Source meta-information authentication along adaptive network paths for policy enforcement

Lukas Limacher

Master Thesis

Supervised by

Open Systems AG
Thomas Oberhammer
Andreas Jaggi
Stefan Lampart

ETH Zürich
Cristina Basescu

ETH Zürich
Prof. Dr. Adrian Perrig

July 5, 2015

Source meta-information authentication along adaptive network paths for policy enforcement
by Lukas Limacher (lul@open.ch)

This master thesis was under the patronage of

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Network Security Group D-INFK, ETH Zurich

and conducted in the industry at

Open Systems AG
Räffelstrasse 29
CH-8045 Zürich
`https://www.open.ch`

Abstract

Source authentication is an important concept in networking which can be used to construct higher-level secure systems. However, building such systems has proven to be challenging in today's Internet due to the prevalence of IP address spoofing. Proposed solutions attempting to address this problem are not efficient as they (1) need a lot of computational resources to provide source authentication or (2) do not take into account network properties for efficient routing.

First, this master thesis presents an adaptive source authentication scheme which allows participating intermediate network entities to efficiently authenticate the source. The scheme does not require per-source or per-destination state for participating intermediate network entities. Besides source authentication, the scheme provides an adaptive routing mechanism to react to the failure of a single unreliable link and supports multiple path-metric routing optimizations. In particular, the scheme allows participating intermediate network entities to adaptively adjust the path in case of a link failure, i.e., react to a link error in the network while still providing reliable source authentication without introducing expensive overhead.

The second pillar of this master thesis is a prototype which illustrates how a basic set of rules for policy enforcement can provide source authentication. These rules allow the source to send additional meta-information for which the authenticity can be assured. The rules have been implemented as Linux kernel modules for `iptables` and are used to construct a *distributed zone-based firewall*. The idea of a distributed firewall is to enforce the same policy at each entry and exit point. In addition, our scheme allows the definition of *zones* which correspond to aggregate sets of source IP addresses to allow more efficient policy enforcement. Our clock cycle measurements show that the proposed adaptive routing mechanism only introduces constant lightweight overhead at participating intermediate network entities.

Acknowledgements

First of all, I would like to especially thank Prof. Dr. Adrian Perrig, Open Systems AG and Stefan Lampart who gave me the possibility to work on this extremely interesting topic and made this master thesis possible in the first place. Secondly but certainly not less, I would like to sincerely thank my supervisors Cristina Basescu at ETHZ and Thomas Oberhammer, Andreas Jaggi and Stefan Keller at Open Systems AG for their precious and outstanding support, discussions and feedbacks throughout the whole thesis. What is more, I want to especially thank Roman Hoog Antink for his valuable inputs on Linux kernel programing, Kirila Adamova for her great inputs on unreliable links, Ehud Ben-Porat and Elias Raftopoulos for great report inputs and inspiring discussions and James Hulka for his vast support during evaluation. In addition, I want to thank the whole network services team including Maitane Zotes, Severin Amrein and James Guthrie for the amazing support. Finally, my special thanks goes to all people from Open Systems AG not yet mentioned who inspired me and made me really enjoy my time during this master thesis.

Contents

1

Introduction

In this chapter, we state the motivation in Section 1.1, give an introduction to the problem statement in Section 1.2, present the contributions in Section 1.3 and provide background information in Section 1.4.

1.1 Motivation

Source authentication is an important primitive in networking which can be used to build high-level secure systems. However, today's employed infrastructure, e.g. the Internet, does not provide intrinsic source authentication neither within the network nor at endhosts due to IP address spoofing. Source authentication helps to prevent attacks such as IP address spoofing. Systems which rely on the authenticity of the sending address of a network packet need a scheme for source authentication. For instance, we provide an example of a *distributed zone-based firewall* which uses source authentication for policy enforcement below. Consider the case of the simple solution which is to distribute symmetric keys between all n nodes, i.e., we must store keys for all other entities at each entity. To authenticate source information we would simply use a MAC scheme with the distributed keys. However, the problem with this approach is that it is not efficient and scalable, e.g., in terms of storage per node and lookup per node which are in $\mathcal{O}(n)$. A more efficient scheme is needed.

1.2 Problem Statement Introduction

Note that this master thesis consists of two main parts. First of all, an adaptive source and data authentication scheme is developed. The scheme can be applied to various systems to provide a source authentication primitive. Afterwards, a case study is conducted where the new scheme is applied to a system in the Linux kernel environment. Moreover, architecture and design considerations are investigated during the deployment which are specific to the system.

1.2.1 Source Authentication Introduction

In short, *source authentication* is the problem of verifying the authenticity of the sending origin entity of a packet. Assume a source S wants to send a packet to a destination D with source authentication. This means that D and the entities along the path towards D should be able to verify the authenticity of S. The main objective is to develop a source authentication scheme which fulfills the following additional problem requirements: (1) the source authentication scheme should provide source authentication not only for an entity D or for the path between S and D but rather for a set of entities in the network between S and D; (2) the set of entities can authenticate the source for a set of paths between the participating entities. (3) The scheme must allow the set of entities to

change the path adaptively. In particular, it should provide an error recovery in case of a link failure. (4) The adaptive routing and error recovery should take into account different path metrics such that the routing can be optimized for different application which want different *application guarantees*. In general, the solution should fulfill the following properties:

- It must be scalable with the numbers of connections and users, i.e., it should have negligible or no storage and state overhead.
- It must be scalable in path length, i.e., the overheads in terms of computation, latency and state should not depend on path length.
- It must be efficient for nodes, i.e., it should have low computation overhead per node.
- It must be fast for entities and forwarding traffic at entities, i.e., it should have low latency overhead per node.
- It must be robust and secure, i.e., it must fulfill the properties for the specified attacker model.
- It must ensure that the size of the header is fixed once created such that packets do not grow.[1]

An extensive problem description is given in Chapter 3.

1.2.2 Prototype Implementation Introduction

In the following, we describe the problem statement with regard to the prototype implementation. The main task was to apply the adaptive source authentication scheme to implement an extension for an existing system. The system provides policy enforcement with a firewall which is built on a hardened Linux kernel using `iptables` [10]. In particular, it uses the Linux kernel packet filter framework netfilter [12]. In addition, these firewall systems are used in combination over a VPN domain to build a *distributed firewall*. The idea of a distributed firewall is to enforce the same policy on each entry point and to use encryption mechanisms as identifier for an increased security [22]. In particular, rules for policy enforcement can be efficiently defined once for all entry and exit points and deployed on all systems. Of course, additional local exceptional rules per system are still configurable. The goal of the implementation is to extend this concept to a *distributed zone-based firewall* with global scope within the VPN domain. In short, we define a *zone* as a collection of origins which are identified with a zone number. The idea behind this concept is that rules for policy enforcement can be efficiently defined on zones instead of single addresses. In particular, these policies can be defined in global scope such that zone information from other systems within the domain can be used for policy enforcement at entry and exit points. Therefore, the main task is to apply the adaptive source authentication scheme such that policy enforcement on authenticated source origin and meta-information can be performed. We call this concept *source meta-information authentication*. As an additional requirement, no packet with adaptive source authentication header should leave the internal domain network of the distributed firewall. To show the feasibility of the approach, the implementation is performed for an existing distributed firewall system between VPN gateways provided by Open System AG. More information is given below. On the one hand, the goal is to implement a method to provide source authentication in the system. On the other hand, the implementation needs to be carefully designed to make the implementation feasible and efficient in netfilter and the Linux kernel. In particular, the enormous size of the Linux kernel and related components introduces additional design complexity for a working prototype. More information about the implementation is given in Chapter 4. The implementation is evaluated in Chapter 5.

[1] This is to prevent fragmentation of packets in the case that the packet would grow bigger as the current MTU.

1.3 Contributions

We present the main contributions of this master thesis.

- An efficient adaptive source authentication scheme which enables a set of entities in the network between a source S and destination D to independently verify the packet origin. Besides source authentication, the scheme provides an adaptive routing mechanism to react to the failure of a single unreliable link. Moreover, the adaptive routing can optimize for different application which prefer different path metrics. For instance, low latency for VoIP or high throughput for data transfers.
- A robust key exchange protocol for the set of entities which want to apply the adaptive source authentication scheme. Keys are not stored but efficiently derived and therefore require no state.
- Design, implementation and evaluation of the application of the scheme on a distributed firewall system built on the Linux kernel using netfilter `iptables`. In particular, a set of `iptables` rules is provided such that source authentication can be used for policy enforcement.
- An implementation of the key exchange protocol.

1.4 Background Open Systems AG and ETH Zurich

This master thesis was conducted at the *Open Systems AG* headquarter in Zurich in collaboration with the ETH Zurich. Open Systems AG is is a company highly specialized in Internet security for more than 25 years. As part of the Mission Control Services offering, Open Systems AG runs global VPN networks for large and midsize companies, which allow different sites to securely communicate with each other. The VPN gateways run aforementioned hardened Linux system and also protect remote locations with a netfilter `iptables` based firewall. These firewalls are centrally configured by a distributed firewall policy approach. Currently Open Systems operates over 3500 VPN gateways with firewall capabilities in 179 countries. The great amount of knowledge and experience present at Open Systems AG provided valuable inputs for the implementation. In particular, Open Systems AG provided an inspiring development and testing environment for the research and implementation. Last but certainly not least, the research and design phase profited from extremely valuable and inspiring inputs from the Network Security Group, ETH Zurich.

2

Related Work

In this chapter, we will outline related work.

2.1 Source Authentication

First, we outline related work in the direction of the field of *source authentication.*

2.1.1 Lightweight Source Authentication and Path Validation

In [29], Kim et al. describe key exchange protocols and *Origin and Path Trace (OPT)*, protocols for source authentication and path validation. It strongly inspired the developing of the adaptive source authentication scheme and key exchange protocol described in Chapter 3. We focus on source authentication and will outline the relevant information from the paper.

The paper provides a method for source authentication along one fixed network path and a related key exchange protocol. The main properties of OPT are that it is scalable in state, i.e., there is no per-flow or per-source state at the intermediate nodes. In addition, it is scalable in path length (more expensive at destination). It is lightweight, i.e., the effort for intermediate nodes for source authentication is constant as shown by the prototype implementation. In addition, DRKey, a key exchange protocol, is introduced which is run before OPT to set up keys between the entities on the path which together form a session. The cost for the DRKey protocol are amortized over an OPT session. OPT solves the source authentication problem for a network path known a priori and authenticates the source along one path. The path is defined generically with nodes. One of the main benefits of the protocol is that the keys of a session at each intermediate node are derived on the fly which means that this is stateless for intermediate nodes. The source sets up a hash value of the payload to be authenticated, a sessionID which identifies the session and is used by each intermediate node to derive the key for the specific session. To provide source authentication, the source sets up for each intermediate node an Origin Verification field which is a MAC of the data hash using the key of each intermediate node. Afterwards, the source sends the packet to the first node on the path. The first node can then apply the MAC to the sessionID using its local secret which will yield the key for this specific session. The node applies the MAC to the payload hash using its derived session key and compares the value to the one in the header. If the value is the same, the node forwards the packet to the next node on the path, otherwise, it will drop the packet. At the end, the destination recomputes the payload hash to prevent spoofing. The data hash field has been added to the header for efficiency reason as then the intermediate nodes do not need to compute the hash over the data to be authenticated again. Note that the security properties of OPT have been formally verified in [48].

In the following, we will briefly outline the design goal differences of OPT to our proposed adaptive source authentication scheme. First of all, our proposed scheme needs an adaptive path which uses a

path set instead of one fixed path to provide an adaptive routing mechanism to react to the failure of a single unreliable link. In addition, the proposed scheme should support optimization of multiple path-metrics for application guarantees. The reason for this is that it enables us to provide error recovery in case that a link breaks. For more details see Chapter 3.

2.1.2 Verifying and enforcing network paths with ICING

In [37], Naous et al. describe a new network primitive called Path Verification Mechanism (PVM). In particular, they work studies how to enforce policies as opposed to express policies. In more detail, a PVM is a protocol and mechanism to send a packet to its next hop, i.e., forwarding. The main goals of ICING are to provide the following two properties provided by a PVM provides.

Path Consent Each entity on the path must consent to the whole path based on its own policy before the actual communication starts.

Path Compliance For each received packet each participating entity can verify that (1) it has previously approved the packets path and (2) all previous entities have forwarded the packet according to the path.

Roughly, ICING uses chained MAC values of the packet content which are used to provide authenticity of the packet. However, note that ICING is more heavyweight than other source authentication schemes such as OPT [29]. For instance, ICING routers need to set up keys between each pair of entities on the path, whereas in OPT the intermediate nodes need to setup only two keys, one each with the source and destination. In addition, these keys need not to be stored at the intermediate nodes but can be derived [29, 48]. We do not discuss the paper in more detail since its problem of interest is path enforcement for one fixed path which is not the main focus of this thesis.

2.1.3 Passport: Secure and Adoptable Source Authentication

In [32], Liu et al. present *Passport* which allows each autonomous system (AS) along the network path the verification of the source address. To this end, Passport uses symmetric cryptography to place tokens on packets. The background setting of the paper is source address spoofing, i.e., the forging of source addresses, e.g., in IP headers. For instance, this can be used to perform a reflection attack in which an attacker sends a request to an entity that spoofs the address of a victim in the IP Header source field. This can yield powerful *Denial-of-Service (DoS)* attacks. The problem would be solved with source authentication which makes source addresses authentic. The papers states that the problem has previously been addressed with two different extreme approaches. The first one is ingress filtering in which AS filter the traffic for spoofed addresses since the address range is known. It provides only limited security [32]. On the other hand, the second approach is to use cryptography based authentication to verify the source address. For instance, the packet can contain a digital signature which has been signed with the private key of the source. The intermediate nodes can then verify this signature. However, this approach has the heavy drawback that it requires a public key infrastructure (PKI) per-source and verifying digital signatures by each intermediate entity which is slow.

The focus of the paper is to investigate whether a source authentication scheme which is lightweight and yet still secure is possible. Note that Passport offers weaker security guarantees than OPT since Passport provides only source AS authentication and source and data spoofing attacks are possible [29].

2.1.4 Other schemes

The following related works do not offer direct source authentication but address a related problems such as to limit effects of IP address spoofing.

Advanced and Authenticated Marking Schemes for IP Traceback

In [45] Song et al. present two new schemes for the *IP Traceback* problem. In short, the IP Traceback Problem is to determine the true source of spoofed IP Packets, e.g., as created for a Denial-of-Service attack as explained before in Section 2.1.3. The paper proposes an efficient IP marking approach where routers probabilistically write partial path information to the packet during forwarding. Roughly, if enough packets are gathered, the attack path can be reconstructed. Unfortunately, this method is not suitable for Source Authentication with our assumptions and goals and has been designed to solve a different problem, respectively [29].

Practical Network Support for IP Traceback

In [42], Savage et al. describe a mechanism for IP Traceback based on probabilistic packet marking. The main idea is the same as in [45] and therefore not suitable for Source Authentication for the same reasons as above.

Hash-Based IP Traceback

In [44], Snoeren et al. present a hashed-based technique for IP Traceback. The main differences to the marking approaches seen above is that with this scheme it is possible to trace even one single IP packet. In short, the invariant portion of the IP header as well as the first 8 bytes of the payload are used to compute the 32-bit packet hash which uniquely identifies almost all differing packets. In addition, the information is stored space-efficiently using a bloom filter. For more information please consult [44]. Unfortunately, this method violates, e.g., our assumption of having no per-source state at the intermediate routers and is therefore not suitable for Source Authentication with our assumptions and goals.

SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks

In [46], Yaar et al. present SIFF, a Stateless Internet Flow Filter which allows an end-host to mitigate DDoS attacks by stopping individual flows. In short, this is achieved using a network capability mechanism, i.e., divide the network in two classes: Privileged and unprivileged. The privileged packets of privileged channels carry capabilities which are verified piecewise by the routers. Unfortunately, network capability mechanisms are not suitable for Source Authentication with our assumptions and goals since the capability can be copied and inserted by the last AS [29].

Multicast Schemes for Source Authentication

We will not discuss schemes for multicast settings as our design goals and assumptions differ from the settings in multicast schemes such as in [38].

2.2 Adaptive Routing and Application Guarantees

In this section, we outline related work with regard to adaptive routing and application guarantees. On the one hand, we present work which supports the idea of an adaptive routing mechanism to react to a failure of an unreliable link. On the other hand, we present work which supports the idea of optimizing for multiple path set metrics to provide application guarantees.

2.2.1 Achieving Convergence-Free Routing using Failure-Carrying Packets

In [30], Lakshminarayanan et al. propose the technique Failure-Carrying Packets (FCP) which is a routing paradigm without a convergence process. In this scheme, each participating entity is aware of

the network. The main idea is that each entity in the network needs to know the list of failed links in the network to compute the path for the packet. In more detail, FCP adds failed links in the header of the packets, such that each participating entity can compute the path and the next hop.

The scheme is related to our proposed adaptive source authentication scheme as it can change the path adaptively with regard to failed links. However, note that the approach is different from ours since our scheme uses the information for a failed link locally available at each entity while in FCP the packet header carries a list of currently failed links so far. In particular, our proposed scheme does not add information which increases the header during routing. Moreover, in our proposed adaptive source authentication scheme the source provides support information to the intermediate entities such that they do not need to recompute the path and know which entities participate in the scheme. In addition, it enables efficient source authentication without reestablishing an additional session. For more details see Chapter 3.

2.2.2 Consensus Routing: The Internet as a Distributed System

In [28], John et al. introduce *consensus routing* which provides two modes for packet delivery: (1) a *stable* mode in which a new route is only adopted after all participating entities have consent on it and (2) a *transient* mode in which packets get heuristically forwarded if they occur packet loss. In particular, it allows a packet to be routed in the stable mode and switched to the transient mode in case of a link failure. It also allows coexistence of different transient schemes. Note that this scheme is not usable for our problem since we need to know the participating entities before and the intermediate participating entities must be able to infer other participating entities for the adaptive source authentication.

2.2.3 Detour: a Case for Informed Internet Routing and Transport

In [21], Savage et al. investigate the inefficiencies in the Internet. They argue that the Internet performance can be improved by more intelligent active routing at certain key points. In addition, Detour is presented which attempts to improve Internet performance by more efficient routes and by accumulating connection information. What is more, they argue that there is the opportunity to specialize routing for different *service classes*, e.g., minimum latency which supports our approach for application guarantees as described in Chapter 3.

2.2.4 The End-to-End Effects of Internet Path Selection

In [41], Savage et al. investigate the End-to-End performance of a packet sent though the Internet on its default selected path and investigate the potential possible improvement for alternative paths. Their investigation shows that in 30% to 80% cases there is a better alternative path. The measurements have been performed for five distinct *path qualities* such as latency, loss rate and bandwidth. Note that these findings support our approach for multiple path metric optimizations as described in Chapter 3.

3

Adaptive Source Authentication Method

In this chapter, we will outline the development of the *adaptive source authentication method*. We start with the problem statement in Section 3.1. Afterwards, we introduce the notation in Section 3.2, the attacker model in Section 3.3 and present the design overview in Section 3.4. We continue to state the assumptions in Section 3.5. The components of the scheme follow which are the k-shortest d-path set in Section 3.6.1, the key exchange protocol in Section 3.7 and the path indicator in Section 3.9. The adaptive routing scheme is given in Section 3.10. Finally, an example of the method and its components is given in Section 3.12.

3.1 Problem Description

In this section, we describe the problem statements and main questions which guided the further analysis.

3.1.1 Problem Statement Overview

We outline the two main *problem statements* and explain them in more detail below.

On the one hand, we have a source S which wants to send authenticated data to a destination D over a selected set of nodes, called *path set*, between them. We call the nodes of a path set *path set entities*. These path set entities can also authenticate the source and data. For instance, these nodes can be different *Autonomous Systems (AS)* or simply routers within a network. On the other hand, we have applications which want to use the data authentication and want additional guarantees. For instance, an application such as `Skype` benefits from low latency in the network and therefore prefers paths with low latency. On the contrary, another application such as an FTP client benefits from high throughput and therefor prefers paths with high throughput. An overview of the problem statements is given in Figure 3.1. In summary, we have the following two main questions or problem statements.

Question 3.1. *How can S provide source and data authentication to D and the path set nodes between them (regardless of the application)?*

Question 3.2. *Applications want guarantees, e.g., some prefer low latency paths while others prefer high throughput paths. How can we support multiple different path metrics for the data and source authentication scheme?*

3.1.2 Problem Statement One: Source and Data Authentication

In the following, we will outline the first problem statement. The main task is that S wants to send to D with source meta-information authentication and achieve the properties as described in Chapter 1.

