) @ #i) ⇒
         ((∃ #j. (Running( S, R, marksrc ) @ #j) ∧ (#j < #i)) ∧
          (∀ S2 #i2. (Commit( R, S2, marksrc ) @ #i2) ⇒ (S = S2)))"
simplify
solve( (∀ #j.
         (Running( $S, $R,
                   truncate_32(MAC(RDK,
                                   <<$IPsrc, $PATH, $flag, $seq, $epoch, $idi
                                    $idi, h(<$IPsrc, $IPdst, hmac_key>)>))
          ) @ #j)
        ⇒
         ¬(#j < #i)) ‖
```

Figure 5.3: lemma source authentication results

is associated with one unique source router and validating source authentication.

6 Conclusion

The purpose of this thesis is to investigate and verify the Multi-AS-Key (MASK) protocol, which is intended to improve security in satellite communication networks. My main goals were to understand the fundamentals of MASK and Segment Routing over IPv6 (SRv6), create an implementation proof-of-concept, and use the Tamarin prover to formally validate the security features of the protocol. I now conclude by discussing how these goals were achieved and their contributions to the field of satellite communication security. I started by understanding the concepts of SRv6 and MASK. The workings of SRv6 were explained in detail, demonstrating how senders can specify the whole path for packets via Segment Routing Headers (SRH) by utilising source routing. This approach makes traffic engineering easier, lowers network state, and provides explicit routing. I pointed out the MASK protocol as a security enhancement, designed to improve data transmission security with minimal computational overhead. I did a software implementation of MASK integrated with SRv6 in a simulated environment using Mininet and IPMininet after completing the necessary background research. The integration of MASK with SRv6 was demonstrated in this controlled environment. A thorough explanation was given in the implementation section, which covered the definition of the MASK header, configuring the network architecture, and the logic for packet processing over the network. The proof-of-concept implementation proved