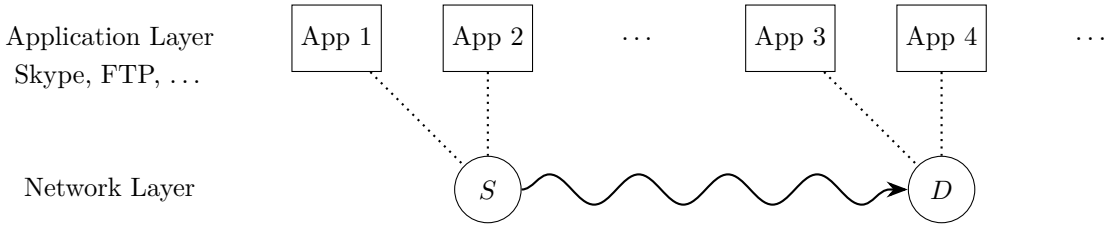


Figure 3.1: Problem Statements Overview. App i denotes an application i which wants some guarantees and are indicated with a rectangle. The dotted lines represent information flow between the application layer (layer 7) and the network layer (layer 3). S wants to send and authenticate some data to D. Both network entities are indicated with a circle.

We present an example which is used to guide through this section. We consider a simple network with entities S, A, B, D in which S wants to send data authenticated to D. The network is depicted in Figure 3.2. To use source and data authentication, S and the path set entities need to setup the

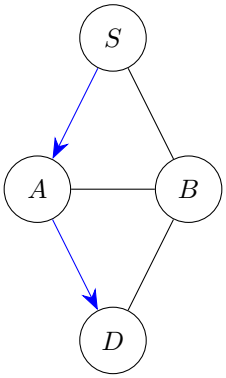


Figure 3.2: An example network diagram for the network with entities $\{S, A, B, D\}$. S sends over A to D indicated by arrows.

cryptographic information needed for the scheme. We call this cryptographic information a *session*. A session is only valid within a certain short time frame. Note that S needs to send to all path set entities to setup a session between them. In this example, assume that S has chosen the path over A to send to D. Assume that S is using this path to send to D and the link between A and D breaks. This setting is depicted in Figure 3.3. Now, S would like to send on a different path over B to D. However, now S would need to create a new session for this new path. We propose to use a path set rather than a simple path to (1) prevent a new session setup and to (2) provide an adaptive routing mechanism to react to the failure of a single unreliable link. In particular, intermediate nodes should be able to adaptively change the path to provide error recovery. To achieve this, S chooses the path set for the session setup accordingly. However, the network path(s) may not be known a priori due to topology changes from failures. This leads to the following question.

Question 3.3. *How should a source S choose the entities to create the path set and how can a path be adapted in case of an error without requiring another session setup?*

3.1.3 Problem Statement Two: Adding Application Guarantees

In the following, we will outline the second problem statement which is an extension of the first problem statement. Applications which use the source and data authentication from S to D want different additional guarantees. For instance, for different applications at S, paths with different

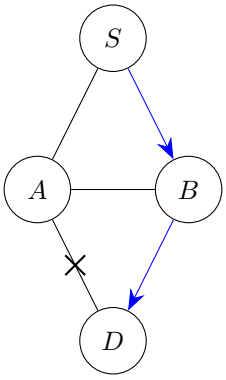


Figure 3.3: An example of a breaking link for the network from Figure 3.2. The broken link is depicted by a cross. S sends now over B to D indicated by arrows.

metrics are optimal. We define the following primary path metrics to find the best path and give an example of applications.

Low Latency Channel with low latency or low delay e.g. VoIP. Primary metric: RTT

High availability Channel with high availability, e.g. for web servers. Primary metric: Packet Loss

High throughput Channel with high throughput, e.g. file transfer. Primary metric: Throughput

We present an example in Figure 3.4 which will be used to guide through this section. Recall that S

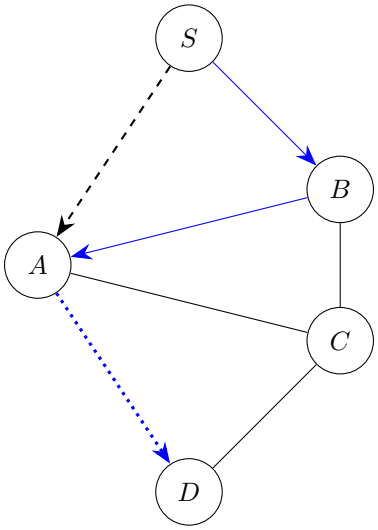


Figure 3.4: An example network diagram for the network with path set entities $\{S, A, B, C, D\}$. The dashed link from S to A is good for latency. The solid links from S to B to A are good for throughput. The dotted link from A to D is good for both latency and throughput.

does not use one path to send to D but rather a path set as mentioned in the first problem statement. For instance, S has chosen entity C as well to be included in the path set, although it has no immediate impact on the paths in the start setting. However, it can be used for new paths in case of error recovery as explained in the first problem statement. In addition, S needs to create the session for the path set entities as well. For instance, S can send on the path as given in Figure 3.5 to setup a session. During the session setup, all path set entities have to be involved. What is more, we have the additional application demands which have to be taken into account when choosing the path set. This leads to the following question.

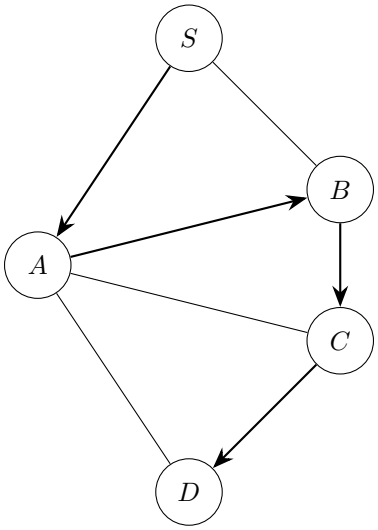


Figure 3.5: An example message to create a session using the path $< S, A, B, C, D >$ for the network diagram in Figure 3.4.

Question 3.4. *Different applications may use different metrics to select paths such as latency, packet loss or throughput. How has the path set and session setup to be created such that optimal paths for each path metric are provided?*

We continue the example with an error recovery case. Assume that the link between A and B fails. In this case, the current paths for each path metric can be adapted. We call such an adapted new path *backup path*. The resulting backup paths are depicted in Figure 3.6. Note that the backup

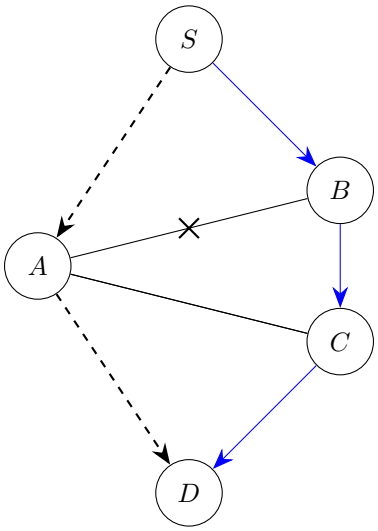


Figure 3.6: Two example backup paths for the network from Figure 3.4 if the link between A and B breaks. The dashed links from S to A to D are good for latency. The solid links from S to B to C to D are good for throughput.

paths can be created only with the path set entities. In addition, the different path metrics for the applications influence the error recovery scheme because for different path metrics different backup paths are optimal for a given path set. This leads us to the following question.

Question 3.5. *How can the optimal paths for each metric be possibly combined and adapted in case*

of an error without requiring another session setup?

3.1.4 Desired Security Properties

In the following, we describe the desired security properties our scheme should achieve. The main security property is to achieve *source authentication* and *data authentication* for a set of entities which provide a set of paths, i.e., adaptive network path. In particular, this means that the path set entities can authenticate the source and the data sent from it, respectively. We later apply this scheme to provide source meta-information authentication.

In the following, we state the desired security properties of each node in our setting.

Authenticate Source and Data The node wants to authenticate the source and the data sent by it.

Traffic Optimization Traffic from different applications should be sent over the link with the specific optimal metric.

Adaptive Routing Information The source does its best to provide good options for paths.

3.1.5 Non-Goals

In the following, we describe the security properties we do not investigate and our scheme does not provide. The main property we do not investigate and do not provide is *path validation*. Roughly, path validation provides a method to validate that a packet has indeed travelled on a specified path. For instance, it is irrelevant for our scheme if a packet has left the path set entities during the routing and later is routed back to a path set entities. Of course, manipulations which affect the desired security properties need still be prevented but path validation specific properties are not provided and investigated. In addition, we do not provide (Distributed) Denial of Service (DDoS) protection, e.g., against packet flooding since we do not guarantee path validation. Another property we do not investigate and provide is packet delivery guarantee as nodes can generally decide if they forward or drop packets. In addition, we provide a partial error recovery mechanism or optimization. However, note that we do not investigate the *first packet problem*, i.e., roughly if the first packet arrived at D. In addition, we do not investigate packet delivery guarantees which would provide error recovery with regard to lost packets, e.g., during the key setup. In particular, our scheme does not retransmit packets to achieve any delivery guarantees.

In the following, we state the non-goals of each node in our setting.

Control previous path A node is not concerned about how the packet has been routed before.

Delivery guarantees A node is not concerned about strict delivery guarantees, i.e., a node cannot guarantee that the packet will arrive in any circumstances

3.2 Notation

In the following, we outline the notation.

We call a node in a given network diagram *entity*. A path set is a a set of entities which can authenticate the source S and the data sent from it for one session. We call the entities in the path set *path set entities* and refer to them simply as *path set*. We use the notation depicted in Table 3.1. It is adapted from [29].

In addition, we use for a path set the common set notation $\{\cdot\cdot\}$ to indicate that the available links between the path set entities can be used in any order. What is more, we use $< \cdot\cdot >$ to denote a fixed sequential path as in [29]. For instance, $< S, A, B, D >$ denotes a sequential path starting at S

Symbol	Definition
Entities	
S	The source, initiating the session
D	The destination
R_x	Node entity in the network
$R_{p.j}$	Path set entity on a specific path p from S to D at position j within this path
Keys	
$\hat{K}_{SD}$	Long-term symmetric key between S and D, valid over multiple sessions
K_X	Symmetric key shared between S, D and entity X for one session
$K_{XY\sigma}$	Symmetric key from X initiated traffic to Y in session σ
SV_X	Local secret value for entity X
$KEYS_\sigma$	Set of shared keys between nodes and S for session σ
$(PK_X,\ PK_X^{-1})$	Public-private key pair of entity X
$(PK_\sigma,\ PK_\sigma^{-1})$	Public-private key pair of a session σ
Information Assets	
P	Network Packet Payload
$PATHSET_\sigma$	Path set information for session σ
$PATH_{i,\sigma}$	Specific path from S to D labeled i using entities R_{ij} as explained above
$INDICATOR_\sigma$	Supportive Routing information for session σ
$INDICATOR_{\sigma,R_x}$	Supportive Routing information for entity R_x for session σ
T_σ	Creation time of session σ
SessionID	Identifier of a session, hash of session σ's public key, path set and session creation time
$AUTH_\sigma$	Authenticated and encrypted session SessionID and private key for session σ
DataHash	Hash of the data to be authenticated
MetaInf	(Optional) application meta-information in the packet header
Position	Label of current active node in the packet header
V_i	Source and data verification field for entity R_i
ID_{R_j}	Representation of entity R_j, e.g., its IP address
$SignKEY_{X,\sigma}$	Signature on a symmetric key for session σ by entity X (using X's private key)
$EncKEY_{X,\sigma}$	Encryption of a symmetric key for session σ by entity X
Cryptographic Functions	
$Sign_{PK_X^{-1}}(\cdot)$	Signature using the private key of entity X
$CheckSig_{PK_X}(\cdot)$	Signature verification using the public key of entity X
$Enc_K(\cdot), Dec_K(\cdot)$	Encryption, decryption using key K
$AuthEnc_K(\cdot)$	Authenticated encryption using key K (and a private key for authentication)
$AuthDec_K(\cdot)$	Authenticated decryption using key K (and a public key for authentication)
$F_K(\cdot)$	Pseudo-random function using key K as seed
$MAC_{K_X}(\cdot)$	*Message Authentication Code (MAC)* Function
$H(\cdot)$	Cryptographic hash function

Table 3.1: The *Notation* used for *Entities*, *Keys*, *Information Assets* and *Cryptographic Functions*.

to A to B to D. On the other hand, $\{S, A, B, D\}$ denotes a path set with the entities S, A, B and D which denotes all possible paths from S to D for the given underlying network. In particular, the path $< S, A, B, D >$ can occur in the setting for this path set, but it might also be different. We give an example for this path set in Figure 3.7 which contains the paths $< S, A, D >$ and $< S, B, D >$. For this network setting, the specific paths (for a session σ) are given by $PATH_{1,\sigma} =< S, A, D >$, $PATH_{2,\sigma} =< S, B, D >$.

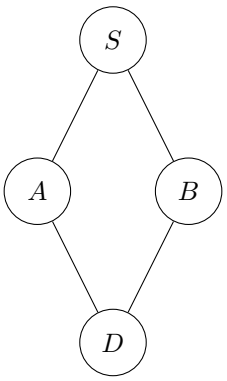


Figure 3.7: Possible network diagram for the path set $\{S, A, B, D\}$.

3.3 Attacker Model

In the following, we will outline the adversary or *attacker model* used. We consider a computationally-bounded network attacker that deviates from the protocol. The attacker has the following possibilities to deviate from the protocol. Note that such deviations can also be created by an error of a node (e.g. hardware error).

Attacker knowledge The attacker knows everything about the scheme and the information apart from the private keys which have not been revealed.

Packet recording An attacker can record an unbounded number of arbitrary packets and use them later for another attack.

Packet alteration An attacker can change packets arbitrarily. In particular, misbehaving nodes can alter any part of the packet.

Packet injection An attacker can inject packets arbitrarily. In particular, misbehaving nodes can create and send arbitrary packets. Note that a packet replay attack is therefore a special case of packet injection.

Denial-of-Service (DoS) attack An attacker tries to disable or severely weaken the system such that it cannot be ordinarily used. To this extent, we consider memory and computation exhaustion attacks on the involved nodes. Note that we do not consider other (Distributed) Denial-of-Service attacks such as packet flooding.

3.4 Design Overview

In the following, we will present the *design overview* of the source and data authentication scheme. The scheme provides source and data authentication per packet which is sent from the source S to a destination D through a set of entities which we call *path set*. In addition, the scheme provides

error recovery for one unreliable link identified between the path set entities. The scheme fulfills the properties mentioned in Chapter 1. In particular, intermediate entities do not need per-source, per-destination or per-flow state for efficient source authentication. The main ideas of the scheme are as follows: (1) the source chooses the path set entities in advance and provides a *path indicator* which gives the intermediate entities supportive information for the adaptive routing. In particular, the path indicator provides alternative paths for error recovery. (2) each path set entity can efficiently authenticate the source and adaptively decide where to send the packet within the bounds of the path indicator. The efficient authentication is ensured as the key for authentication is efficiently derived using only one local secret and an identification value called SessionID which is fast and stateless. (3) the source authentication is performed as an efficient MAC computation. (4) The adaptive routing is performed with the locally available information at each node and the path indicator. The path indicator and scheme are designed to minimize the number of operations needed to determine the next path set entity.

3.5 Assumptions

In the following, we will state the assumptions for the scheme and the key exchange protocol.

First of all, we assume a long-term key is pre-shared between the source S and the destination D. Note that this key can be created, e.g., with the *Diffie-Hellman (DH)* key exchange protocol if there is none. Secondly, we assume all potential sources and destinations know the public keys of the intermediate nodes. This is needed for authentication of the cryptographic material during the session setup in the key exchange protocol. In addition, we assume that the source S is aware of the network topology. In addition, we assume that the source and destination are trusted by the participating entities.

3.6 Unreliable Link Support Scheme

In the following, we will analyze the core part of the scheme in which the path set is determined by the source S. The path set is chosen such that it defends against the most *unreliable link*. We define a link to be unreliable if it fails more or is more often unavailable than the other links on a path. Note that this is orthogonal to the metric used for the link costs. In addition for simplicity, we assume we have one path metric for the link costs. The goal is that S creates a path set such that there is a backup path for the most unreliable link. In case of failure of the unreliable link, the backup path can be used adaptively within the same session.

In the following, we outline a straightforward solution. First of all, we initialize the path set with the shortest path from S to D. Afterwards, we remove the most unreliable link and use the Dijkstra Algorithm from S to D to find the best path without the most unreliable link which gives us the backup path. The additional entities are added to the path set. Note that to keep the path set small, it is desirable that not too many additional entities are added.

In the following, we will analyze this approach further and give examples. The example network is given in Figure 3.8. Note the most unreliable link between B and C. We use an arbitrary path metric for the link costs. From the network diagram we can now infer that E or F are potential backup entities. In addition, we note that there are multiple possible backup paths for each entity, e.g., for F from $< B, F, C, D >$ or $< B, F, D >$. This yields the following questions.

Question 3.6. *Which entity should be chosen as backup entity?*

Question 3.7. *Do we have to choose the exact backup path in advance?*

In the following, we will analyze these problems in more detail. First of all, we argue that if we want the best solution with regard to the path metric, we have to look at all the potential paths

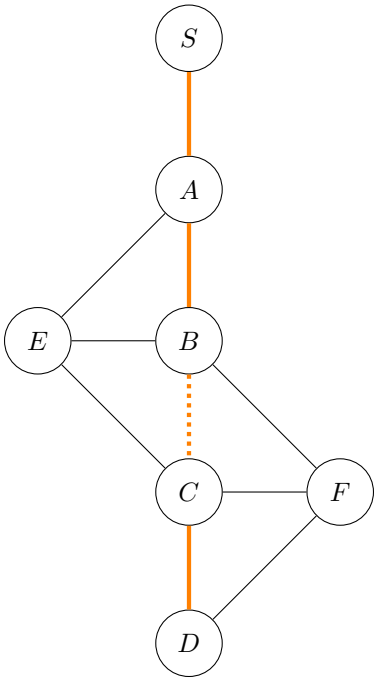


Figure 3.8: An example network diagram with entities S, A, B, C, E, F, D. The thick line represents the shortest path. The dotted part of the shortest path between B and C is the most unreliable link.

available from B to D. In addition, the best backup path in terms of path metric could also go directly from A to D. We give an example with link costs in Figure 3.9. Clearly, if the link B to C fails it is more efficient in terms of link costs to send from A to D on the path $< S, A, E, C, D >$. Note that not all links which belonged to the shortest path are necessarily included again. Nevertheless, there are the following options in this situation.

- B could send packets back over A to E. However, this is not desired since we do not want the additional overhead of sending a packet back towards S. In addition, this can create loops in the network.

- B can tell its predecessor A that the path is broken and that it should send over E to C. This is more efficient, especially if the link from B to C is unavailable for a longer time. If the link is available again, B can also tell A again. In short, this means that routing information is sent to A such that it can see when the unreliable link on the path is broken.

- B only sends packets towards D. In addition, it adaptively decides on its own where to send. In this case, it sends packets to E as long as the link to C is down as it is the cheapest link locally available which does not go back towards the source S. If the link to C is back, B will adaptively use the link to C. However, if the link from B to C is down for a longer time, the topology information at S is updated and it can rerun the scheme to use again the optimal path. This option has the advantage that no state is needed and B can always adaptively decide where to send.

We argue that the last step is most efficient, i.e., we always adaptively route towards the destination D with the local available routing information. We will explain later what we exactly mean by routing towards the destination D in Section 3.9.

Another problem to consider is that the number of potential additional path set entities could in worst case be in the order of total network entities n, $\mathcal{O}(n)$. To see this, consider the example network

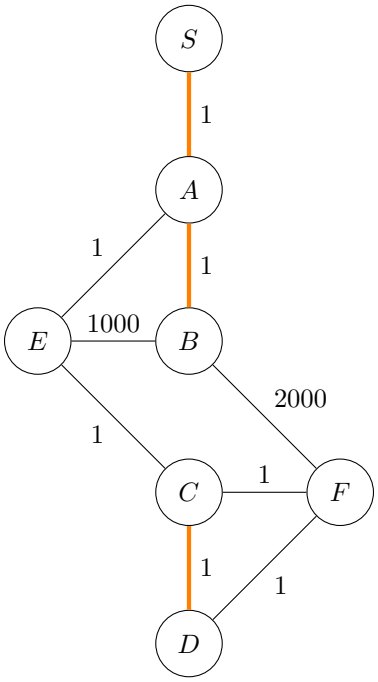


Figure 3.9: An example of link costs such that the backup path from A is the best. It shows the network diagram from Figure 3.8 without the unreliable link.

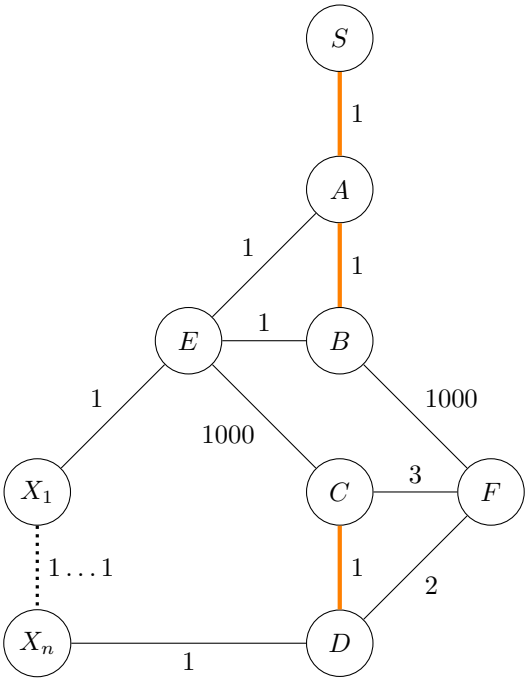


Figure 3.10: An example network diagram with potential huge path set size. It shows the network diagram from Figure 3.8 with additional entities $X_i, i \in \{1, \ldots, n\}, 3 \leq n \leq 999$ and without the unreliable link.

diagram in Figure 3.10. Note that with the simple method described, we remove the unreliable link from B to C and calculate the backup path with Dijkstra as $< S, A, E, X_1, \ldots, X_i, X_n, D >$. Therefore, the final path set size would be 6 for $\{S, A, B, C, E, D\}$ plus n for the entities $X_i, i \in \{1, \ldots, n\}$. In the following, we present options to keep the size of the path set small.

- We can limit the maximum size of the path set and use only the shortest path in case the limit is reached. The reasoning in this option is that if the backup path is too long we omit it.
- We minimize over the number of additional path set entities regardless of the backup path cost. In our example, this would choose E or F as additional path set entities as both backup paths go through a minimum of four entities. This is option minimizes the path set size but provides no guarantees about the cost of the backup path.
- In case the maximum path set size limit is reached for the first backup path with the lowest costs, we investigate if the second best backup path with the second highest link cost would not break the limit etc. Assume the path set size limit is 6 for our example. In this case, E is chosen as backup entity. The reason for this is that the path $< S, A, E, C, D >$ has the second highest total path cost 1003 and a path set size of 6 whereas the shortest backup path results in more than 6 path set entities. This is a tradeoff between backup path cost and number of path set entities.

In the following, we will describe an algorithm which combines the last two options to calculate the backup path and the path set at the source S.

3.6.1 k-shortest d-path set Algorithm

In the following, we present the k-shortest d-path set algorithm which calculates the backup path and the path set at the source S. It makes use of Yen's algorithm, a loopless k-shortest path algorithm [47] [33]. We consider the version of Yen's algorithm in which Dijkstra's algorithm (with fibonacci heap) is used as a shortest path algorithm internally. Roughly, the *k-shortest path* problem is to list the k paths connecting two nodes S and D in a directed graph with minimal total costs. In addition, the loopless k-shortest path requires the paths to be *simple*, i.e. the paths are not allowed to visit the same node twice [5]. As we use an undirected graph we can simply transform it into a directed graph. We combine this algorithm with the idea to minimize over the path set size.

In the following, we describe the k-shortest d-path set Algorithm. S gets the network topology and performs the following steps.

1. Initialize the path set with the entities from the shortest path from S to D.
2. Identify the least reliable link on this shortest path
3. Remove the least reliable link from the network diagram
4. Run the k-shortest path algorithm for S to D
5. Investigate the k-shortest paths up to a total cost threshold of ct and take the path with the least number of additional entities d

Note the parameters k and d which influence the running time and performance of the algorithm, respectively. In addition, note the parameter ct, the threshold of the total cost for each of the k-shortest paths which influences the precision of the algorithm. The ct parameter provides an upper bound on the selected backup path total cost.

We will continue the example from Figure 3.10 as an illustration. If this algorithm is applied for $k = 3$ we get the results as illustrated in Figure 3.11. The algorithm would choose A as backup entity

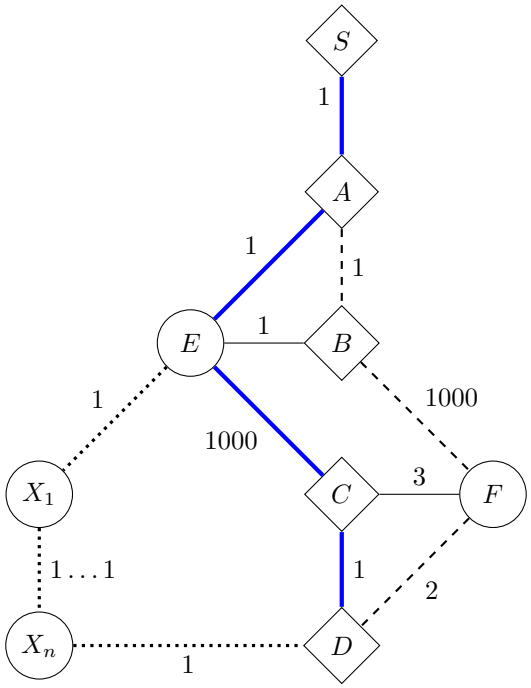


Figure 3.11: An example application of the k-shortest d-path set algorithm in the example network from Figure 3.10. The shortest backup path (S to A to E and then dotted), the second shortest backup path (thick) and the third shortest backup path (S to A and then dashed) are indicated as the algorithm would calculate them. The path set has been initialized with the entities $\{S, A, B, C, D\}$ indicated with diamond shaped nodes.

as it is the first shortest path with only one additional path set entity which is the least possible. However, note that in general, the algorithm needs to consider all k-shortest paths if there is no shortest path with only one additional path set entity.

The time complexity is analyzed in the following. Let l denote the length of the shortest path, E the number of edges in the network and V the number of vertices in the network. For step 1, the complexity is given by the Dijkstra Algorithm with $\mathcal{O}(E + V \cdot log(V))$. For step 2, the complexity is given by l as each link of the shortest path has to be investigated. The step 3, depends on the data structure but assume is at most l. For step 4, the complexity is given by Yen's algorithm which uses Dijkstra's algorithm at most $k \cdot V$ times with $\mathcal{O}(k \cdot V(E + V \cdot log(V))$ [33]. At step 5, we have at most V entities per potential backup path. We minimize over all k backup paths in worst case by looking at each entity in time $\mathcal{O}(k \cdot V)$. In total, the complexity is bounded by $\mathcal{O}(E + V \cdot log(V)) + \mathcal{O}(k \cdot V(E + V \cdot log(V)) + \mathcal{O}(k \cdot V) = \mathcal{O}(k \cdot V(E + V \cdot log(V))$.

3.6.2 Unreliable Link Detection

In the following, we outline how a link can be identified as unreliable. As stated before, note that an unreliable link can still be good in a specific path metric. In short, we want to detect unreliable links with the information from path metrics available. The optimal solution is to have this information available for all links at each potential source entity such that the source node can directly compute and identify the unreliable links. In the following, we present possible metrics to identify unreliable links.

Link Lifetime Low link lifetimes or links which are often unavailable indicate links which are often disrupted for some reasons.

Packet Loss High packet loss could indicate a link which might be unavailable for some short time. For instance, it could be a wireless link which could be temporarily unavailable if there is exceptional strong noise in the environment.

Variance in delay A huge variance in delay can show that a link might be unstable or unreliable.

We suggest to use the link lifetime as a metric for unreliability. In more detail, for each link we calculate a weighted average value which depends on the link's availability. The value is averaged over a specified time period and updated in a certain sampling frequency. The sampling frequency should be chosen depending on the systems update frequency of link lifetimes. Let a_i denote the weighted average at step i, w_o the old value weight, w_n the new value weight, v_o the old value and v_n the new value. v_n is 1 if the link is available during sampling and 0 otherwise. v_o corresponds to the previously computed a_{i-1} then we have $a_i = w_o \cdot v_o + w_n \cdot v_n$. This yields a metric for each link's lifetime reflecting the availability of the link over the specified time period. Given a shortest path from S to D we define the link with the currently lowest a_i value as the most unreliable link. For more information we refer to Section 4.2.4.

3.7 Key Exchange Protocol

In the following, we describe a *key exchange protocol* to create the cryptographic material needed for the source authentication scheme. The protocol is described in Figure 3.12. First, we give a design overview of the protocol. Afterwards, we will explain each step of the protocol in detail.

3.7.1 Protocol Design Overview

Given a source S, a destination D and a path set, S wants to send packets with source and data authentication optimized for specific metric(s) through the path set. The goal of the key exchange protocol is that the necessary cryptographic material is created to enable the source and data authentication. The cryptographic material is created in form of a session for the path set entities. The key exchange protocol generally needs to send multiple packets on different paths since the path set defines a set of multiple possible paths. Note that S has computed the path set before with the k-shortest d-path set algorithm. Assume we consider one unreliable link and one path metric. In this case, the specific paths used for the key setup are the initial shortest path and the backup path which is the k-shortest path selected during the execution of the k-shortest d-path set algorithm. This ensures that each path set entity is visited at least once during the key setup. Note that we do not optimize in case it is possible to send only one packet since the session setup event is rare compared to the adaptive routing. In addition, expensive asymmetric cryptography is used during its execution. First of all, S creates a fresh public private session key pair $(PK_\sigma, PK_\sigma^{-1})$ for the new session σ as well as a session identifier SessionID. The SessionID can be inferred by each protocol participant from forwarded information. The main idea of the key exchange protocol is that each entity locally computes its key based on a local secret and the SessionID and computes a signature of it. Note that this local secret is the same for all sessions. The public session key PK_σ is used in the protocol at each path set entity locally to encrypt its computed key and signature. This encrypted material is accumulatively forwarded to D. D needs the sessions private key to decrypt the accumulated keys. To provide this private session key to D, S computes a key $K_{SD\sigma}$ for S initiated traffic to D and another key $K_{DS\sigma}$ which is used for D initiated traffic to S to send the accumulated keys back. These keys are computed with the longterm key $\hat{K}_{SD}$ pre-shared between S and D. The key for S initiated traffic is used to authentically encrypt the private session key PK_σ^{-1} for D. D waits for a maximum amount of time until all keys have been received, otherwise, it aborts the protocol. If all keys have been received, D uses the session's private key to decrypt the keys and checks the signatures of the

Initialization by Source S

1. Assume long-term symmetric key $\hat{K}_{SD}$ shared with D
2. Assume S computed the path set $PATHSET_\sigma = \{S, R_a, R_b, \ldots, D\}$ and paths $PATH_{i,\sigma}$ to D
3. Initiate new session σ and pick random session key $(PK_\sigma, PK_\sigma^{-1})$
4. Measure current time T_σ
5. Compute SessionID $= H(PK_\sigma \| PATHSET_\sigma \| T_\sigma)$
6. Compute $K_{SD\sigma} = F_{\hat{K}_{SD}}(S \| D \| \text{SessionID})$
   $K_{DS\sigma} = F_{\hat{K}_{SD}}(D \| S \| \text{SessionID})$
7. Compute $AUTH_\sigma = AuthEnc_{K_{SD\sigma}}(\text{SessionID} \| PK_\sigma^{-1})$
8. For each setup path $PATH_{i,\sigma}$, send to the first entity $R_{i.1}$
9. Wait for an answer from D, abort if longer than time T_l

In parallel: $S \to R_{1.1}, S \to R_{2.1}, \ldots$

Path 1 $S \to R_{1.1}$

10. Forward $\{PATH_{1,\sigma}, PK_\sigma, PATHSET_\sigma, T_\sigma, AUTH_\sigma\}$

Pairwise Key Derivation by $R_{1.1}$

11. Compute $K_{R_{1.1}} = F_{SV_{R_{1.1}}}(\text{SessionID})$
12. Encrypt $K_{R_{1.1}}$: $EncKEY_{R_{1.1},\sigma} = Enc_{PK_\sigma}(K_{R_{1.1}})$
    Sign: $SignKEY_{R_{1.1},\sigma} = Sign_{PK_{R_{1.1}}^{-1}}(K_{R_{1.1}} \| PK_\sigma)$

$R_{1.1} \to R_{1.2}$

13. Forward $\{PATH_{1,\sigma}, PK_\sigma, PATHSET_\sigma, T_\sigma, AUTH_\sigma, EncKEY_{R_{1.1},\sigma}, SignKEY_{R_{1.1},\sigma}\}$

Pairwise Key Derivation by $R_{1.2}$

14. Computes $K_{R_{1.2}} = F_{SV_{R_{1.2}}}(\text{SessionID})$
15. Encrypt $K_{R_{1.2}}$: $EncKEY_{R_{1.2},\sigma} = Enc_{PK_\sigma}(K_{R_{1.2}})$
    Sign: $SignKEY_{R_{1.2},\sigma} = Sign_{PK_{R_{1.2}}^{-1}}(K_{R_{1.2}} \| PK_\sigma)$

$R_{1.2} \to D$

16. Forward $\{PATH_{1,\sigma}, PK_\sigma, PATHSET_\sigma, T_\sigma, AUTH_\sigma, EncKEY_{R_{1.1},\sigma}, SignKEY_{R_{1.1},\sigma}, EncKEY_{R_{1.2},\sigma}, SignKEY_{R_{1.2},\sigma}\}$

Path 2 $S \to R_{2.1}$

17. Forward $\{PATH_{2,\sigma}, PK_\sigma, PATHSET_\sigma, T_\sigma, AUTH_\sigma\}$

Pairwise Key Derivation by $R_{2.1} \cdots$

$R_{2.1} \to \cdots, R_{2.2} \to D$

Key Retrieval by Destination D **for each packet sent from** S **to** D

18. Already has $\hat{K}_{SD}$, which is the long-term shared symmetric key with S
19. Compute SessionID $= H(PK_\sigma \| PATHSET_\sigma \| T_\sigma)$ and check the integrity
20. Compute $K_{SD\sigma} = F_{\hat{K}_{SD}}(S \| D \| \text{SessionID})$
    $K_{DS\sigma} = F_{\hat{K}_{SD}}(D \| S \| \text{SessionID})$
21. Decrypt $AUTH_\sigma$ and authenticate PK_σ^{-1}: $AuthDec_{K_{SD\sigma}}(AUTH_\sigma)$
22. Decrypt $K_{R_{1.1}}$, $K_{R_{1.2}}$ and $K_{R_{2.1}}$, .. and check their signatures:
    $Dec_{PK_\sigma^{-1}}(EncKEY_{R_{1.1},\sigma}), CheckSig_{PK_{R_{1.1}}}(SignKEY_{R_{1.1},\sigma})$
    $Dec_{PK_\sigma^{-1}}(EncKEY_{R_{1.2},\sigma}), CheckSig_{PK_{R_{1.2}}}(SignKEY_{R_{1.2},\sigma})$
    $Dec_{PK_\sigma^{-1}}(EncKEY_{R_{2.1},\sigma}), CheckSig_{PK_{R_{2.1}}}(SignKEY_{R_{2.1},\sigma})$
    $K_{R_{1.1}}$, $K_{R_{1.2}}$ and $K_{R_{2.1}}$ become shared symmetric keys between each node and D
23. Check that D has the keys of all entities from $PATHSET_\sigma$, if not, wait for missing key(s) for time T_l
24. If the packets are not received within time T_l, abort
25. Compute $K_D = F_{SV_D}(\text{SessionID})$

$D \to S$

26. Forward authenticated and encrypted shared keys:
    $KEYS_\sigma = AuthEnc_{K_{DS\sigma}}(K_{R_{1.1}} \| K_{R_{1.2}} \| K_{R_{2.1}} \| \ldots \| K_D \| AUTH_\sigma)$

Key Retrieval by Source S

27. Decrypt and authenticate the keys received from D: $AuthDec_{K_{DS\sigma}}(KEYS_\sigma)$
    $K_{R_{1.1}}$, $K_{R_{1.2}}$, $K_{R_{2.1}}$, .. and K_D become shared keys between S and each node $R_{p.j}$ and D.

Figure 3.12: Key exchange protocol for a path set.

keys. If all keys are correct, D adds its own key, authentically encrypts all keys and sends them back to the source S. After the key exchange protocol, S and D share a key for this session with each path set entity.

3.7.2 Detailed Protocol Description

In the following, we illustrate the different stages and explain in detail for each step why it is needed. First of all, step 1 follows from our assumptions mentioned in Section 3.5. At step 2, the path set for the session is created by S which also gives the specific paths $PATH_{i,\sigma}$ to send to D such that all path set entities are visited. At step 3, S creates the new session's public private key pair $(PK_\sigma, PK_\sigma^{-1})$. The session's public key from step 3 is needed later in the protocol by the path set entities to encrypt their key and signature of the key to forward it confidentially and authenticly to D. At step 4, S measures the current time which is needed for the next step. The SessionID computed at step 5 is used to identify the session. The public session key is used to bind the key to the session and to have an input with enough entropy for the hash function. The path set is included to bind it as well to the SessionID such that it cannot be altered. In addition, the timestamp is included to prevent packet injection such as replay attacks. The keys $K_{SD\sigma}$ and $K_{DS\sigma}$ computed at step 6 are used to encrypt the information for D and S for each direction. $K_{SD\sigma}$ is used for S initiated traffic and $K_{DS\sigma}$ for D initiated traffic, respectively. We use these keys for each direction to prevent reflection attacks, e.g., for DoS resources attacks. At step 7, we compute $AUTH_\sigma$ which is needed to send the private session key to D. The SessionID is included to authentically bind the key to the session. At step 8, S sends the information to each first node on each path in parallel as described in the steps below. At step 9, S starts a timer for an abort timeout. This is needed if S does not receive an answer from D, e.g., if D has not received all keys. The step 10 denotes the information S sends to the first node in the first path. The fields have been explained above. Note that $PATH_{1,\sigma}$ defines the first path during protocol execution such that each entity knows where to send next during the key exchange protocol. At steps 11 and 12, $R_{1.1}$ computes, encrypts and signs its key for this session. The signature on its key and the session's public key ensures that an attacker cannot simply substitute old values of the encrypted key and signature. At step 13, $R_{1.1}$ forwards the previous information and its encrypted key and signature to the next node on the path $R_{1.2}$. The step 14 and following for the other nodes are similar. In the end, the last node on each path will forward the accumulated information to D. The step 18 and following are similar to the steps at S. At step 21, D retrieves the private session key which is authenticated if the SessionID matches. At step 22, D decrypts and checks the keys of the nodes from the received packets from all paths. At step 23, D waits for maximal total time T_l if not all packets have been received. At step 24, D aborts if not all packets have been received within time T_l. At step 25, D computes its key. At step 26, D encrypts all shared keys and sends them to S. The encryption is authenticated with the inclusion of $AUTH_\sigma$. Finally at step 27, S decrypts and authenticates the keys. The keys are now shared for this session between S and D separately with each node $R_{p.j}$.

3.8 Path Metrics For Adaptive Routing

In the following, we show the insights which lead to the adaptive routing scheme described later. The main goal is to show that in our scheme the source decides for each packet for which path metric the adaptive routing is performed. However, this is possible within the same session and for the same path set entities. The reason for this is that the mix of best paths using different metrics such as low latency and high throughput is in general not optimal. In other words, a local optimum link does not imply a global optimum link on a path for different metrics. We give an example in Figure 3.13 which is used to analyze the following questions.

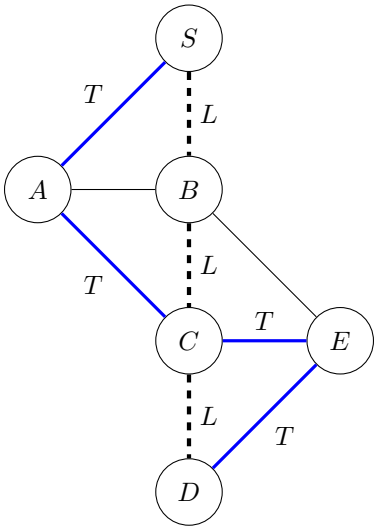


Figure 3.13: An example of two path metrics in a network with path set $\{S, A, B, C, E, D\}$. L denotes a link good for low latency traffic but with bad throughput and T denotes a link with good throughput but with high latency. The thick line is a best path with throughput as metric and the dashed line is a best path with latency as metric.

- Does it makes sense to always specify the paths directly in advance by the source S?
- Should nodes influence each other in sending such that entities change options of the successor nodes?
- Are paths *fixed* after one node has chosen adaptively?

The first observation is that adaptive routing is useful as the path is generally not fixed before. For instance, assume in the example that the link between B and C breaks, B has one link to A or E each for which the metric is unknown and one can be better than the other. In particular, there are multiple options for the new path including $< B, A, C, D >, < B, E, C, D >$ *or* $< B, E, D >$. In addition, the path metrics of the links can change after the source sent the packet. What is more, we argue that the sending node mainly influences the other nodes as it decides which node has to decide next and ensures that there are no loops in routing.

The second observation is that within adaptive routing of one packet, the different path metrics cannot always be mixed. Imagine again that the dashed link between B and C breaks. Then for all possible paths from S to D, at least one link from the throughput path with high latency is used. In particular, we can in general not use links of another optimized metric, here metric throughput for metric low latency, for adaptive routing.

We propose that the source supports the routing of the intermediate nodes with information passed to them. We denote this information scheme with *path indicator* and explain it more in the following. Note that this encourages the idea of simple intermediate nodes.

3.9 Path Indicator

In the following, we describe the idea of a *path indicator*. For simplicity, we first consider the case where we have only one path metric. The main idea is to indicate to a node locally which links are preferred and which ones have to be avoided. In particular, it should provide the following information.