that MASK could be implemented successfully. I employed the Tamarin prover, which is adept at analysing and confirming cryptographic protocols, for the formal verification of the MASK protocol. I went into depth on how MASK was modelled inside the Tamarin framework, paying particular attention to confirming important security attributes like data integrity, source authentication, and resistance to replay attacks. The formal proofs produced by the Tamarin prover verified that MASK achieves the security goals described in the original paper.

7 Future work

Various amounts of further work can still be done on satellite communications and the MASK protocol. While a proof of concept has been implemented and some security proofs demonstrated, the project is only at the beginning and a lot is to be done to fully adopt MASK into satellite communications.

The first and soonest improvement to do would be to implement support for multi-path. This is essential for two reasons. The first one is because it is crucial for our network to have multi-path since we are planning on using both the space and ground networks. We talked about the availability of the satellite network in Section 3.1. However, due to some events (link down, satellite getting out of reach before sending all of the data, low transfer rate, etc.) it is important for our source to switch paths quickly in case of a failure or use multiple ones to distribute the load. Implementing multi-path will also allow for extensive performance testing of the protocol (throughput, goodput, bandwidth, packet size, etc.) in a more realistic environment.

Another milestone for the future would be to fully implement the control plane. This would mean further security testing with key exchange. This is a huge task that was beyond the scope of this master thesis.