- Indicate to a node what nodes to never forward to

- Indicate to a node what nodes are good to forward to

For instance, situations in which the packet would be routed in a loop must be prevented. In addition, we want the traffic to be routed towards D on the most efficient links available. In particular, the path indicator describes directed links between the path set entities. The path indicator consists of the following three parts.

- *Shortest path* from S to D as calculated during the k-shortest d-path set algorithm. The links are directed form S to D.
- *Backup path* from S to D as calculated during the k-shortest d-path set algorithm. The links are directed form S to D. Note that the backup path usually shares links and entities with the shortest path.
- *Support links* which are directed links added at the node where the unreliable link begins. The links are added according to the following rule: We only look at potential support links from the entity on the shortest path where the unreliable link begins. Check each link at this node if it connects to an entity on the backup path after the fork of the shortest path and the backup path. If there is such a link, we call it support link and add this link to the path indicator. The direction of such a support link is always from the shortest path towards the backup path. If there are multiple such support links and the path indicator size should be limited per entity, keep the links which connect as close as possible to the destination D along the backup path. The reason for this is that the maximal number of support links is limited by the maximum number of encoded successors in the path indicator in the header. The complexity is bound by the number of links at the node where the unreliable link starts. Note that this approach of support links further supports the idea of backing up the unreliable link.

It is desirable that the default main path to be used is the shortest path from S to D as used for the path set initialization. Note that the *global* shortest path from S to D is not necessarily the *local* optimal link at an intermediate node. However, if one of the links on the shortest path breaks, we want to adaptively use another link which still routes the packet towards D. The idea is that a path set entity locally makes the best decision possible with the *adaptive routing information* about the links it has available. This corresponds to the problem we have seen before in Section 3.6 in Figure 3.9. The entity before the unreliable link tells its predecessor, that the link is down. If the adaptive routing information reaches the node, it can then use the link of the backup path. In general, this information can be distributed with a gossip protocol, such that each entity of the path set has with a high probability the correct information about whether the unreliable link is available. In addition, the support links provide additional links to bypass the unreliable link in case it is not available. However, note that these links do not provide bounds on the path metric themselves. However, they connect the node which is most likely to have a link breaking with the backup path again such that the packet can be routed adaptively and a new session setup is not needed immediately. Note that the size of the path indicator is analyzed in Section 3.10.2.

To indicate to a node which links are preferred we label the links with a *priority number*. Therefore, a node sees immediately in which order the available links should be taken. In case of a failure of a link with first priority, it can take the link with second priority and so on. We denote the priorities with roman numbers, I is the first priority for the best path, II is the second best and so on. Links with no priority indicate that at least one end of the link is no path set entity. Therefore, such a link is never assigned any priority by S and is not part of the path indicator. We define the priorities for each link as described in the following. For the best priority I, we take the original shortest path from the path set initialization. All following worse priorities $II, III, \ldots$ are defined by links of the backup path and the support links. The links of the backup path are labeled with priority II. Note that a link can belong to multiple paths. In this case, the link is still labeled with the best priority.

Finally, we define a priority for each support link. We label the support link which connects closest to D with priority III, the one second closest to D with priority IV. This choice of priorities ensures that support links are only taken if all other links are not available.

3.9.1 Path Indicator Example

In the following, we show an example of a path indicator. We continue the example we have seen for the k-shortest d-path set algorithm in Figure 3.11. We show the result after the execution of the algorithm and label the links between the path set entities with priorities. The resulting link prioritization is depicted in Figure 3.14. Now, the path indicator is created according to the rules described before in Section 3.9. The resulting path indicator is depicted in Figure 3.15.

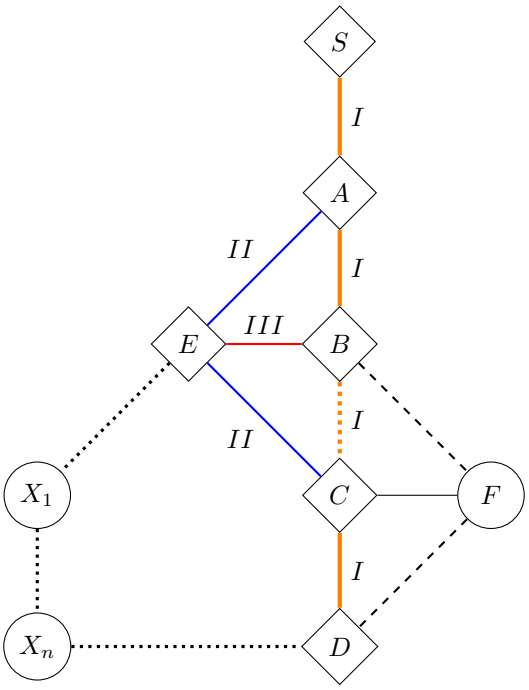


Figure 3.14: An example for link priorities for the network from Figure 3.11. Links of the shortest path have priority I, links of the backup path have priority II and potential support links have priority III. The path set is $\{S, A, B, C, D\}$ indicated with diamond shaped nodes.

3.9.2 Encoding of a Path Indicator

In the following, we will present an encoding for a path indicator. We can state for each node in the path indicator its successors. We denote the separator between the node and its successor with $\succ$ and the separator before the next node with $|$. In addition, we denote with 0 the empty successor. For instance, for the example in Figure 3.15 we can write the partial ordering with $S \succ A|A \succ EB|E \succ C|B \succ EC|C \succ D|D \succ 0$. First, we note that the separator $\succ$ is not needed for the encoding since the predecessor is always the first node between two $|$. In addition, it is possible to first label the nodes and use the labels to define the order relation. Note that we order the path set entities such that the labeling is implicitly defined by the position within the path set. For visibility, we write the index of an element as its power. For instance, the same partial ordering as before with labels is given by $S^1A^2B^3C^4E^5D^6\|12|253|54|354|46|60$. Note that the symbol $\|$ is not needed if the number of path set entities is known. Moreover, note that we can state the successors for each node in the order of the labeling, i.e., we do not need to mention the first node at all. Therefore, our example labeling

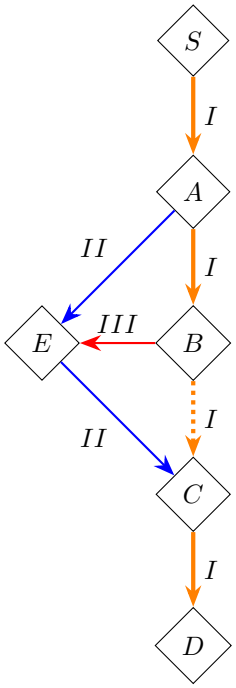


Figure 3.15: The resulting path indicator from the network in Figure 3.11. Note that all links are directed and the support link is directed from the shortest path towards the backup path.

simplifies to $S^1A^2B^3C^4E^5D^6\|2|53|54|6|4|0$. We argue that the encoding with implicit labeling is best suited for the scheme as it is the most compact one.

Finally, we include the priorities in the encoding. We simply order the successors of a node by priorities and can therefore directly infer the priorities. For instance, $A \succ BE|\ldots$ which is identical to $ABE|\ldots$ denotes a node A with successor B with priority I and successor E with priority II. Note that this implies that each node has a link with local priority of I and so on which does not reflect the global view. However, this is not a problem since only the order in which the links are locally chosen is important and this order is preserved. The final encoding of the example path indicator is therefore $S^1A^2B^3C^4E^5D^6\|2|35|45|6|4$.

3.9.3 Path Indicator Loop Free Property

In the following, we prove that the path indicator cannot contain a loop. We prove this by breaking the path indicator into its three parts and show that the sequential combination of the parts does not contain any loops. Recall that the three parts are shortest path, backup path and support links as illustrated in Figure 3.16.

First, we prove the statement 3.1. It will be used to prove that adding the backup path and the support links to the shortest path does not introduce any loops in the path indicator. Note that for the proof the statement 3.1 is only needed if at least one backup path exists. Therefore, in the following we assume that there exists at least one backup path for the unreliable link on the shortest path.

Statement 3.1. *The backup path intersects the shortest path exactly two times, i.e., it leaves the shortest path along one link and merges later with the shortest path again along another link. In other words, it diverges exactly once from the shortest path.*

One divergence of the backup path from the shortest path is defined by the collection of links forming the part of the backup path which leave the shortest path until the backup path merges again

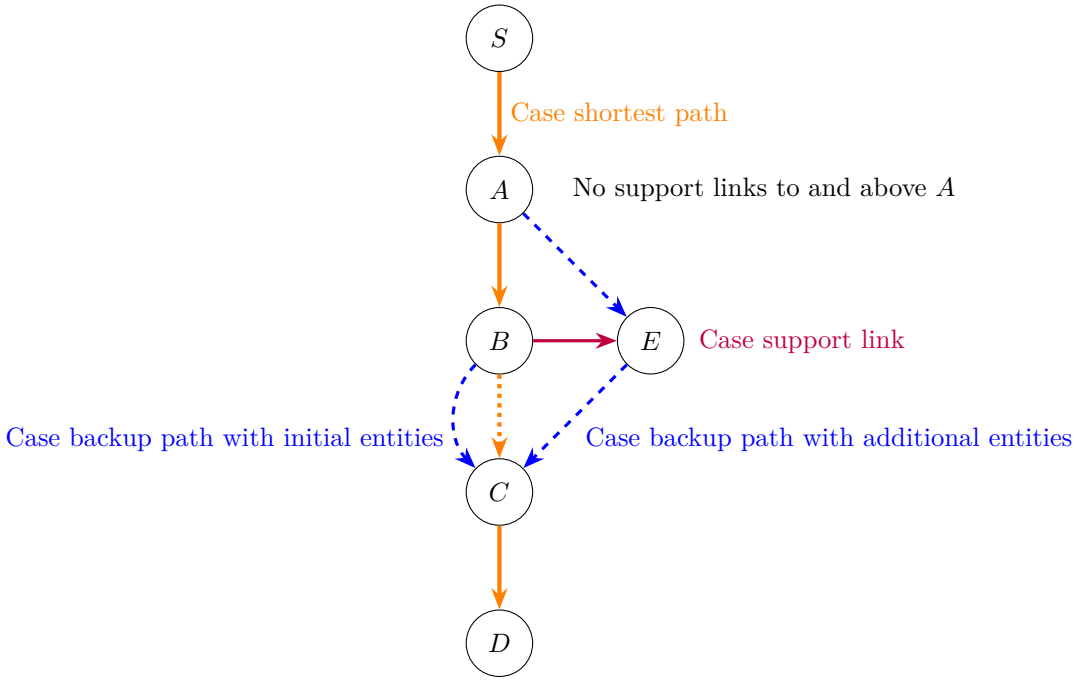


Figure 3.16: The different cases illustrated for path set entities $\{S, A, B, C, E, D\}$. The unreliable link on the shortest path is the dotted link from B to C. Note that a backup path reuses part of the shortest path.

with the shortest path. Therefore, the number of divergences of a backup path from the shortest path are the number of such divergences, i.e., the number of times the backup path forks from the shortest path on a link which is not part of the shortest path. The statement says that a backup path from our k-shortest d-path set algorithm can only consist of one such divergence. As an example, we have the backup path from entity A over entity E to entity C in Figure 3.16. The idea behind this statement is that once we are on the shortest path there is no other option for any better path since the links of the shorter path are by definition optimal in terms of costs. In particular, once we are on the shortest path again after the unreliable link, we will not leave it since it is the optimal path towards D.

We prove the statement by assuming that it is not true. We do not need a case distinction on whether there are additional backup entities or not since the statement is only concerned about the number of intersections of the backup path with the shortest path. However, we make a case distinction on the three different kind of loops which can occur within the backup path with regard to the position of the unreliable link: (1) the loop is before the unreliable link, (2) the loop bypasses the unreliable link and (3) the loop is after the unreliable link. Note that one directed link between two entities on the shortest path cannot intersect the shortest path twice since there is only one link. Therefore, in the following we assume that the backup path consists of at least two links and contains a loop, i.e., intersects the backup path more than once.

First of all, we start with the case (1) where there is a loop in the backup path before the unreliable link. An example of such an outcome of the k-shortest d-path set algorithm is depicted in Figure 3.17. Assume we have a backup path which contains a loop before the unreliable link, e.g., the path in Figure 3.17. Without loss of generality, consider the first part of the path which creates a loop in Figure 3.17, i.e., the links S to B to X to A. These links do not contribute to bypass the unreliable link, i.e., these are only chosen by the k-shortest d-path set algorithm if the costs and the number of additional backup path set entities is minimized. In this example, the afore mentioned links have been chosen to reach A even though there is an initial shortest path to A. Therefore, the links S to

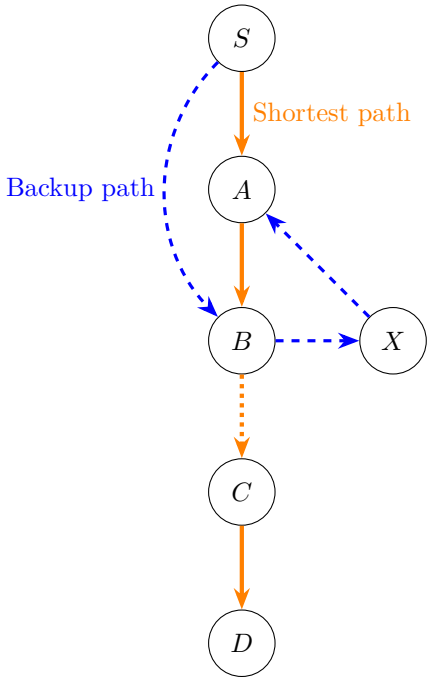


Figure 3.17: Case of multiple intersections of the shortest path by the backup path before the unreliable link. The unreliable link is indicated by the dotted link from B to C.

B to X to A would constitute a new shortest path to the entity A and contradict the fact that the link from S to A is the shortest path. Therefore, the part of the backup path with a loop before the unreliable link yields a contradiction and is not possible.

We prove the second case (2) where there is a loop in the backup path which bypasses the unreliable link within the loop. For instance the path in Figure 3.18. Such a loop implies that (a) the path set algorithm has chosen a path which bypasses the unreliable link more than once and (b) has chosen a path after the unreliable link different than the shortest path. However, this contradicts the fact of the k-shortest d-path set algorithm that minimizes over the number of additional path set entities and the costs of the path for (a) and the fact that the shortest path provides the links with the lowest cost closer towards the destination D for (b) (as in case (1)). Therefore, the part of the backup path with a loop which bypasses the unreliable link yields a contradiction and is not possible.

The contradiction in the third case (3) follows from the same reasons as in case (1).

Therefore, we showed that there is a contradiction for all paths which diverge more than once from the shortest path for all possible locations of the unreliable link. Therefore, we have shown that the backup path diverges *at most* once from the shortest path. In addition, by definition the k-shortest d-path set algorithm will choose at least one additional backup path to bypass the unreliable link if one is available. Therefore, the statement 3.1 must hold.

In the following, we will show that the path indicator cannot have loops. First of all, we only look at the shortest path. Clearly, there are no loops since it is only one directed path. Secondly, we add the backup path entities and its backup path. We see that the backup path is a directed link from S to D which intersects the shortest path exactly two times, i.e., it diverges exactly one time from the shortest path. In particular, it will be connected to the shortest path on two nodes, one before and one after the unreliable link. This follows directly from the statement 3.1 and the fact that the k-shortest d-path set algorithm only considers paths from S to D which guarantees the direction of the links towards D. Therefore, the backup path in combination with the shortest path does not introduce any loops. Finally, we have the support links. All support links will only start at the node

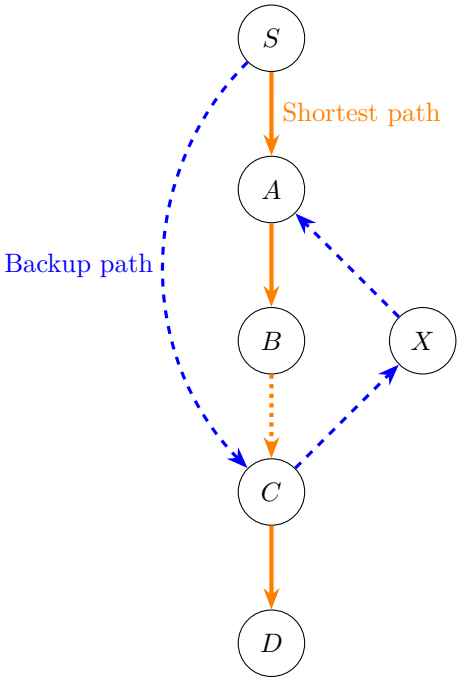


Figure 3.18: Case of multiple intersections of the shortest path by the backup path which bypasses the unreliable link within the loop. The unreliable link is indicated by the dotted link from B to C.

where the unreliable link starts and end on the backup path after the fork of the shortest path and the backup path. In addition, all support links will be directed from the shortest path to the backup path. Therefore, after a packet takes a support link, it will follow the direction of the backup path and only join the shortest path again on the last link as this link connects the shortest path back to the backup path. If the support link connects to a node which is on the shortest path and the backup path, it will also follow the shortest path until it reaches D. This covers all cases of a path indicator which concludes the proof that the path indicator is loop free. □

3.9.4 Multiple Path Metrics

In this section, we state how multiple path metrics are supported. As indicated before, the source simply computes the path indicator for each desired path metric individually. In particular, the source can use the keys of the same session for the same path set entities. For each packet, the source decides which path indicator should be used depending on the path metric to optimize for. We do not investigate how exactly the source chooses the path metric to optimize.

3.10 Adaptive Routing Scheme

In the following, we will outline how S can send to a destination D with source and data authentication. We state how S creates the packet header, how the size of the packet header is determined and how an intermediate node can use the packet header for source and data authentication. Notice that we assume that S chooses the path indicator for the desired path metric for each packet beforehand.

3.10.1 Packet Header Construction

In the following, we will outline how S creates the packet header. We assume that S has used the k-shortest d-path set algorithm before to create the path set. In addition, S has used the key exchange

protocol before to create the session. Now, S shares the keys with the other path set entities to create the header to send a packet with source and data authentication. In the following, we describe the fields S includes in the packet header. The structure of the packet header is given in Figure 3.19.

- MTL: MetaInf Length, i.e., the length of the MetaInf field given in multiples of 32 bits The length field is of size 4 bits. Therefore, the maximum size of the MetaInf field is $2^4 \cdot 32 = 512$ bits
- Position: Label position of the current path set entity for efficient field access. A pointer which is used to calculate the offsets to the i-th identification field ID_{R_i}, the i-th verification field V_i and the entry of the i-th path set entity in the path indicator $INDICATOR_\sigma$. More details are given below.
- The number of path set entities n which is used to calculate the size of the header as explained below
- DataHash $= H(P \| INDICATOR_\sigma \| \text{MetaInf})$: The hash over the constant information to be authenticated. P is (part of) the payload to be authenticated. $INDICATOR_\sigma$ is the path indicator and MetaInf is the application meta-information. Note that the path indicator is composed of the path indicators for each entity, i.e., $INDICATOR_\sigma = \sum_{i=1}^{n} INDICATOR_{\sigma,R_i}$
- SessionID $= H(PK_\sigma \| PATHSET_\sigma \| T_\sigma)$: The hash of the session's public key, path set and creation timestamp
- MetaInf: (Optional) authenticated application meta-information. Its size is given with the MT length field
- $\mathrm{V}_j = MAC_{K_j}(\text{DataHash})$: The verification fields V_j's for all path set entities R_j created with a MAC function over the DataHash keyed with the shared key K_j
- $INDICATOR_{\sigma,R_j}$: Part of the previously calculated path indicator for the node R_j, i.e., the successors of path set entity R_j. Note that the path indicator is divided into equal sized parts to enable efficient access
- ID_{R_j}: Identifier for each R_j which defines the order or labeling j for the path indicator labeling and the order of the V_j's

3.10.2 Packet Header Size

In the following, we will present a bound on the size of the packet header. Let n denote the number of path set entities.

First of all, we will analyze the size of the path indicator $INDICATOR_\sigma$ in general and then a limited variant which is used in the packet header. Note that the number of successors for the shortest path and backup path are fixed, as each path separately is a path and each path set entity has one successor, i.e., is bounded by $\mathcal{O}(n)$. However, the number of support links is bounded by the links from the node where the unreliable link starts to each entity on the backup path after the fork of the shortest path and backup path. Therefore, the maximum worst case size of the path indicator in general occurs if (1) the shortest path is only between source and destination $< S, D >$, (2) the backup path starts at S and ends at D and has b many additional entities on the path and (3) the b entities on the backup path each have a link to the node where the unreliable link starts. In this case, b is in the order of $\mathcal{O}(n)$. Therefore, in general in the worst case, the path set size is in $\mathcal{O}(n)$.