Finally, a full model of MASK as well as security testing in the Tamarin prover to prove the remaining security features claimed in the original paper, such as packet authentication and path validation would be a good extension of this thesis. Then, when multipath is implemented or designed for MASK, additional modelling and security testing could be done in Tamarin to prove that multipath is resilient (or not) to common attacks and keep guaranteeing the desired security features that we want to have with path aware networking.

References

[1] *Scion website*, Accessed: 23-05-2024. [Online]. Available: https://scion-architecture.net/.

[2] L. Chuat, M. Legner, D. Basin, *et al.*, *The complete guide to scion. from design principles to formal verification*, en, Edited Volume, Cham, 2022. DOI: 10.1007/978-3-031-05288-0.

[3] M. Legner, T. Klenze, M. Wyss, C. Sprenger, and A. Perrig, "Epic: Every packet is checked in the data plane of a path-aware internet", in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20, USA: USENIX Association, 2020, ISBN: 978-1-939133-17-5. DOI: 10.5555/3489212.3489243.

[4] A. He, K. Bu, J. Huang, Y. Pang, Q. Gu, and K. Ren, "Swiftparade: Anti-burst multipath validation", *IEEE Transactions on Dependable and Secure Computing*, pp. 1–16, 2023. DOI: 10.1109/TDSC.2023.3315457.

[5] C. Krähenbühl, M. Wyss, D. Basin, V. Lenders, A. Perrig, and M. Strohmeier, "FABRID: Flexible Attestation-Based routing for Inter-Domain networks", in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 5755–5772, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/krahenbuhl.

[6] L. Iannone and A. Fressancourt, "Segment Routing over IPv6 (SRv6) Proof of Transit", Internet Engineering Task Force, Internet-Draft draft-iannone-spring-srv6-pot-00, Feb. 2024, Work in Progress, 11 pp. [Online]. Available: `https://datatracker.ietf.org/doc/draft-iannone-spring-srv6-pot/00/`.

[7] T. Zhang, H. Li, J. Liu, *et al.*, "High-risk leo satellite network path detection based on spatial and temporal delay anomaly analysis", *China Communications*, vol. 20, no. 7, pp. 57–71, 2023. DOI: `10.23919/JCC.fa.2023-0077.202307`.

[8] J. Lawton, "Numerical method for rapidly determining satellite-satellite and satellite-ground station in-view periods", *Journal of Guidance, Control, and Dynamics*, vol. 10, no. 1, pp. 32–36, 1987.

[9] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, *Segment Routing Architecture*, RFC 8402, Jul. 2018. DOI: `10.17487/RFC8402`. [Online]. Available: `https://www.rfc-editor.org/info/rfc8402`.

[10] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, *IPv6 Segment Routing Header (SRH)*, RFC 8754, Mar. 2020. DOI: `10.17487/RFC8754`. [Online]. Available: `https://www.rfc-editor.org/info/rfc8754`.

[11] B. Trammell, *Current Open Questions in Path-Aware Networking*, RFC 9217, Mar. 2022. DOI: `10.17487/RFC9217`. [Online]. Available: `https://www.rfc-editor.org/info/rfc9217`.

[12] B. Sengupta, Y. Li, K. Bu, and R. H. Deng, "Privacy-preserving network path validation", *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 1, pp. 1–27, 2020.

[13] L. Gommans, C. de Laat, and R. Meijer, "Token based path authorization at interconnection points between hybrid networks and a lambda grid", in *2nd In-*

ternational Conference on Broadband Networks, 2005., IEEE, 2005, pp. 1378–1385.

[14] H. Cai and T. Wolf, "Source authentication and path validation with orthogonal network capabilities", in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2015, pp. 111–112.

[15] S. Fu, Q. Li, M. Zhu, *et al.*, "Mask: Practical source and path verification based on multi-as-key", *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1478–1493, 2023. DOI: `10.1109/TNET.2022.3222610`.

[16] Accessed: 20-05-2024. [Online]. Available: `https://github.com/cnp3/ipmininet`.

[17] *Netfilterqueue 1.1.0*, Accessed : 12-05-2024. [Online]. Available: `https://pypi.org/project/NetfilterQueue/`.

[18] *Tamarin-prover manual*, Accessed: 25-05-2024. [Online]. Available: `https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf`.

Appendix A Tamarin results

A.1 Injective agreement results

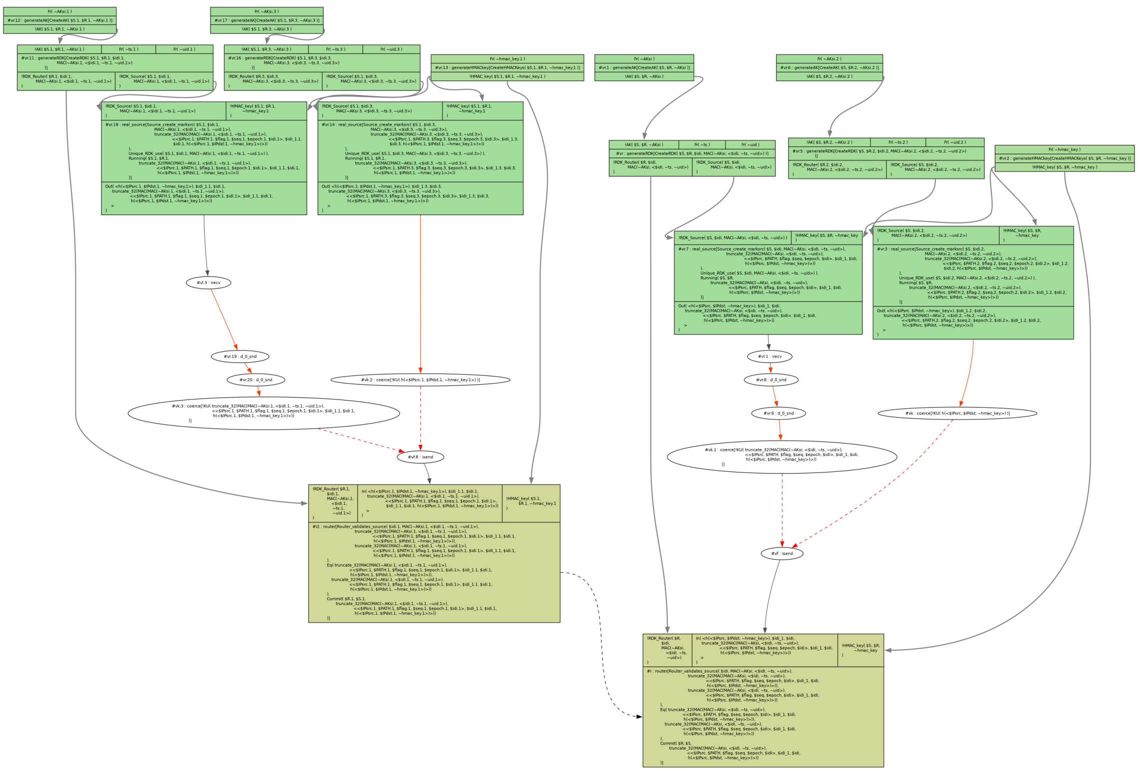

Figure A.1: Injective agreement flowchart results

A.2 Source authentication results

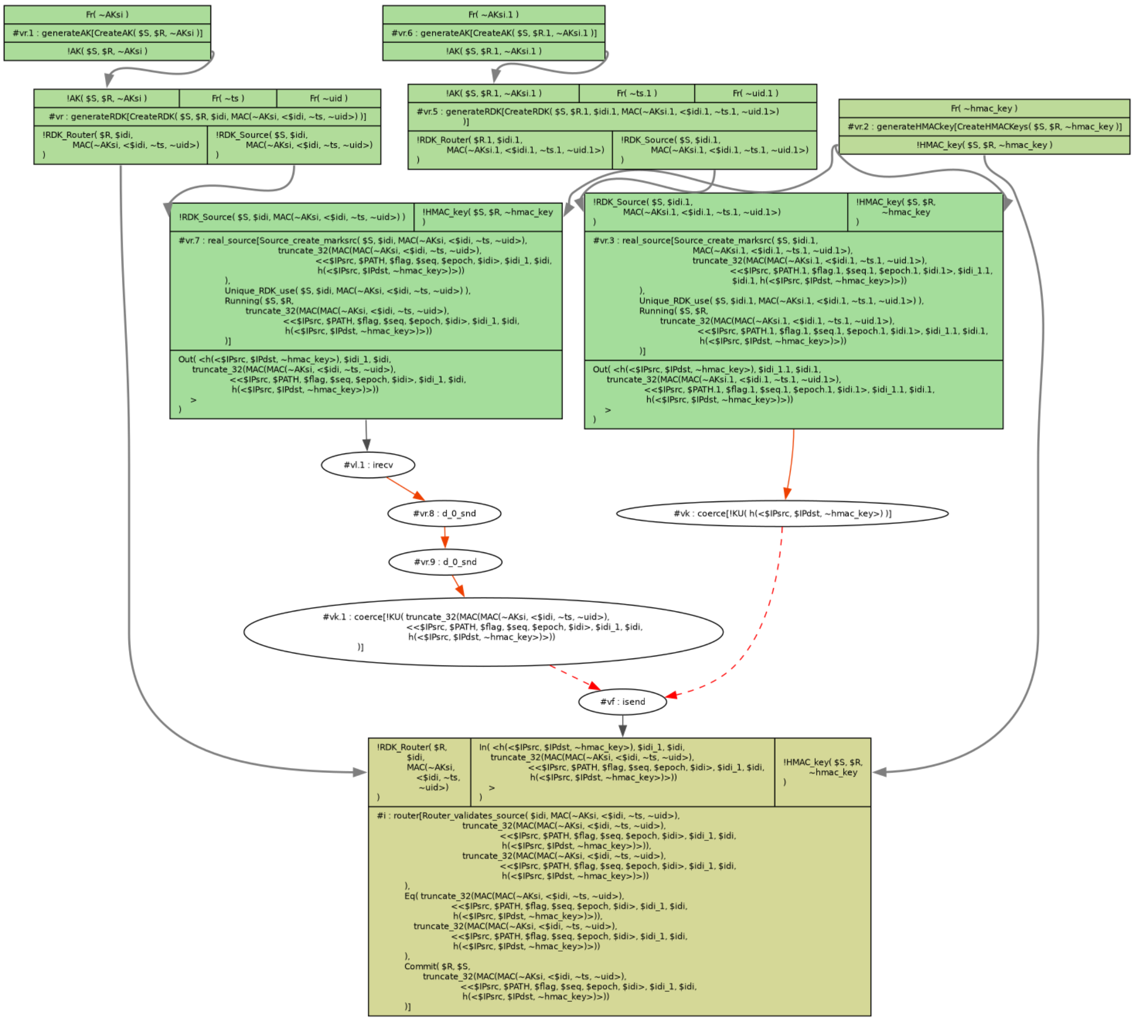

Figure A.2: Source authentication flowchart results

Appendix B List of codes

B.1 maskHeader.py