However, we limit the size of the path indicator to at most 4 successors for each node. This means that the path indicator can encode two support links in the worst case for the node where the

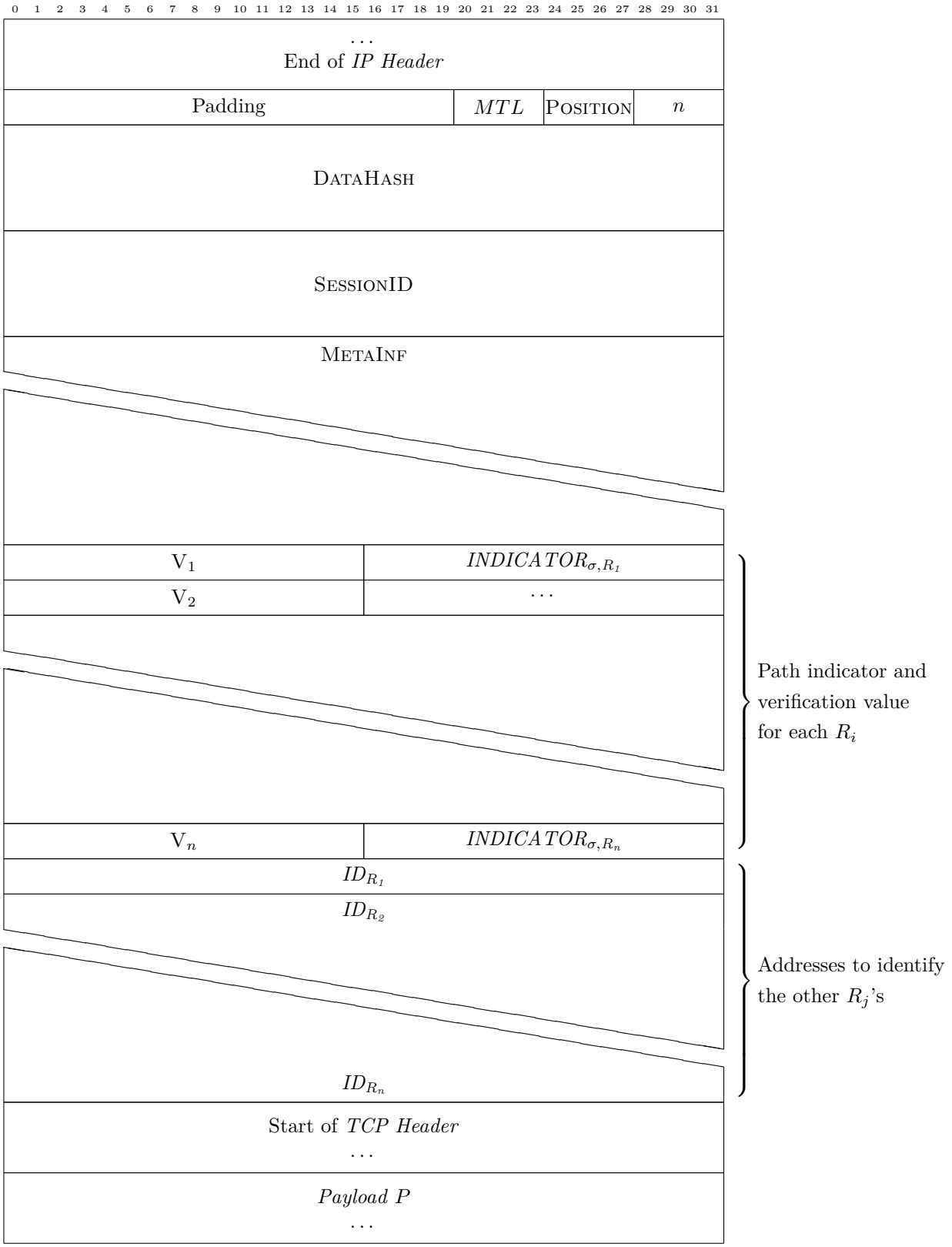


Figure 3.19: Protocol header of the source and data authentication scheme. n is the number of node entities R_j.

unreliable link starts. Note that this is only with regard to the path indicator size, but the adaptive routing takes the first successor which is available. The limiting factor is the node where the unreliable link starts since it has one successor from the shortest path and potentially one successor from the backup path which leaves two potential successors as support links. In addition, we split the path indicator into a part for each path set entity which denotes the successors for this path set entity. We fix the size of it to 16 bits. In particular, we use 4 bits per successor encoding. Therefore, we can support a path set of maximal size $n = 2^4 = 16$. Note that this also defines the size of the POSITION in the header as the labeling corresponds directly to the scaling factor used to calculate the offset for each path set entity.

In the following, we analyze the size of the scheme header as a function of the meta information length MTL and the path set size n. Note that the header is 32 bits aligned. The size of the first 6 32 bits words is fixed and given by $7 \cdot 32$ bits. The size of the METAINF field is variable and given by $MTL \cdot 32$ bits. The size of the dynamic fields depends on n and is given by $2n \cdot 32$ bits. Therefore, the header size is given by $(7 + MTL + 2n) \cdot 32$ bits. For instance, for $n = 6$ path set entities and $MTL = 1$ we have $(7 + 1 + 12) \cdot 32 = 640$ bits or 80 Bytes. For $MTL = 1$, the minimum size is 48 Bytes and the maximum size 152 Bytes.

3.10.3 Cryptographic Properties

In the following, we will discuss on what the provided level of security of the scheme depends. Mainly, the verification fields V_i's are used by the path set entities to authenticate the source and the data by authenticating the DATAHASH. It is important that the DATAHASH includes at least part of the hash of the payload to prevent an attacker to copy and add a packet header for another packet with different payload. In addition, the SESSIONID is crucial since it is used to derive the keys for a specific session within a path set. For both hashes, DATAHASH and SESSIONID we want the used hash function to be robust against a *second-pre-image collision attack*. In short, in such an attack an adversary is given a hash value $h(x)$ and the input used x and needs to find another input $x' \neq x$ for the hash function h which hashes to the same value, i.e., $h(x) = h(x')$. The complexity of this problem for n bit hash output is given by $\mathcal{O}(2^n)$. Note that this problem is harder than the general collision resistance property in which an attacker needs to find two distinct inputs which hash to the same output, i.e., produce a collision[1] The common complexity bound to be secure against a second-pre-image collision attack is given by 2^{80} [14, 40]. Therefore, we need at least 80 bits of security, i.e., at least 80 bits hash output size. This means we can for instance use SHA1 or SHA3 truncated to more than 80 bits. In addition, in our setting the time of a session is limited in orders of minutes. Therefore, an attacker does not have much time available to compute collisions for the given hashes.

We want to optimize the time needed at the intermediate nodes for the encryption during the key setup and the hash calculation during the adaptive routing, i.e., we prefer public private key encryption schemes which take longer to decrypt at the destination than to encrypt at the intermediate nodes. Therefore, we propose to use the following hash functions and output sizes.

- For the DATAHASH over potential a lot of data we propose to use PMAC and truncate its value to 128 bits. PMAC is an efficient block-cipher mode of operation to produce a MAC as it can be fully parallelized [25]. However, we did not have an efficient implementation of PMAC available and for simplicity used SHA1. We truncated the output value to 96 bits which is still secure as described before.

- To calculate the SESSIONID we use SHA1 and truncate its value as described above.

- We instantiate the MAC function seeded with the local keys and the pseudo random function

[1] Which for n bit hash output is given by $\mathcal{O}(2^{\frac{n}{2}})$.

(PRF) F with the same hash function, CBC-MAC based on AES which is efficiently implementable on AESni hardware.

- Note for the verification values V_i's if we assume a secure MAC function the success probability of a forged MAC is given by 2^{-n}. In addition, we claim the main information for the attacker which is interesting to forge is the DataHash $= H(P \| INDICATOR_\sigma \| \text{MetaInf})$, i.e., to forge part of the payload, part of the path indicator or the MetaInf. The attacker cannot know the derived key at the nodes since the local secret is not known and can therefore only guess a verification value. In addition, the session and therefore the keys are only usable for a limited time. Therefore, we propose to use 16 bits which keeps the scheme header size small. This size already results in a sufficiently low success probability. The forgery of one verification value is successful with probability $2^{-16} = \frac{1}{65536}$. Therefore, forging m many verification values is only successful with probability $2^{-16^m} = \frac{1}{2^{16m}}$. In particular, the attacker cannot check the validity offline but only when it sends the packet to the entities. Therefore, a size of 16 bits is sufficient.

3.10.4 Adaptive Source Authentication and Routing

In the following, we will outline in detail how a node in the path set uses the packet header to authenticate the source and data. First, we explain how a path set entity R_i extracts the information from the header when it receives a new packet. It reads the Position and the length field n. With these two fields the path set entity can infer the whole structure of the packet header and knows its own labeling i from Position with which it can access the verification value V_i and its successors in the path indicator $INDICATOR_\sigma$ in constant time. For instance, the first path set entity ID_{R_1} receives label 1. Note that the start of the sections of each field is given with the total length n and the fixed size fields. To calculate the offset in bits for each field individually, the Position number is multiplied with a fixed size for each of the fields as explained in the following. We define the *total offset* of a field as offset from the start of the scheme header to the respective field. For the aligned verification value V_i and path indicator $INDICATOR_\sigma$ we have in total $(7+MTL+\text{Position})\cdot 32$ bits offset and for the path indicator successors we have Position $\cdot\, 4$ bits offset within the path indicator itself. For the inferred successor id ID_{R_j} the offset is given in total with $(7 + MTL + n + j) \cdot 32$ bits offset.

In the following, we explain how the path set entity R_i uses the information from the packet header in the *verification step*. After the initial Position and the length field n have been read, it uses the SessionID and its local secret to calculate the secret K_i shared with S for this session, i.e., $K_i = F_{SV_{R_i}}(\text{SessionID})$. Afterwards, R_i uses the calculated key and the DataHash to calculate its MAC verification value $\mathrm{V}_i = MAC_{K_i}(\text{DataHash})$. If the value matches with the MAC verification value in the header at its own position V_i, the authentication has been successful and the node continues to the adaptive routing step. If the values do not match, the packet is discarded. Note that the destination will in addition also recalculate the entire DataHash value. The advantage of that is that (1) intermediate entities will save computation time as they can reuse the precomputed hash and (2) recalculation at the destination ensures that a packet alteration is still detected at the destination since the recomputed DataHash value would differ.

In the *adaptive routing step*, the path set entity R_i reads the next path set entity to which the packet has to be sent next. To achieve this, it accesses the successors in the path indicator $INDICATOR_\sigma$ at its position as explained above. It chooses the path set entity with the highest priority which is available. We define an entity to be available as (1) the link between the entities has not been detected to be broken and (2) the information that the unreliable link is broken has not been distributed or reported, in particular not by the successor.

In more detail, the path set entity must read the label j of the successor R_j with the highest priority. Afterwards, it uses the label j to calculate the total offset to the identity field ID_{R_j} as explained above. With this offset, the path set entity R_i reads the identity of the potential next node

and can check if it is available. If it is not available, it will repeat the procedure for the entity with the second best priority and so on. Note that each access is constant since the offsets for each fields can be calculated directly by the path set entity R_i. After the next path set entity R_x has been selected, the POSITION field is updated directly with the label x from the next path set entity in the path indicator. This is possible since the path indicator uses the same labels as for the offset calculations in the packet header to avoid additional calculations. Finally, the packet is forwarded to the next path set entity R_x. The properties of the path indicator ensure that the packet is routed towards D in an efficient matter supporting error recovery for the identified unreliable link(s).

3.11 Adaptive Source Authentication Method Security Analysis

In the following, we will prove that the adaptive source authentication scheme provides source authentication. Remember the assumptions from Section 3.5. The proof is independent of the underlying network, i.e., holds for all network topologies. Note that the routers in the network can be malicious, but the source and destination are trusted. In addition, note that our cryptographic primitives used are secure as argued in Section 3.10.3. We prove that the source authentication property always holds as it defends against our attacker model from Section 3.3.

Packet alteration To alter a packet, a malicious router needs to recompute the verification values V_i's of all path set entities where the packet could be potentially routed after the packet alteration. However, these values can only be computed with the keys K_i's of the path set entities derived in the session setup which only S and D see. Note that this implies that S or D indeed could forge packets and therefore, they must be trusted. Note that the DATAHASH is only recomputed fully at the destination D. However, on the one hand, this still prevents forged packets to be accepted at D and on the other hand the DATAHASH could be recomputed at each intermediate entity if faster detection is preferred to efficient forwarding.

Packet injection A malicious node cannot create an arbitrary packet and inject it into the network for the same reasons as in the *packet alteration* attack. The malicious router does not posses the shared keys K_i's of the path set entities and can therefore not create the verification values V_i. However, for the special case of a replay attack, a malicious node can resend a packet with the exact same payload. Nevertheless, should the scheme also prevent replay attacks, a timestamp can be added to the packet header and included into the DATAHASH. In an additional step a path set entity can verify if the timestamp is still within reasonable bound.

Denial-of-Service (DoS) attack Note that the source S only sends packets and can therefore not be targeted for DoS attacks. We consider memory exhaustion attacks. Each path set entity uses only one local key to derive the session key and does not state per-flow, per-source or per-destination state at routers memory exhaustion attacks are not possible for the adaptive source authentication scheme. Now, we consider computation exhaustion attacks. By design, the scheme puts the workload on the source and destination. Consequently, path set entities need only few clock cycles for the verification and adaptive routing step as analyzed in Chapter 5 which makes them resilient against computation exhaustion attacks.

3.12 Adaptive Source Authentication Method Example

In the following, we give an example of the adaptive source authentication method. S wants to send to a destination D with source and data authentication. We assume that S already performed the following tasks: (1) S calculated the shortest path, (2) identified the unreliable link, (3) calculated

the path set with the k-shortest d-path set algorithm and (4) run the key exchange protocol for the path set. We continue the example from this chapter from Figure 3.11 which resulted in the path indicator Figure 3.15. Assume the path indicator is given by $S^1A^2B^3C^4E^5D^6\|2|35|45|6|4$ as described in Section 3.9.2. Note that the labeling corresponds to the POSITION number of each path set entity.

We show how S sends a packet with adaptive source authentication. Assume S has the packet H with payload and transport header. In addition, assume $MTL = 1$ and we have $n = 6$. S creates the fields for the packet header as described before. In particular, it will create verification values V_i for each path set entity R_i. Now, S performs the adaptive routing phase. S initializes its own POSITION with 0. It gets the label i of its successor with highest priority at total offset $(7 + MTL + \text{POSITION}) \cdot 32 + 16 + 0 \cdot 4 = 272$ bits. In our example, $i = A = 2$. Now, S looks up the identity R_A of its successor at offset $(7 + MTL + n + A) \cdot 32 = 512$ bits. It will check if there is routing information available to each R_A. Assume in our example that this is the case, then S will update the POSITION field with A's label 2. S adds the finished scheme header to the packet headers after the transport header and sends it to R_A which means it adds the network layer and finally link layer header.

Afterwards, the packet is received at path set entity A. The situation at A is depicted in Figure 3.20. A starts with the verification step. First, it reads the values $MTL = 1$, POSITION $= A = 2$ and $n = 6$ from the packet header. Now, it reads the SESSIONID and takes its local secret to calculate the secret K_A shared with S for this session, i.e., $K_A = F_{SV_{R_A}}(\text{SESSIONID})$. Afterwards, A reads the DATAHASH field and calculates its verification value $\mathrm{V}_A = MAC_{K_A}(\text{DATAHASH})$. A reads the verification field V'_A from the packet at offset $(7+MTL+\text{POSITION})\cdot 32 = 300$ bits. If the values match, i.e. $\mathrm{V}_A = \mathrm{V}'_A$, the authentication has been successful and the node continues to the adaptive routing step. If the values do not match, the packet is discarded. The adaptive routing step at A is similar as at the

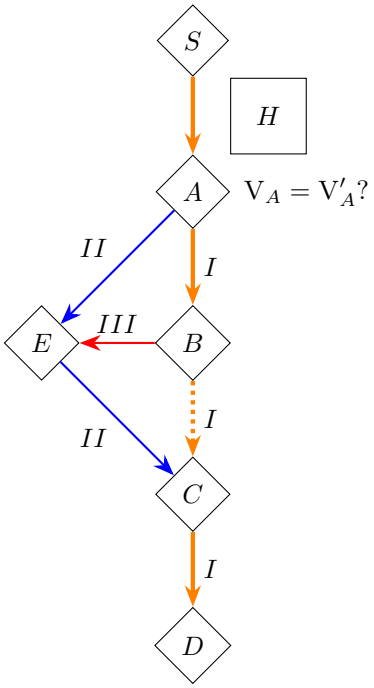


Figure 3.20: The packet H with source authentication header is processed at path set entity A which checks if the verification value is correct and afterwards proceeds to the routing step.

source S. A looks up the label $B = 3$ of the successor with highest priority, gets the id R_B and checks if it is available. As B is available, it updates POSITION to 3 and forwards the packet to B.

The packet is received at path set entity B. Assume that the unreliable link now is not available.

The situation at B is depicted in Figure 3.21. B starts with the verification step which is the same

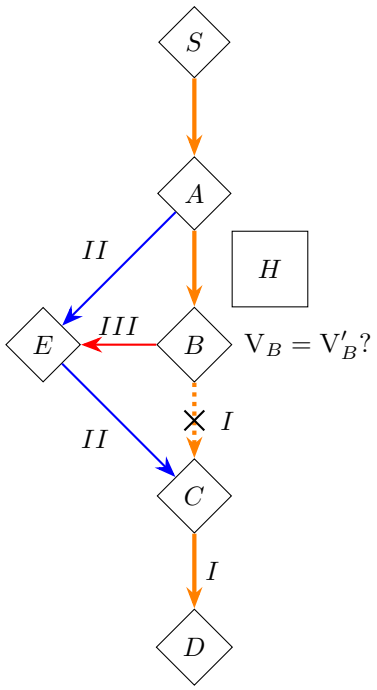


Figure 3.21: The packet H with source authentication header is processed at path set entity A which checks if the verification value is correct and afterwards proceeds to the routing step.

as at path set entity A. If the verification values match, i.e. $V_B = V'_B$, match, B will continue to the adaptive routing step.

The adaptive routing step at A is similar as at A and S. B looks up the label $C = 4$ of the successor with highest priority, gets the id R_C and checks if it is available. B realizes that the link is not available. Therefore, B looks up the label $E = 5$ of the successor with second highest priority, gets the id R_E and checks if it is available. As E is available, B updates the POSITION to 5 and forwards the packet to E.

The cases for the other path set entities are similar.

3.13 Adaptive Source Authentication Method Summary

In the following, we will give a high level summary of the adaptive source authentication method. It enables a source S to send to a destination D over a path set with source and data authentication. Furthermore, it enables a adaptive routing to support a backup mechanism for selected unreliable link. In addition, it supports application needs, e.g., the best low latency path for one application and the best high throughput path for another application. In the following, we summarize the involved steps for the source S of the scheme.

- S is given a destination D and computes the shortest path to D
- S identifies the unreliable link(s) on the shortest path
- S calculates the path set with the k-shortest d-path set algorithm
- S runs the key exchange protocol to create the session
- S creates the path indicator for the adaptive routing

- S can now use the path indicator and the cryptographic material from the session setup to create the packet header to send packets authenticated

4

Implementation

In this chapter, we describe the implementation architecture in Section 4.1 and outline implementation design adaptions and details in Section 4.2.

4.1 Implementation Architecture

In this section, we will outline the *distributed zone-based firewall* implementation architecture and design of the scheme. In short, the scheme has been implemented as a set of new rules for `iptables` which provide source authentication. In addition, the session setups, i.e., key exchange protocol is performed in user space and for efficient access exported to the kernel space. Note that during development we used the Linux kernel version `3.18.10` and the `iptables` version `1.4.18`.

4.1.1 Implementation Overview

First of all, we give an overview of the implementation about the possible packet flows in the netfilter Linux kernel. In particular, we explain the different cases of packet flows which can occur in the implementation of the scheme. The scheme can be activated per incoming and outgoing interfaces at one system. In more detail, we can specify for which interfaces the source authentication scheme is activated, i.e., for which interfaces the header must be added or later removed. No packet with the custom header should leave the internal network.

In short, we have the following scenarios for a packet which comes from an external interface.

- The packet arrives at an incoming interface, must be forwarded and..
  - .. carries the scheme header and the node is not the last destination within the scheme, i.e., the header must be updated
  - .. carries the scheme header and the node is the destination within the scheme, i.e., the scheme header must be checked and removed
  - .. does not carry our scheme header but a session has been setup, i.e., we can add the scheme header with this information
  - .. does not carry our scheme header and a session needs to be setup before we can add the scheme header
- The packet is created locally, does not carry our scheme header and..
  - .. a session has been set up, i.e., the scheme header can be added with this information
  - .. a session needs to be setup before the scheme header can be added

- The packet arrives at an incoming interface, is delivered locally, carries the scheme header and we are the destination within the scheme, i.e., the scheme header must be checked and removed

Note that the different scenarios can be nested, for instance, after the session setup, we need to add the scheme header.

4.1.2 Iptables Introduction

`iptables` is a user space program in Linux systems to manage the Linux kernel firewall [7]. The Linux kernel firewall is abstracted into tables of *chains* of *rules*. Each table corresponds to a different stage during the packet processing by the Linux kernel. The rules are applied sequentially per chain to provide policy enforcement. `iptables` allows to change and configure these tables, chains and rules. The Linux kernel firewall is implemented as different netfilter kernel modules called packet filtering framework [12]. Netfilter provides a hook API in the kernel which allows kernel modules to register callback functions for these hooks. This means that at each hook of the network stack in the kernel the registered functions are called if a packet traverses this hook [43]. The registration of these hooks is managed by `xtables` on which `iptables` itself is built. An overview of the netfilter components is given in Figure 4.1. Each rule in `iptables` is composed of one or multiple options

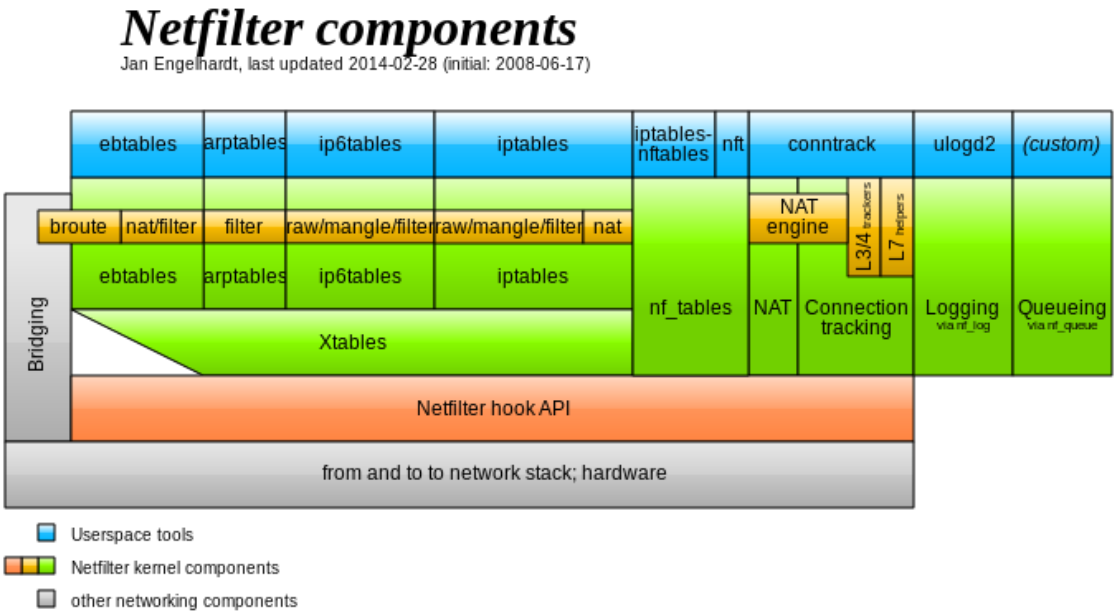


Figure 4.1: The netfilter components by [11].

for classification which are referred to as *matches*. These options classify a packet and if all match, one action which is referred to as *target* is performed on the packet. A `iptables` match or target module consists of a kernel module which defines a callback function to be registered at specific hooks and a `iptables` user space extension which define the actual rule interface for the defined kernel modules. The callback function in the kernel module is called whenever a corresponding `iptables` rule in a table for a specific hook has been defined. Note that a netfilter module is usually written for `xtables` such that the module can be written generically for IPv4 and IPv6 [26]. For details about the kernel module implementation please consult the source code provided in [31]. More information about kernel module programing is available in [26].

4.1.3 System Modules Architecture

In short, three kernel modules, two user-space `iptables` extensions and one user space daemon are needed to implement the scheme. For `iptables` we provide a target `SRCAUTH` with three options and a match `srcauthmatch` with three options which each correspond to one kernel module. In addition, the target and match each need a lightweight `iptables` user space extension to make them available to the user space `iptables` command. In addition, we provide a database (DB) kernel module to hold the session material and additional routing information. The DB module also provides an API to the other kernel modules and the user space daemon. The daemon handles the session setup and update of routing information in user space. To achieve this, it calls the session setup Perl script if a packet with a new destination id arrives and it periodically calls the routing Perl script. In addition, it calls the server which handles incoming packets from the session setup script. The daemon converts the data from the Perl script for the kernel and passes this information to the DB kernel module though the `procfs` interface. The main design decision behind the architecture is to make the kernel modules fast such that the adaptive source authentication performed according to the `iptables` rules is efficient and scalable. We describe each module in more detail below. An overview of the system modules architecture is given in Figure 4.2.

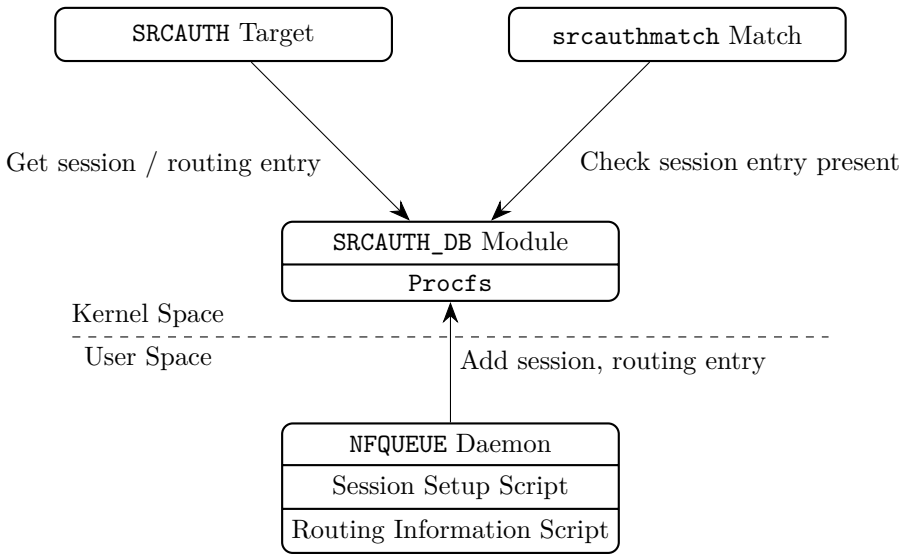


Figure 4.2: System modules architecture diagram depicting the kernel modules and the user space daemon.

The first kernel module, the `iptables` target `SRCAUTH`, handles all manipulations of the scheme header and main cryptographic operations with regard to the scheme. It provides the following three options. First of all, we need to be able to add the header of the scheme to the packet. This can be achieved with the option to specify the meta-information, e.g. a zone number x, `--set-information` x. A network interface can now be bound to a zone using the interface option in `iptables`, e.g. `-i eth1`. The second option `--update-header` is used to update the header at the intermediate nodes. In particular, it will perform the adaptive source authentication verification and adaptive routing step. Using the Position value, it detects if it has to perform the efficient verification for intermediate path set entities or an extensive final verification at the destination which recomputes the DataHash. As only this option is responsible for the verification, its correct use is crucial for the correct secure use of the system. It drops the packet if there is a scheme header but the validation is not successful. The third option `--remove-header` is used at the destination to remove the header, e.g., before it leaves the internal network.

The second kernel module is a match called `srcauthmatch` for the header of the scheme. We provide three options. The first option `--no-header` is used to check if no scheme header is currently present. The second option `--session-present` checks if the session setup for this destination has been done, i.e., if the session information is available to create the header. These two options can also be inverted using the standard `iptables` notation `! --session-present` and `! --no-header` to match for the inverted result. Finally, the third option `--has-information` x is used to match against a specific meta-information, e.g., a source zone number x and validate the header. As the match module might be called quite often for packet classification it is by design kept small and efficient. However, note that this means that the verification of the meta-information has to be performed in advance with the target module.

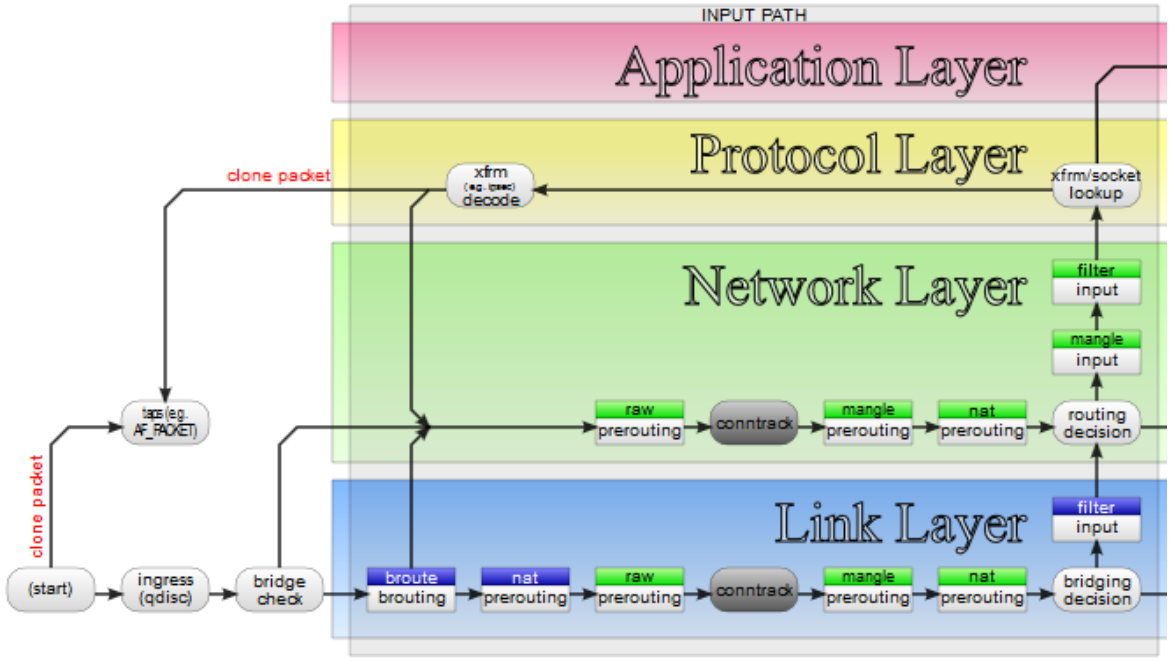
The third kernel module `SRCAUTH_DB` holds the actual information about the current sessions and the routing information for adaptive routing. It provides an API to the other two kernel modules to read the information. In addition, it provides an API to the user space using `procfs` to receive the session information from new sessions and new routing entries for the adaptive routing. In short, the `procfs` is a special file system which allows to support writing and reading information to and from kernel memory [15]. In addition, it provides debug state information and database state information through `procfs`. Note that the adaptive routing information is the local mapping from destination identifiers, e.g. OSPF router ids, to the actual destination IP address. We give more information about OSPF later in 4.2.3.

The session setup by the source, i.e., the path set calculation and the key exchange protocol is implemented in user space in a daemon. The daemon makes also use of scripts in `Perl` to run specific tasks. In particular, the actual key exchange protocol and script to get the routing information are implemented in `Perl`. The daemon gets new packets from `iptables` if no session information is available for this packet. To achieve this, the `NFQUEUE` target is used. This target is provided by the `libnetfilter_queue` which is a user space library providing an API to the queued packets in the kernel [9]. It is possible to read and manipulate the packets in user space and then inject them back to the originating `iptables` chain and table in the kernel. In addition, it uses an API provided by the kernel module `SRCAUTH_DB` over `procfs` to store the data for the new session created.

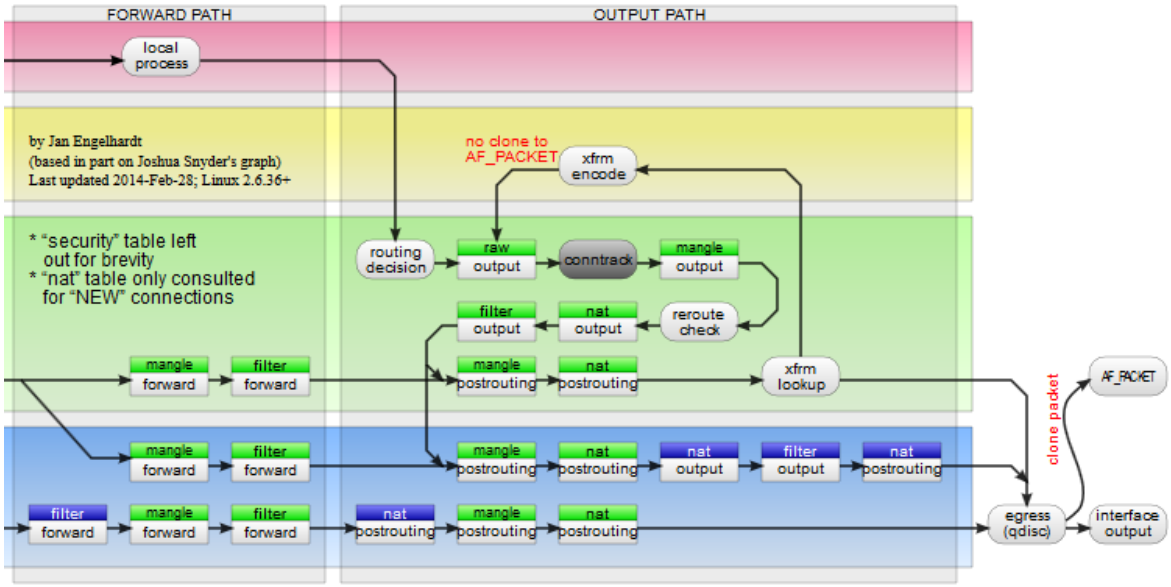
To store and search for a specific session we use the original destination address to look up the unique destination identifier. We call this step *mapping* which is needed to minimize the number of sessions needed as they map to the same destination identifier for packets with the same exit point within the *distributed zone-based firewall*. The key exchange protocol and scheme is performed between the entry and exit points of the distributed zone-based firewalls. It is computed by the session setup script and described in more detail in Section 4.2.4. This unique identifier serves as an index for a hash table which stores the actual session information. Afterwards, the packets are injected back to the `iptables` chain and table and processed further by the next rules. In particular, the header of the scheme will be added afterwards to the packet if an appropriate rule is present.

4.1.4 Netfilter Hooks Selection

Note that the adaptive source authentication rules are designed to activate adaptive source authentication per interface or for the local output. Nevertheless, the rules can be used at any table in `iptables`. Note that the implementation is strict in the sense that for every packet sent to the options of the target module, the header is added, updated and verified or removed. In the following, we outline at which netfilter hooks the rules have to be applied for the common case where the packet comes from an external incoming interface and continues on an outgoing interface. We refer to the netfilter hooks as shown in Figure 4.3 [11]. Consider the case for an incoming packet, this happens at an entry point of the distributed firewall. To support the `iptables conntrack` module, we check if a session setup has to be performed and add the scheme header at the hook `PREROUTING` of table `mangle`. The reason for this is that we additionally alter the packet to achieve custom paths for the adaptive routing as



(a) `INPUT` path of the netfilter packet flow.



(b) `FORWARD` and `OUTPUT` path of the netfilter packet flow.

Figure 4.3: The netfilter packet flow and hooks by [11].

described later in Section 4.2. On the other hand, we can verify and potentially update the scheme header when we first receive the packet which is at the hook `PREROUTING` in table `raw`. This ensures that the packet is efficiently discarded if the authentication fails and prevents that other operations are performed before the packet is discarded. Finally, we check for the meta-information and remove the scheme header at the netfilter hook `POSTROUTING` in table `mangle`. The reason for this choice is that it is the last hook applied for every packet and we need to have the outgoing interface available for the packet classification.

4.1.5 Example Rules

The `iptables` rules in Table 4.2 illustrate how the modules are used in combination to activate the adaptive source authentication scheme for an incoming interface and an outgoing interface abstracted by a zone number 1. In particular, other interfaces could also be assigned to this zone number which can be used independently of the interfaces for policy enforcement. For convenience we provide an overview of the target and match options in Table 4.1.

`SRCAUTH --set-information` x `/ --update-header / --remove-header`
`srcauthmatch --no-header / --session-present / --has-information` y

Table 4.1: Target and match options overview.

4.1.6 Minimum System Requirements

The kernel modules use kernel code infrastructure which require the kernel version `3.10` or newer. The reason for this is that the interface to the `procfs` has been changed in kernel version `3.10` which is used by the database kernel module.

4.2 Implementation Details and Adaptions

In the following, we will outline design decisions and adaptions which have been conducted during design and implementation of the prototype. Some minor adaptions only enabled a cleaner and more efficient implementation of the prototype with no effect on performance or functionality while other adaptions were needed in the specific system to make the prototype implementation feasible.

4.2.1 Socket Buffer Header Injection

A main challenged during the implementation was to add the custom scheme header to the socket buffer while ensuring that the normal operation of the Linux network stack is not disturbed. The scheme is implemented for IPv4 as it is still most widely used and deployed. The IP protocol header provides a method to add arbitrary data to its header called *IP option* [8]. However, the maximal size of such an option is given by the ip header length field `IHL` which is 4 bits long. In addition, the required IPv4 header fields are of size 20 bytes Therefore, the maximal option size is $10 \cdot 32 = 320$ bits or 40B. We see that this is not enough to store our header as our header size in the implementation varies from 52B to 156B. Note that the header is 4B larger than described in Section 3.10.2 as an additional implementation specific field is needed as described below in Section 4.2.8. Therefore, we use the custom IP header `protocol` number 250 and add the adaptive source authentication header after the IP header on the packet. Note that this implies that we have to save the original protocol field to ensure proper forwarding after the adaptive source authentication when the packet leaves the distributed firewall. This method has the advantage that we do not interfere with the IP header, especially if IP header options are defined. However, it required a lot of effort to identify how the

Rule	`iptables -t raw -A PREROUTING -i eth1 -m srcauthmatch ! --no-header -j SRCAUTH --update-header`
Description	Check if the header is already present. If it is, perform the adaptive source authentication verification and adaptive routing step. This rule ensures proper forwarding and header verification and update at intermediate nodes and the destination.
Rule	`iptables -t mangle -A PREROUTING -i eth1 -m srcauthmatch --no-header ! --session-present -j NFQUEUE --queue-num QUEUE_SESSIONSETUP`
Description	Pass packets for which no session is present to user space daemon for session setup, i.e., execute the key exchange protocol including path set calculation at the source.
Rule	`iptables -t mangle -A PREROUTING -i eth1 -m srcauthmatch --no-header --session-present -j SRCAUTH --set-information 1`
Description	Add the header of the scheme at the source. In addition, perform initial operations needed for the adaptive routing. Note that this rule usually immediately follows the previous rule for session setup.
Rule	`iptables -t mangle -A POSTROUTING -o eth2 -m srcauthmatch --no-header -j DROP`
Description	Note that if only packets verified with the scheme are desired, all packets which do not contain the scheme header have to be dropped. For instance, this can be done at the output interface before the header is removed.
Rule	`iptables -t mangle -A POSTROUTING -o eth2 -m srcauthmatch --has-information 1 -j SRCAUTH --remove-header`
Description	Check if the (previously authenticated) meta-information is present and remove the header if so. This rule is needed if the packet leaves the internal network at the distributed firewall exit point.
Rule	`iptables -t mangle -A POSTROUTING -o eth2 -m srcauthmatch ! --no-header -j DROP`
Description	Note that successfully verified packets are not dropped if the meta-information does not match in a rule. On the one hand, this provides the option to react with different rules on different meta-information. On the other hand, the packet has to be discarded by default if no rule to remove the header matched.

Table 4.2: Example rules to perform policy enforcement with source authentication.

header can be injected within the target module netfilter environment and the socket buffer structure. The target can inject and remove our scheme header into the common Linux network stack data structure used for packet called *socket buffer* or `skb` [23]. Note however, that these operations make especially the injecting of the header more expensive, especially since all headers in the skb are also updated relatively to the new header to prevent potential side effects in the complex Linux network stack and to ensure that other target and match modules continue to work as expected.

4.2.2 Database in the Kernel

As mentioned before, the database provides an API to the user space daemon which can write the session, mapping and adaptive routing information. In addition, these information can be accessed by the target and match module. The three information assets are stored each in a hash table for efficient lookup. The implementation has been designed to optimize the access to the database from the target and match modules. However, note that we have concurrent access and even potential deletions on these information. The main problem is that outdated session material can be deleted while it is read by another process shortly before it timed out. Therefore, for the sake of simplicity, the current prototype implementation enforces sequential access on the essential lookup or deletion functions. This ensures that no concurrency problems such as wrongly read information can occur. In addition, we copy the session data or the routing information to the caller such that if later the data is deleted this is not a problem at all. However, this also means that the implementation is not yet fully optimized for speed but as a first step, correctness has been prioritized over speed. Nevertheless, the lookup of the session information is already designed to support a more efficient concurrent lookup, i.e., roughly that session information is only deleted in one function at one point if the timestamp is too old. However, unfortunately, these optimizations could not be fully implemented, refined and tested. We give more information about the performance of the prototype in Chapter 5.