```python
import uuid
import random
import hmac
import hashlib
from scapy.all import *
from scapy.layers.inet6 import IPv6, IPv6ExtHdrSegmentRouting,
    IPv6ExtHdrSegmentRoutingTLVHMAC
# Define the custom header
class maskh(Packet):
    name = "maskh"
    fields_desc = [
        BitField("flag", 0, 8),
        BitField("epoch", 0, 8),
        BitField("seq", 0, 16),
        BitField("id", 0, 32),
        BitField("marksrc", 0, 32),
        BitField("markr", 0, 32),
        BitField("markpkt", 0, 32)
    ]

bind_layers(IPv6ExtHdrSegmentRouting, maskh)

```

```python
# Define constants and keys
PATH = ["fc00:a::2", "fc00:b::3"]
RDKpath=["11111111","22222222"]
IDrouter_on_path = ["1","2"]
HDK = "1234567890"
CSH_M = [5, 10]

def calculate_hmac(key, message):
    return hmac.new(key.encode(), message, hashlib.sha256).digest()
AUTH = calculate_hmac(HDK, b'AUTH')

# Compute marksrc and marki
def calculate_marki(CSTpkt, idi_1, idi, AUTH):
    mac = hmac.new(RDKpath[int(idi)-1].encode(), digestmod=hashlib.
        sha256)
    mac.update(CSTpkt + idi_1.encode() + idi.encode() + AUTH)
    marki = mac.digest()
    return int.from_bytes(marki[:4], 'big'), int.from_bytes(marki
        [4:8], 'big')

#Calculate the markpkt based on input markr (used for the markpkt
    check at end host), the input markr at end host should be the
    same as the hash in "init_markpkt"
def calculate_markpkt(CSTpkt, markr):
    mac = hmac.new(HDK.encode(), digestmod=hashlib.sha256)
    mac.update(CSTpkt + markr.to_bytes(4, 'big'))
    markpkt = mac.digest()
    return int.from_bytes(markpkt[:4], 'big')

#initialize the markpkt with CSTpkt and a hash
def init_markpkt(CSTpkt):
    mac = hmac.new(HDK.encode(), digestmod=hashlib.sha256)
    init_hash = (uuid.uuid4().bytes)[:4]
```

```
    mac.update(CSTpkt + init_hash)
    init_markpkt = mac.digest()[:4]
    return int.from_bytes(init_markpkt, 'big'), int.from_bytes(
        init_hash, 'big')

#Compute the full markr (markr = init_hash XOR markr1[32:64] XOR
    markr2[32:64]...markri[32:64])
def full_markr(init_hash, CSTpkt, IDrouter_on_path, AUTH):
    markr = init_hash
    for i in range (len(IDrouter_on_path)):
        if i == 0:
            _,markri = calculate_marki (CSTpkt, "0",
                IDrouter_on_path[i], AUTH)
        else:
            _,markri = calculate_marki (CSTpkt, IDrouter_on_path[i
                -1], IDrouter_on_path[i], AUTH)
        markr = markr ^ markri
    return markr



def create_packet(seq, epoch, id, marksrc, markr, markpkt):
    pkt = IPv6(src="fc00:a::1", dst="fc00:b::4") /
        IPv6ExtHdrSegmentRouting(addresses=PATH,segleft=len(PATH)-1,
         tlv_objects=IPv6ExtHdrSegmentRoutingTLVHMAC(hmac=AUTH))/
        maskh(flag=0, epoch=epoch, seq=seq, id=id, marksrc=marksrc,
        markr=markr, markpkt=markpkt)
    return pkt
```

B.2 send.py

```
import uuid
```

```
import random
import hmac
import hashlib
from scapy.all import *
from scapy.layers.inet6 import IPv6, IPv6ExtHdrSegmentRouting,
    IPv6ExtHdrSegmentRoutingTLVHMAC
from maskHeader import *

#Main procedure at Source Host
packets_sent=[]
for epoch in range(len(CSH_M)):
    seq_count = CSH_M[epoch]
    while seq_count > 0:
        # Create constant packet header (CSTpkt)
        id_router = random.randint(0, len(IDrouter_on_path)-1)
        CSTPath = ""
        for i in range (len(PATH)):
            CSTPath += PATH[i]
        CSTpkt = (CSTPath+ str(epoch) + str(seq_count) +
            IDrouter_on_path[id_router]).encode()

        # Determine the id and calculate the marks
        idi = IDrouter_on_path[id_router]
        if id_router == 0:
            idi_1 = "0"
        else:
            idi_1 = IDrouter_on_path[id_router - 1]
        marksrc, _ = calculate_marki(CSTpkt, idi_1, idi, AUTH)

        # Calculate markr using XOR operation as described
        markpkt, init_hash = init_markpkt(CSTpkt)
        markr= full_markr(init_hash, CSTpkt, IDrouter_on_path, AUTH
            ) #Full calculation of markr
```

```python

        # Create and send the packet
        packet = create_packet(seq_count, epoch % 256, int(
            IDrouter_on_path[id_router]), marksrc, markr, markpkt)
        packets_sent.append(packet)
        send(packet)
        packet.show()