4.2.3 OSPF Information

In the following, we will outline information about *Open Shortest Path First (OSPF)* and how it is used for the prototype implementation. First, we will outline the key features and algorithms used in OSPF V2 [34, 35, 36]. In [34], Moy summarizes and analyzes the OSPF protocol version 2 (V2) as described in [35]. OSPF is an interior gateway protocol (IGP) to be used in one autonomous system. It uses a link-state routing algorithm, i.e., roughly, every node constructs locally a map of the network. Each node uses its map to calculate the best path to every other potential destination. These best paths yield the node's routing table. In more detail, in OSPF each node maintains a link state database. This database is formed using many link state advertisements (LSAs). To ensure an identical link state database at each node, the LSAs are flooded and a reliable flooding algorithm is used to synchronize the databases. Each node sets itself as the root and applies a method based on the Dijkstra Algorithm on the link state database to calculate the shortest path tree which is used to form the routing table. OSPF supports and provides complex features and abstractions such as the definition of OSPF areas, flexible import of external routing information and an adjacency concept for routers, roughly, to manage the flooding of information. In particular, one area is the backbone area `0` which manages and distributes the information from all other areas. The participating nodes identify each other with a unique *OSPF router id*. In short, this router id is used to determine the origin of aforementioned announcements as *advertising router*. What is more, OSPF provides flexible routing metric by definition of a metric rule for outgoing router interfaces. The cost of a path is then the sum of the interface costs. In addition, the costs can be set manually to reflect any arbitrary combinations of metrics. For more information please consult [34, 35, 36].

We give a simplified overview of the distributed firewall system setting. On our distributed firewall systems, OSPF is implemented with *Quagga* [16]. Quagga is a routing software suite which among

others provides implementations of *OSPF v2* for Unix platforms [16]. To access the information for the scheme, the session setup and routing script access the information from OSPF through the standard integrated shell `vtysh` for Quagga [20]. In more detail, OSPF is run on a separate network on top of the underlying networks of each distributed firewall. This separate network is realized with the *Generic Routing Encapsulation GRE* protocol which allows other protocols to be encapsulated to point-to-point links over an IP network [4, 27]. In our system, the GRE itself runs over the IPsec tunnels, i.e., is encapsulated with the IPsec ESP header. Each of the distributed firewall systems can be uniquely identified with an *OSPF router id* which is part of the GRE network. Note that all our target systems run *OSPF* with one backbone area `0` which means that all participants have access to the whole distributed firewall topology. In the following sections, we will outline how the information from OSPF is used in the session setup and routing script.

4.2.4 Session Setup and Routing Script

First of all, the source needs to be aware of all the potential path set entities and the links between them. All our target systems run OSPF with one backbone area `0` which means that all participants have access to the whole distributed firewall topology. In addition, the OSPF router ids are used to identify each path set entity uniquely. The source accesses the OSPF database information through the standard Quagga integrated shell `vtysh` within the session setup and routing `Perl` script.

The session setup script parses the complete OSPF database to extract the topology for the scheme. In addition, the *mapping* is performed which calculates the OSPF router id of the exit point within the distributed zone-based firewalls. In particular, the script finds the distributed firewall system which announces the network in the range of the packet's original destination address. The scheme is now performed between the entry and exit points of the distributed zone-based firewalls. Afterwards, the k-shortest d-path set algorithm is performed as described in Chapter 3 to calculate the path set. For Yen's Algorithm we make use of an external implementation written in `C++` by [39]. The unreliable link identification is done with a weighted average value available for each link between all OSPF router id pairs. The link with the lowest value reflects to be most unreliable. The weighted average values are provided by an additional small external daemon. The external daemon simply parses the whole OSPF database within a certain period and updates the weighted average lifetime values as described in Section 3.6.

We do not outlined detailed parsing of the OSPF database and refer to the actual source code for more information [31]. To enforce the paths during the key setup, the actual key exchange packet is sent directly to the address of the next path set entity in the path. In addition, a field is sent which tells the next recipient its position within that path such that entities efficiently know where they are.

The routing script parses the OSPF neighbor information locally available. In short, this returns the OSPF router ids which are reported by OPSF to be accessible by a certain address. When the routing script is called by the nfqueue daemon, it returns this mappings of OSPF router id to address. The prototype implementation of the daemon periodically calls the routing script and writes the routing information to the kernel DB. The implementation of the scheme is otherwise following the formal description in Chapter 3.

4.2.5 Adaptive Routing Implementation

To support adaptive routing, our implementation required a method to enforce that the packet is indeed sent to the next path set entity. Ideally, this needs to be implemented in the actual Linux kernel routing. However, for the sake of simplicity and for the prototype implementation the adaptive routing has been implemented as follows: (1) The original destination IP address is stored in the scheme header. (2) Each path set entity replaces the current IP destination address by the address of the next path set entity evaluated in the adaptive routing step. Note that this is done before the actual

routing takes place in the Linux network stack. (3) At the very end when the scheme header is removed, the original destination IP address is restored and the packet is forwarded in the original state before the adaptive source authentication scheme has been applied. (4) To make the actual adaptive routing step each path set entity uses the kernel DB module to lookup the routing information of the path set entity from highest to lowest priority. For the sake of simplicity in the prototype implementation, the available routing information in the prototype implementation is limited to the locally available links. As mentioned above, the routing information is made available through the kernel DB module which receives the information from the nfqueue daemon. Note that our adaptive routing method including source authentication and additional routing information lookup is efficient in terms of clock cycles as described in Chapter 5.

4.2.6 Perl and Linux Kernel Cryptography Implementations

For the session setup Perl script, we used the Perl libraries `Crypt::Digest::SHA1`, `Crypt::CBC`. The Perl library `Crypt::RSA` was by orders of magnitudes slower than an alternative module which wraps `OpenSSL` [13]. For this reason, we used the Perl wrapper libraries `Crypt::OpenSSL::RSA`, `Crypt::OpenSSL::Random` and `Crypt::OpenSSL::AES`. To provide fast and storage efficient signatures, we used the *Ed25519* public-key signature system. Among others, Ed25519 provides fast single-signature and batch verification, fast signing and small signatures. Moreover, it protects against software side-channel attacks and provides a high security level corresponding to roughly 3000 bit RSA keys [2] [24].

For the Linux kernel target module, we used the Linux kernel crypto API. These provide a generic interface to various hardware and software implemented cryptography primitives. On the systems of the distributed firewall systems, efficient AESni implementation was used to derive the key for the session and to create and verify the verification values for source authentication. The DataHash has been computed with SHA1-SSE3 [6] since unfortunately we did not have an efficient PMAC implementation available. The DataHash has been computed over the whole packet from and including the meta information field MetaInf in the scheme header until the end of the packet. In addition, checksum fields of the transport layer protocols TCP and UDP had to be zero padded before hashing and then later restored as these fields were updated in later rules as part of the change of the IP header fields as described before. Obviously, hashing the whole packet creates a huge overhead at the source and destination and could be optimized, e.g., with an efficient PMAC implementation.

4.2.7 Path Indicator Bit-Representation

In the following, we outline the path indicator bit-representation which is created in the session setup script and passed to the kernel DB. Recall that we use 4 bits for each entity label. The entity labels are encoded naturally as a four bit number, e.g., 0101 for the label or Position 5. Therefore, the bit-representation of the path indicator is simply the encoding of the labels as bits ordered by the priorities of the successors. Note that the path set entity label assignment is given implicitly by the order of the path set and the arrangements of the scheme header fields.

4.2.8 Adapted Protocol Header

In the following, we present the adapted scheme header used for the actual protocol implementation. The additional fields are Old Protocol to store the value of the original IP protocol field and Original Destination Address to store the original IP destination address of the intended recipient. The updated scheme header is depicted in Figure 4.4. In our prototype, we fix the MTL field to one such that the maximum meta-information is limited by 32 bits and leave the extension as future work. Note that the MTL, Position and n fields are updated to 8 bits and the Old Protocol is added which replace the Padding from the scheme header in Chapter 3 for cleaner implementation

in the Linux kernel. The header size increases by fixed 4B and is given for n path set entities with $(9 + 2n) \cdot 32$ bits. Note that this has no influence on the scheme as described in Chapter 3.

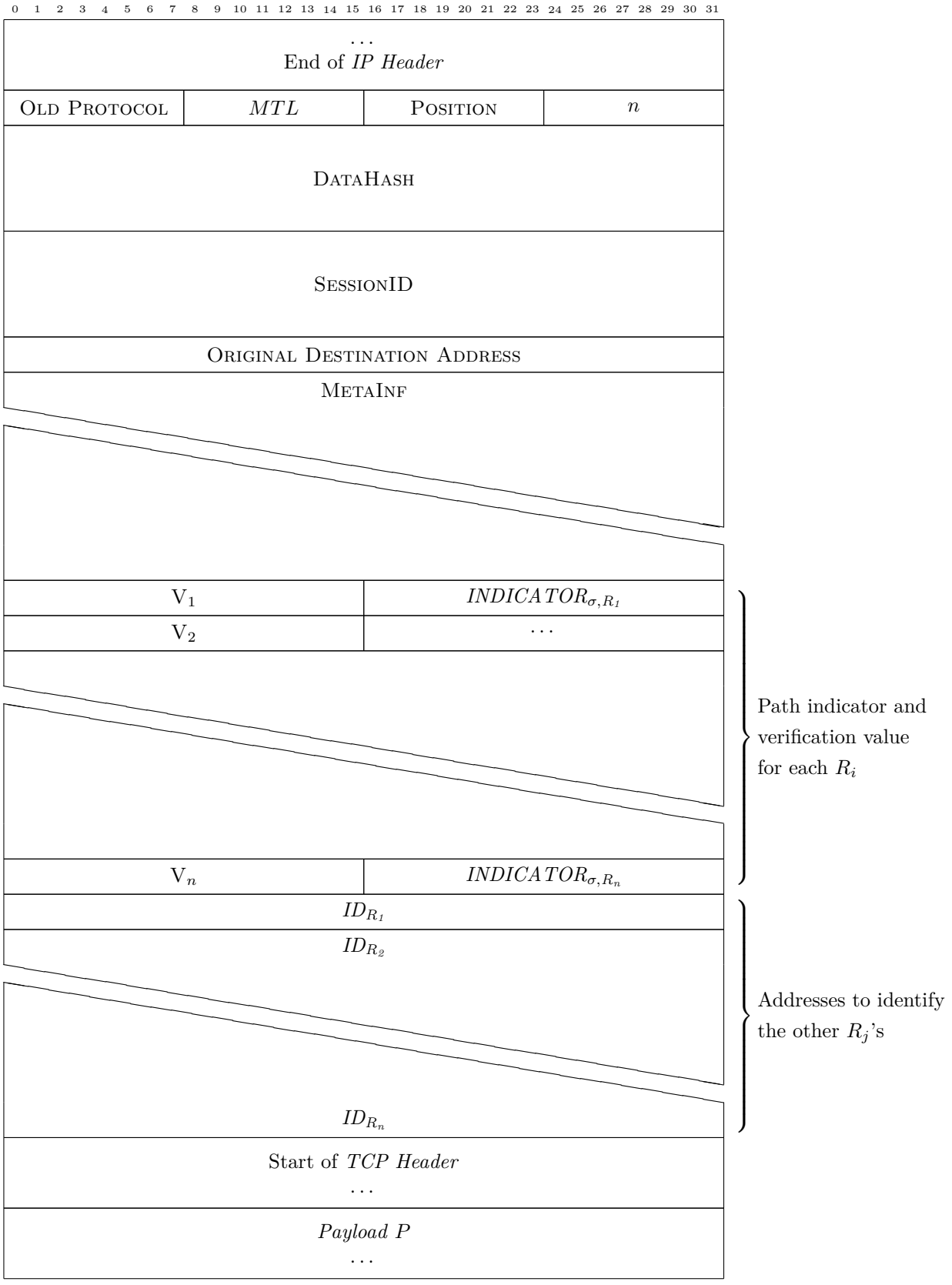


Figure 4.4: Protocol header of the source and data authentication scheme implementation. n is the number of node entities R_j.

5

Evaluation

In this Chapter we introduce the methods for the evaluation in Section 5.1, describe the evaluation setups in Section 5.2 and present and discuss the results in Section 5.3.

5.1 Methods

To evaluate the efficiency of our *distributed zone-based firewall* prototype, we perform four different measurements. First of all, we measure the time of the key exchange protocol execution for a network with 5 path set entities. Secondly, we evaluate the scheme in a network with 5 path set entities and compare its goodput and throughput to the system's baseline performance without our scheme. In addition, we verify if the adaptive routing scheme and the source authentication itself work under the system's maximum throughput. In addition, we perform explicit throughput and goodput measurements to compare the adaptive source authentication scheme against the system's baseline performance without our scheme in a network with 2 path set entities. Finally, we measure the CPU clock cycles of specific parts in the `iptables` target module.

We measure the time of the key exchange with the help of the linux `time` command which returns the real elapsed time between command invocation and termination [19]. The command does not guarantee atomic measurement of the command's execution time as the system can execute arbitrary other code in between, e.g., through scheduling [18]. Therefore, we measure the session setup script for the given network with 5 path set entities one hundred times with the command `time ./sessionSetup.pm -dk -destination 10.0.2.50` and compute the median, mean and standard deviation of the measurements. Unfortunately, due to time limitations we leave the evaluation of calculation overhead at the source, destination and intermediate entities for future work.

To send and receive packets as well as to read the measurements, we used the *CT50310G HighSpeed LANforgeFIRE Traffic Generator* and its provided GUI [1]. We performed simple measurements with UDP packets from source to destination. We perform measurements with packets of the following payload sizes: 36B, 128B, 256B, 576B, 768B, 1024B and 1338B in the network with 5 path set entities and 1362B for the network with 2 path set entities. The minimum payload size depends on the minimum *ethernet framesize* of 64B which results in a 36B payload besides the 20B IP and 8B UDP header as reported by the traffic generator [3]. The maximal payload size depends on the size of the scheme header as well as on the size of the GRE and IPsec header which also depends on padding for the encryption. It is discussed below in more detail. We evaluate the systems as follows. First, the basic sending rate is evaluated at which the baseline without our scheme did not incur packet loss. Afterwards, the system is measured once without and once with the adaptive source authentication scheme activated. In addition, we make measurements of the adaptive routing scheme in case of a link failure for a packet size of 1338B to see if the maximal reachable throughput and goodput can be

achieved even with adaptive routing. Unfortunately, due to time limitations we leave measurements for the adaptive routing with other packet sizes for future work. In this case, traffic is sent on the shortest path until the measurement values are stable, afterwards, a failure of the unreliable link is simulated by disabling the corresponding IPsec tunnel. The adaptive routing scheme will reroute the traffic as soon as the routing information has been propagated by `OSPF` and the local user space daemon has updated the adaptive routing database in the kernel. Note that we do not measure convergence times for the adaptive routing information as these strongly depend on the `OSPF` configuration. More information about OSPF with regard to our system is described in Section 4.2.3.

The goodput values are derived and reported directly from the traffic generator feedback as endpoint to endpoint goodput in mega bits per seconds (Mbps). The throughput values we calculate as follows: (1) with the basic goodput sending rate and the packet payload size we calculate the number of packets sent per second. In addition, we observe the actual real total packet sizes in the network which additionally have a GRE, IPsec and IP header. We do not consider the link layer header for throughput calculations. Afterwards, we calculate the throughput as total packet size times number of packets per second for each case. We give an example later in Section 5.3. The scheme header is 76 Bytes for the network with 5 path set entities and 52 Bytes for the network with 2. By observation, we evaluate the maximum payload size for each network setting such that the resulting packet in the network is 1500B with IGRE, IPsec and additional IP header. For the setting with 5 path set entities the maximum payload size is 1338B and for the setting with 2 path set entities 1362B. Notice that for this maximal payload sizes the difference of total packet size is given by the different size of the scheme header. The packet observations are performed with the command `tcpdump` to sniff the packets at the tunnel interfaces, at the ethernet interfaces and at the traffic generator input interface. Our test machines have a *Intel Xeon E3-1275 @ 3.40GHz* CPU, a *Intel C206* chipset, *82583V* NIC chip and `16`GB RAM. The various implementations used for cryptography are described in Section 4.2.6.

The clock cycles are read by injecting the `RDTSC` assembler instruction within the kernel module to read the *Time Stamp Counter (TSC)* which is the number of clock cycles since reset [17]. In more detail, we calculate and report the difference between start and end of the TSC value between the below specified code sections. As we cannot enforce sequential atomic execution of our code, the CPU might decide to execute arbitrary other code within the execution of our functions. Therefore, depending on the case we measure at least 250 values and calculate mean, median and standard deviation of the measurements. We measure the clock cycles of the following operations in the kernel target module which are described in more detail below.

- Full target to set the scheme header.
- Full target to remove the scheme header.
- Full, part and core update of the scheme header at an intermediate entity. The *core update* includes the verification and adaptive routing step. We analyze a case where the first path indicator field is used to identify the next path set entity and a second case where the second field of the path indicator is used to identify the next path set entity.
- Full and part update of the scheme header at the destination entity.

In more detail, the assembler instruction we inject is as follows.

```
__asm__ volatile("rdtsc" : "=a" (lo), "=d" (hi));
```

It pushes the 64 bit TSC value of the CPU into the processor registers EDX:EAX [17]. The value is then converted in C and used as a 64 bit value.

5.2 Evaluation Setups

The network setup for the key exchange time measurement and the adaptive routing evaluation with 5 path set entities is given in Figure 5.1. The resulting path indicator for the scheme is depicted in Figure 5.2. Unfortunately, due to hardware limitations the setup is connected through an intermediate additional router, which connects all the entities S, A, B, C and D with each other with one interface for all entities. This clearly limits the systems maximum throughput since the traffic has to be routed three times through this intermediate bottleneck router. The intermediate bottleneck router has a maximum throughput of 1Gbps per interface. As the traffic passes the intermediate bottleneck router three times on the shortest path, the maximum observed throughput is roughly $\frac{1}{3}\cdot$ of 1Gbps or $322Mbps$.

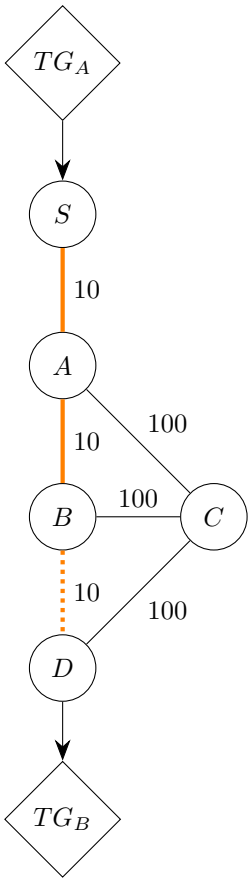


Figure 5.1: The evaluation setting used with the path set entities S, A, B, C, D. TG_A denotes the ingoing connection and TG_B the outgoing connection of the traffic generator indicated by diamond shaped nodes. The dotted link between B and D denotes the unreliable link. The link costs are given next to the link and represent an arbitrary cost configured in OSPF. These costs are only used by the source to calculate the path set.

The network setup for the throughput evaluation with 2 path set entities is given in Figure 5.3.

In both network settings, the systems of the distributed firewall were configured with `iptables` rules as described in Chapter 4. In particular, rules add the scheme header at the incoming interface at the source which is connected to the traffic generator and remove the header at the destination before the packet leaves the VPN domain. In addition in the adaptive routing setting, the intermediate nodes update the scheme header, i.e., perform the verification and adaptive routing step. In addition, when evaluating the adaptive source authentication scheme, the user space daemon is running. To test the

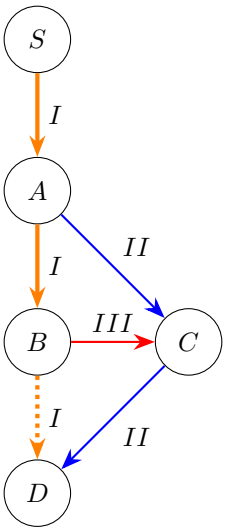


Figure 5.2: The path indicator for the adaptive routing evaluation setting from Figure 5.1.

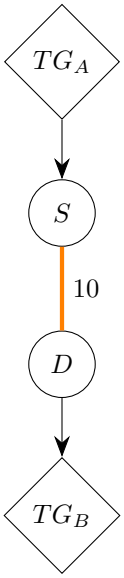


Figure 5.3: The evaluation setting used for the throughput measurements. TG_A denotes the ingoing connection and TG_B the outgoing connection of the traffic generator. The link costs are given next to the link and represent an arbitrary cost configured in OSPF. These costs are only used by the source to calculate the path set.

adaptive routing, we deactivate the link, i.e., IPsec tunnel of the unreliable link from B to D to test the support link from B to C and then C to D. To test the backup path, we simulate the propagation of the adaptive routing information in the sense that we artificially deactivate the link between A and B such that traffic gets adaptively routed at A to C and then to D.