        # Decrement the sequence count
        seq_count -= 1

print("Packets sent successfully.")
```

B.3 router.py

```python
from scapy.all import *
from scapy.layers.inet6 import IPv6, IPv6ExtHdrSegmentRouting,
    IPv6ExtHdrSegmentRoutingTLVHMAC
import hmac
import hashlib
from netfilterqueue import NetfilterQueue
from maskHeader import *

router_id = "1"

def process_packet(pkt):
    scapy_pkt = IPv6(pkt.get_payload())
    if scapy_pkt.haslayer("maskh"):
        mask_header = scapy_pkt[maskh]
        pktPath = scapy_pkt[IPv6ExtHdrSegmentRouting].addresses
        hmac = scapy_pkt[IPv6ExtHdrSegmentRoutingTLVHMAC].hmac

```

```
        CSTPath = ""
        for i in range (len(pktPath)):
            CSTPath += pktPath[i]
        CSTpkt = (CSTPath + str(mask_header.epoch) + str(
           mask_header.seq) + str (mask_header.id)).encode()
        idi = str(mask_header.id)

        if idi == router_id:

            idi_1 = "0" #hardcoded because first router on the path
            idi = str(mask_header.id)

            # Calculate marksrc
            expected_marksrc, _ = calculate_marki(CSTpkt, idi_1,
               idi, hmac)

            # Verify the marksrc
            if mask_header.marksrc != expected_marksrc:
                print("Source verification failed, dropping packet.
                   ")
                pkt.drop()
                return
            elif (mask_header.marksrc == expected_marksrc):
                print ("marksrc validated")
        else:
            idi_1 = "0"

        _, expected_markri = calculate_marki(CSTpkt, idi_1,
           router_id, bytes(AUTH))

        # Update markr
        new_markr = mask_header.markr ^ expected_markri

```

```
        # Update the packet with new markr
        scapy_pkt[maskh].markr = new_markr
        pkt.set_payload(bytes(scapy_pkt))

        # Forward the packet to the next hop
        print("Packet forwarded with updated markr:", new_markr)
        print ("-------------------------------------NEXT packet
            -----------------------------------------")
        pkt.accept()
    else:
        print ("no maskh found")
        scapy_pkt.show()

# Set up NetfilterQueue to use the packet processing function
def main():
    nfqueue = NetfilterQueue()
    nfqueue.bind(0, process_packet)

    try:
        nfqueue.run()
    except KeyboardInterrupt:
        pass
    finally:
        nfqueue.unbind()

if __name__ == '__main__':
    main()
```

B.4 receive.py

```
from scapy.all import *
```

```python
from scapy.layers.inet6 import IPv6, IPv6ExtHdrSegmentRouting,
    IPv6ExtHdrSegmentRoutingTLVHMAC
import hmac
import hashlib
from netfilterqueue import NetfilterQueue
from maskHeader import *

sCounter = {}
epoch_counter={}
# Algorithm 2: Procedure at Destination Host
def destination_host(pkt):
    # Extract MASK header
    scapy_pkt = IPv6(pkt.get_payload())
    if scapy_pkt.haslayer("maskh"):
        mask_header = scapy_pkt[maskh]
        epoch = scapy_pkt[maskh].epoch
        router_id = mask_header.id

        # Calculate CSTpkt
        CSTpkt = (PATH[0] + PATH[-1] + str(mask_header.epoch) + str
            (mask_header.seq) + str(mask_header.id)).encode()

        # Calculate markpkt
        expected_markpkt = calculate_markpkt(CSTpkt, mask_header.
            markr)

        # Verify markpkt
        if mask_header.markpkt != expected_markpkt:
            print("Packet verification failed, dropping packet.
                Expected: " + str(mask_header.markpkt) + " but got:
                " + str(expected_markpkt))
            pkt.drop()

```

```
        # Initialize the sCounter if necessary
        if epoch not in sCounter:
            sCounter[epoch] = {}
            epoch_counter[epoch] = 0  # Initialize the epoch counter

        if router_id not in sCounter[epoch]:
            sCounter[epoch][router_id] = 0

        # Increment the router-specific counter and the epoch counter
        sCounter[epoch][router_id] += 1
        epoch_counter[epoch] += 1

        # Check if the epoch counter has reached the threshold defined in CSH_M
        if epoch_counter[epoch] >= CSH_M[epoch]:
            for router in sCounter[epoch]:
                # Calculate the expected number of packets for this router
                N = CSH_M[epoch] / len(PATH)  # Assumed default policy
                print("N is:"+ str(N)+ " sCounter" + str(sCounter))
                # Check if the actual count is within 5% of the expected count
                if not (N * 0.95 <= sCounter[epoch][router] <= N * 1.05):
                    print(f"Verification failed at router {router} in epoch {epoch}")
                else:
                    print(f"Verification succeeded for router {router} in epoch {epoch}")
            # Clear the counters for the epoch
```

```
                sCounter[epoch] = {}
                epoch_counter[epoch] = 0

        print("Packet processed successfully at destination.")

    else:
        print("no mask header found")
        pkt.show()



# Set up NetfilterQueue to use the packet processing function
def main():
    nfqueue = NetfilterQueue()
    nfqueue.bind(2, destination_host)

    try:
        nfqueue.run()
    except KeyboardInterrupt:
        pass
    finally:
        nfqueue.unbind()

if __name__ == '__main__':
    main()
```

B.5 topology.py

```
from ipmininet.iptopo import IPTopo
from ipmininet.srv6 import SRv6Encap, SRv6EndXFunction,
    LocalSIDTable
from ipmininet.ipnet import IPNet
```

```
from ipmininet.cli import IPCLI
from ipmininet.srv6 import enable_srv6
from mininet.node import Controller
from ipmininet.iptopo import IPTopo
from ipmininet.router.config import IP6Tables, RouterConfig, Rule,
    InputFilter, NOT, Deny, Allow

class MyTopology(IPTopo):

    def build(self, *args, **kwargs):

        r1 = self.addRouter("r1")
        r2 = self.addRouter("r2")
        h1 = self.addHost("h1")
        h2 = self.addHost("h2")

        link1 = self.addLink(h1, r1)
        link1[h1].addParams(ip=("fc00:a::1/64"))
        link1[r1].addParams(ip=("fc00:a::2/64"))

        link2 = self.addLink(r2, h2)
        link2[r2].addParams(ip=("fc00:b::3/64"))
        link2[h2].addParams(ip=("fc00:b::4/64"))

        link3 = self.addLink(r1, r2)
        link3[r1].addParams(ip=("fc00:c::1/64"))
        link3[r2].addParams(ip=("fc00:c::2/64"))

        ip6_rules_r1 = [Rule('-I FORWARD -s fc00:a::1 -j NFQUEUE --
            queue-num 0 ')]
        r1.addDaemon(IP6Tables, rules=ip6_rules_r1)

```

```
        ip6_rules_r2 = [Rule('-I FORWARD -s fc00:a::1 -j NFQUEUE --
            queue-num 1 ')]
        r2.addDaemon(IP6Tables, rules=ip6_rules_r2)

        ip6_rules_h2 = [Rule('-I INPUT -s fc00:a::1 -j NFQUEUE --
            queue-num 2 ')]
        h2.addDaemon(IP6Tables, rules=ip6_rules_h2)
        super().build(*args, **kwargs)

    def post_build(self, net):
        for n in net.hosts + net.routers:
            enable_srv6(n)



    def clean(self):
        for table in self.tables.values():
            table.clean()

net = IPNet(topo=MyTopology(), use_v4=False, allocate_IPs=False)
try:
    net.start()
    IPCLI(net)
finally:
    net.stop()
```

B.6 Tamarin_prover.spthy

```
theory source_auth_MASK
begin

builtins: hashing
functions: MAC/2, truncate_32/1
```

```

rule generateHMACkey:
 [Fr(~hmac_key)] --[CreateHMACKeys($S, $R, ~hmac_key)]->[!HMAC_key(
    $S, $R, ~hmac_key)]

rule generateAK:
 [Fr(~AKsi)]
 --[CreateAK($S, $R, ~AKsi)]->
 [!AK($S, $R, ~AKsi)]

rule generateRDK:
 let
  indicator = <~ts, ~uid>
  RDK = MAC(AKsi, <$idi, indicator>)
 in
  [!AK($S, $R, AKsi), Fr(~ts), Fr(~uid)]
  --[CreateRDK($S, $R, $idi, RDK)]->
  [!RDK_Router($R, $idi, RDK), !RDK_Source($S, $idi, RDK)]

rule real_source:
 let
  HMAC = h(<$IPsrc, $IPdst,hmac_key>)
  CSTpkt = <$IPsrc, $PATH, $flag, $seq, $epoch, $idi>
  marki = MAC(RDK, <CSTpkt, $idi_1, $idi, HMAC>)
  marksrc = truncate_32(marki)
 in
  [!RDK_Source($S, $idi, RDK), !HMAC_key($S, $R, hmac_key)]
  --[Source_create_marksrc($S, $idi, RDK, marksrc),
     Unique_RDK_use($S, $idi, RDK),
     Running($S, $R, marksrc)]->
  [Out(<HMAC, $idi_1, $idi, marksrc>)]

rule router:
```

```
 let
  HMAC = h(<$IPsrc, $IPdst,hmac_key>)
  CSTpkt = <$IPsrc, $PATH, $flag, $seq, $epoch, $idi>
  marki = MAC(RDK, <CSTpkt, $idi_1, $idi, HMAC>)
  marksrc_prime = truncate_32(marki)
 in
  [!RDK_Router($R, $idi, RDK), In(<HMAC, $idi_1, $idi, marksrc>), !
     HMAC_key($S, $R, hmac_key)]
  --[Router_validates_source($idi, RDK, marksrc, marksrc_prime),
     Eq(marksrc, marksrc_prime),
     Commit($R, $S, marksrc)]-> []

// Restriction to ensure each RDK is only used once by a source
restriction Unique_RDK_usage:
 "All S idi RDK #i #j. Unique_RDK_use(S, idi, RDK)@i &
    Unique_RDK_use(S, idi, RDK)@j ==> #i = #j"

restriction Equality:
 "All x y #i. Eq(x,y) @i ==> x = y"

// Lemma to ensure an RDK is created before it is used by a source
lemma RDK_created_before_use:
  "All S idi RDK marksrc #i.
  Source_create_marksrc(S, idi, RDK, marksrc)@i
  ==> Ex R #j. CreateRDK(S, R, idi, RDK)@j & #j < #i"

lemma executable:
exists-trace
  "Ex S R idi RDK marksrc hmac_key marksrc_prime #i #j #k #l.
   CreateHMACKeys(S, R, hmac_key) @l &
   CreateRDK(S, R, idi, RDK) @ i &
   Source_create_marksrc(S, idi, RDK, marksrc) @ j &
   Router_validates_source(idi, RDK, marksrc, marksrc_prime) @ k &
```

```
   Eq(marksrc, marksrc_prime) @ k &
   #l < #i & #i < #j & #j < #k"


lemma injectiveagreement: //prove that it's not resilient to replay
    attack
 "All R S marksrc #i.
 Commit(R, S,marksrc) @i
 ==> (Ex #j. Running(S, R, marksrc) @j
 & j < i
 & not (Ex R2 S2 marksrc #i2. Commit(R2, S2, marksrc) @i2
 & not (#i2 = #i)))"

lemma source_authentication_unique:
  "All R S marksrc #i.
    Commit(R, S, marksrc) @ i
    ==> (Ex #j. Running(S, R, marksrc) @ j & j < i)
        & (All S2 #i2. Commit(R, S2, marksrc) @ i2 ==> S = S2)"

end
```