5.3 Evaluation Results and Discussion

The result of the session setup script latency is depicted in Table 5.1. We see that the session setup

Case	Median	Mean	Standard Deviation
Session Setup Latency	348ms	371ms	70ms

Table 5.1: Session setup script latency in milliseconds for the network form Figure 5.1.

script has a high latency with an average time of 371ms. We compare our results to OPT [29]. The DRKey protocol was measured in packet latency during processing and introduces only $621\mu s$ at the source and $3820\mu s$ at the destination for a path length of 2 (excluding source and destination). Moreover, the latency is $381\mu s$ at an intermediate router independent of the path length. Note that we measure the time used in total for all participating entities differently and can therefore not directly compare the values. However, we can estimate the total time needed for OPT for a total path length of 4 (including source and destination) as in our case with $621\mu s + 3820\mu s + 2 \cdot 381\mu s = 5203\mu s$. Clearly, OPT is orders of magnitude faster than our Perl implementation. However, we argue that the specific implementation in Perl introduces overhead which is not optimized for really fast execution as our protocol itself is adapted from [29]. In particular, note that our Perl implementation derives a RSA key which could be loaded instead to further speed up the implementation. The prototype session setup performance has to be increased to ensure that packets can be sent to the nfqueue daemon, buffered and reinjected while not disrupting a connection. In our case for UDP packets, the traffic generator did receive the packets correctly which shows that the concept is feasible if the latency is optimized. Remember that this session setup has to be done only rarely for the first packet of a scheme source and destination pair and can be done while an old session is still used.

The observed goodput, packet sizes and the calculated throughput for the baseline and the scheme for both evaluation settings are depicted in Table 5.2. We give an example of a throughput calculation. Consider the values for payload 256B in the adaptive routing network with 5 path set entities for the scheme. The throughput is given by the number of packets times the total packet size $\frac{128Mbps}{256B} \cdot 420B = 210Mbps$.

The throughput and goodput results for the adaptive routing evaluation network are illustrated in Figure 5.4. In this first measurements, traffic was adaptively routed on the default shortest path within the path set. The throughput and goodput results for the throughput evaluation network are illustrated in Figure 5.5 As can be seen in the Figures, the scheme incurs overhead which is most likely limited by the protocol implementation in the Linux kernel netfilter environment which is investigated more in detail below with the TSC value evaluation. We compare our results to OPT [29]. OPT achieves throughput close to 40 Gbps in the best case. In our case, for the throughput evaluation setting with 2 path set entities the scheme achieves a maximal throughput of 569Mbps, goodput 517Mbps for a baseline with throughput 728Mbps, goodput 685Mbps. In the adaptive routing evaluation setting with 5 path set entities, the scheme achieves similar throughout and goodput values as the baseline system. Note that the setting in the OPT evaluation differs from our setting and can therefore not be directly compared. The most important difference is that we did not directly measure the routing forwarding overhead but the overall whole execution of the adaptive source authentication scheme with the operations of adding the header at the source and removing the header at the destination as implied by the prototype implementation of the *distributed zone-based firewall*. On

Payload	TG interface	IPsec Tunnel	Full Packet	Goodput	Throughput
Adaptive Routing Network, $n = 5$					
Baseline					
36B	64B	64B	124B	20Mbps	69Mbps
128B	156B	156B	216B	70Mbps	118Mbps
256B	284B	284B	344B	139Mbps	187Mbps
576B	604B	604B	664B	250Mbps	288Mbps
768B	796B	796B	856B	275Mbps	307Mbps
1024B	1052B	1052B	1112B	293Mbps	318Mbps
1338B	1390B	1390B	1448B	303Mbps	322Mbps
Scheme					
36B	64B	140B	200B	18Mbps	100Mbps
128B	156B	232B	292B	65Mbps	148Mbps
256B	284B	360B	420B	128Mbps	210Mbps
576B	604B	680B	740B	220Mbps	283Mbps
768B	796B	872B	932B	245Mbps	297Mbps
1024B	1052B	1128B	1188B	261Mbps	303Mbps
1338B	1390B	1442B	1500B	282Mbps	316Mbps
Support Link Adaptively Taken					
1338B	1390B	1442B	1500B	171Mbps	191Mbps
Backup Link Adaptively Taken					
1338B	1390B	1442B	1500B	281Mbps	315Mbps
Throughput Network, $n = 2$					
Baseline					
36B	64B	64B	124B	19Mbps	65Mbps
128B	156B	156B	216B	67Mbps	115Mbps
256B	284B	284B	344B	135Mbps	181Mbps
576B	604B	604B	664B	295Mbps	340Mbps
768B	796B	796B	856B	355Mbps	396Mbps
1024B	1052B	1052B	1112B	498Mbps	540Mbps
1338B	1390B	1390B	1448B	685Mbps	728Mbps
Scheme					
36B	64B	116B	176B	15Mbps	73Mbps
128B	156B	208B	268B	52Mbps	109Mbps
256B	284B	336B	396B	101Mbps	156Mbps
576B	604B	656B	716B	213Mbps	265Mbps
768B	796B	848B	908B	279Mbps	330Mbps
1024B	1052B	1104B	1164B	356Mbps	405Mbps
1338B	1390B	1442B	1500B	517Mbps	569Mbps

Table 5.2: Observed packet sizes, goodput and calculated throughput in the evaluation settings.

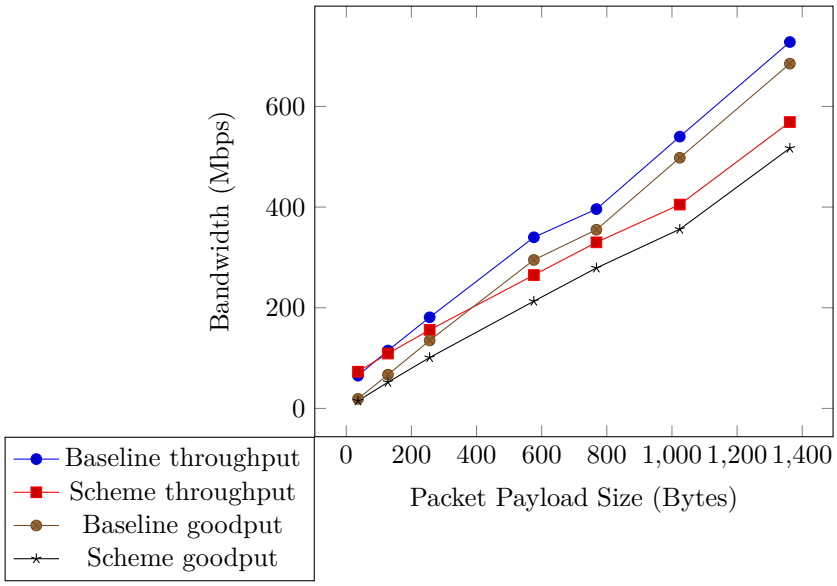


Figure 5.4: Throughput and goodput of the adaptive source authentication scheme and the baseline system in the throughput evaluation setup from Figure 5.3.

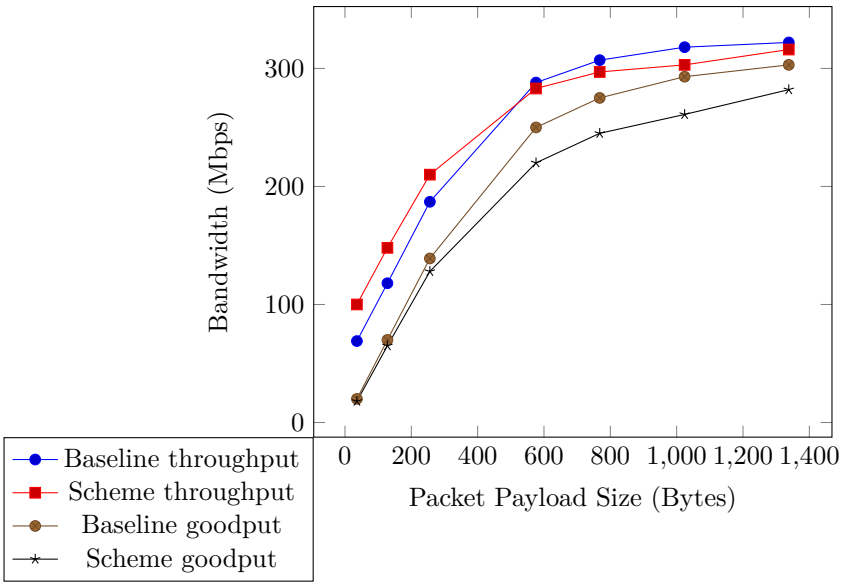


Figure 5.5: Throughput and goodput of the adaptive source authentication scheme and the baseline system in the adaptive routing evaluation setup from Figure 5.1.

the one hand, we argue that our implementation is slower due to the additional overhead incurred by adding and removing the header at the source and destination. In particular, for the biggest packet size in the throughput evaluation setting both, the source and destination, need to compute the DataHash over the whole data packet. In addition, the aforementioned limitations of the prototype database implementation which has been designed for correctness slows down the prototype implementation. Clearly, this leaves room for a more efficient prototype implementation as described in Chapter 6. Note that the differences in the routing evaluation setting with 5 path set entities might also be due to the bottleneck intermediate router as during our measurements it was visible that the overall system's throughput has been exceeded as could be seen by packet drop rates of 8% to 12%. In addition, for packet sizes which did not result in a too high throughput for the network, such behavior was not observed, showing that the scheme itself is correct.

The throughput and goodput results of the adaptive routing are depicted in Table 5.2. As can be seen from the values, the case in which the support link is taken significantly slows down the throughput and goodput because the aforementioned intermediate router is traversed an additional time since the path adaptively taken has length 5 instead of the default path with length 4. However, as can be seen from the measurements, the back up path case has only a minor lower throughput and goodput compared to the case when the default shortest path is adaptively taken. The behavior of the adaptive routing is as expected. When the link breaks, there is no traffic until the adaptive routing information has been updated by the nfqueue daemon and the packet is afterwards successfully adaptively routed further according to the path indicator. The reason for the delay is that in our prototype implementation the link failure detection is implemented through parsing information from OSPF as explained in Section 4.2.4 which introduces a convergence time until detection. We argue that the convergence time can be reduced and the link failure detection can be improved in future work as described more in Chapter 6.

The measurements of the TSC values of the target module is given in Table 5.3. We measured the following cases

- Full target to set the scheme header, indicated by *Full Set Header*.
- Full target to remove the scheme header, indicated by *Full Remove Header*.
- Full, part and core update of the scheme header at an intermediate entity, indicated by *Full, Part or Core Adaptive Routing (yx)* where the y indicates which successor in the path indicator field has been used to send the packet to next. For instance, 1x indicates the first successor field has been used while 2x indicates that the first successor field has been looked up and the second field has been successfully used.
- Full and part update of the scheme header at the destination entity, indicated by *Full or Part Adaptive Routing (destination)*.

The RDTSC values further support our statement from before that the header injection is the limiting factor and the most expensive operation. In addition, it shows that the actual core adaptive routing step is efficiently implemented with clock cycles in orders of 120 to 220 cycles despite the rather not optimized database lookup. However, note that most measurements over the whole `iptables` modules, indicated by *full*, have a high standard deviation of roughly 25%.

Case	Median	Mean	Standard Deviation
Full Remove Header	928	970	150
Full Set Header	20344	22432	4654
Core Adaptive Routing (1x)	112	121	25
Part Adaptive Routing (1x)	5548	5845	1286
Full Adaptive Routing (1x)	10368	10861	1623
Core Adaptive Routing (2x)	196	219	97
Part Adaptive Routing (2x)	6980	7585	2540
Full Adaptive Routing (2x)	13100	14282	4648
Part Adaptive Routing (destination)	13442	14019	1854
Full Adaptive Routing (destination)	18894	20061	3342

Table 5.3: TSC measurements results rounded to integers given in clock cycles.

6

Future Work

In this Chapter, we state future work on the one hand for the implementation in Section 6.1 and on the other hand for the adaptive routing scheme in Section 6.2.

6.1 Implementation and Method Future Work

A straightforward future work is to optimize and further evaluate the prototype implementation of the Kernel target and match module. As mentioned before in Chapter 4, there are many optimizations which can be performed. In particular, the Kernel database module can be optimized for speed e.g. by concurrent access. Moreover, the target module can be made more efficient by refactoring and optimizing code. In particular, the calculation of the DataHash can be speed up by another more efficient hash function implementation such as PMAC. In the prototype implementation, the size of the meta-information is further limited by 32 bits which can also be easily extended as future work since the design and most of the code are already implemented. In addition, we propose as future work to redesign the target module with the asynchronous Kernel crypto API to further speedup the module. Note that in this way for instance efficient asynchronous CBC-AESni can be for hashing. Anyway, more evaluation tests are necessary for the prototype and especially for the refactored version to evaluate the actual real maximum throughput and goodput of the fully implemented adaptive source authentication scheme.

Another possible future work is to further develop the perl session setup setup and routing script. In particular, the session setup script can be extended to handle concurrent handling of packets to speed up throughput for the session setup. As mentioned before, the adaptive routing mechanism can be extended. Either link failure detection might be read directly from the available information fr om `OSPF` or can be distributed by the path set entities themselves through a gossiping protocol or even another routing protocol. In addition, the method to pass the adaptive routing information to the kernel can be further optimized for (1) efficient data passing to the kernel and (2) enable efficient concurrent access to the other Kernel modules. We think this offers a lot of potential for future work. What is more, the current and optimized session setup script have to be evaluated and tested to make more precise statements about its performance.

Similar to the previously mentioned refinements of the implementation, the user space daemon can be improved in terms of functionality and efficiency. On the one hand, the daemon needs to be extended with a queuing and packet identification system such that for each destination id within the adaptive source authentication scheme only one key exchange protocol is performed and packets which need the same key material are queued. In addition, the daemon can be extended to support concurrent handling of multiple packets and queues which further speeds up the daemon and increases the systems overall throughput in case of many first packets or short session lifetimes. Detailed design

needs to be conducted as future work.

The implemented `iptables` rules can be applied independently of other rules. We see a huge potential for future work in how to apply this new source meta-information authentication primitive in an actual distributed firewall environment or in another firewall based system. In particular, different combinations and applications of the rules and similar implementations are possible. For instance, other meta-information can be used such as identification information, e.g. in form of an user id, or authenticated packet tagging.

6.2 Adaptive Source Authentication Future Work

Our adaptive source authentication scheme can be further developed, especially with regard to the possibilities with application guarantees and the adaptive routing information. In addition, the k-shortest d-path set algorithm can also be analyzed more thoroughly and might yield options for efficiency improvements.

There are many options to investigate about how to infer or distribute the adaptive routing information and to rapidly detect when a link fails. For instance, the adaptive routing information could be distributed between the network entities with the use of a gossiping protocol.

The extension of the scheme for multiple unreliable links can be further analyzed. For instance, it is possible to aggregate unreliable links in a sequence to one link. We call this prozess *unreliable link aggregation*. Afterwards, we can simply apply the scheme from before for one link again. In particular, we identify the first and last node of the aggregated unreliable links. However, there are multiple ways to extend the scheme. For instance, the aggregation can be performed by just deleting all the unreliable links in the diagram. On the other hand, one could aggregate the links such that paths over the entities within the aggregation zone are removed. In addition, the extension of the scheme to multiple unreliable links which are not sequentially connected can be investigated.

The extension of the scheme for multiple path metrics can be further analyzed and refined. Moreover, path metric is only one guarantee out of various options for other application classes as we presented in this work. To the best of our knowledge, the combination of a source authentication protocol with adaptive routing and efficiency improvements, e.g. in form of application guarantees is a new concept. Similar combinations of other protocols might offer various possibilities for future work. In particular, we think that source authentication will be an important network primitive needed in the future.

7

Conclusion

During this master thesis, an adaptive source authentication scheme has been developed. The scheme allows to react to the failure of a single unreliable link through adaptive in-network routing by participating intermediate entities. In addition, the scheme increases performance by taking into account application guarantees in form of multiple path-metric routing optimizations. In addition, we adapted a key exchange protocol from [29] for a set of network entities to efficiently derive keys for a session instead of one fixed path. In particular, our scheme is efficient as it does not need per-source, per-destination or per-flow state at the intermediate entities. What is more, the adaptive routing and source authentication are efficiently performed by intermediate entities. The first prototype implementation showed the feasibility of the whole adaptive source authentication scheme concept. Moreover, it showed that the scheme can be used to solve an applied problem,e.g., to implement rules for policy enforcement. In particular, we showed that these rules can be used to enable new concepts such as the *distributed zone-based firewall*. However, we have also shown that the implementation of such a scheme is hard for a given complex system such as the Linux kernel. The reasons for this are twofold: (1) the complex netfilter Linux kernel packet filtering framework and network stack are hard to extend with the new concept of the adaptive source authentication scheme since a new header has to be added after the actual IP header and (2) the adaptive routing concept is not directly supported by the current Linux Kernel routing implementation as it partly contradicts the present concepts. However, we showed that our scheme concept is implementable in a secure and partly efficient manner. In particular, the main core part of the scheme which are the adaptive routing, e.g. in case of error recovery, and the source authentication verification step by intermediate network entities is efficient as shown by our clock cycle measurements. We think that source authentication will be an important topic in the future as it is a network primitive with crucial applications such as policy enforcement but nevertheless not widely deployed.

Appendix

List of Figures

List of Tables

References

[1] *Ct50310g highspeed lanforgefire traffic generator.* https://www.candelatech.com/pdfs/ct503-10G_product.pdf.

[2] *Ed25519: high-speed high-security signatures.* http://ed25519.cr.yp.to/.

[3] *Ethernet frame, wikipedia.* https://en.wikipedia.org/wiki/Ethernet_frame.

[4] *Generic routing encapsulation, wikipedia.* https://en.wikipedia.org/wiki/Generic_Routing_Encapsulation.

[5] *Graph theory, wikipedia.* https://en.wikipedia.org/wiki/Graph_theory.

[6] *Improving the performance of the secure hash algorithm (sha-1).* https://software.intel.com/en-us/articles/improving-the-performance-of-the-secure-hash-algorithm-1.

[7] *iptables, wikipedia.* https://en.wikipedia.org/wiki/Iptables.

[8] *Ipv4, wikipedia.* https://en.wikipedia.org/wiki/IPv4.

[9] *libnetfilter_queue.* http://netfilter.org/projects/libnetfilter_queue/.

[10] *The linux kernel archives.* https://www.kernel.org/.

[11] *Netfilter, wikipedia.* https://en.wikipedia.org/wiki/Netfilter.

[12] *netfilter/iptables project.* http://www.netfilter.org/.

[13] *Openssl: The open source toolkit for ssl/tls.* https://www.openssl.org/.

[14] *Preimage attack, wikipedia.* https://en.wikipedia.org/wiki/Preimage_attack.

[15] *procfs, wikipedia.* https://en.wikipedia.org/wiki/Procfs.

[16] *Quagga routing suite.* http://www.nongnu.org/quagga/.

[17] *Time stamp counter, wikipedia.* https://en.wikipedia.org/wiki/Time_Stamp_Counter.

[18] *Ubuntu manpage: time - run programs and summarize system resource usage.* http://manpages.ubuntu.com/manpages/maverick/man1/time.1.html.

[19] *Unix man pages : time.* http://unixhelp.ed.ac.uk/CGI/man-cgi?time.

[20] *vtysh(1) - linux man page.* http://linux.die.net/man/1/vtysh.

[21] Aggarwal, Stefan Savage Tom Anderson Amit; Cardwell, David Becker Neal; Snell, Andy Collins Eric Hoffman John and Zahorjan, Amin Vahdat Geoff Voelker John. *Detour: a case for informed internet routing and transport.*

[22] Bellovin, Steven M. *distributed firewalls.* http://static.usenix.org/publications/login/1999-11/features/firewalls.html.

[23] Benvenuti, Christian. *Understanding Linux network internals*. " O'Reilly Media, Inc.", 2006.

[24] Bernstein, Daniel J; Duif, Niels; Lange, Tanja; Schwabe, Peter and Yang, Bo-Yin. *High-speed high-security signatures*. *Journal of Cryptographic Engineering*, volume 2(2):77–89, 2012.

[25] Black, John and Rogaway, Phillip. *A block-cipher mode of operation for parallelizable message authentication*. In *Advances in Cryptology?EUROCRYPT 2002*, pages 384–397. Springer, 2002.

[26] Engelhardt, Jan and Bouliane, Nicolas. *Writing netfilter modules*. *Revised, February*, volume 7, 2011.

[27] Farinacci, D; Li, T; Hanks, S; Meyer, D and Traina, P. *Rfc 2784-generic routing encapsulation (gre)*. *IETF, March 2000*, 2000.

[28] John, John P; Katz-Bassett, Ethan; Krishnamurthy, Arvind; Anderson, Thomas and Venkataramani, Arun. *Consensus routing: The internet as a distributed system*. *Proc. NSDI (April 2008)*, 2008.

[29] Kim, Tiffany Hyun-Jin; Basescu, Cristina; Jia, Limin; Lee, Soo Bum; Hu, Yih-Chun and Perrig, Adrian. *Lightweight Source Authentication and Path Validation*. In *Proceedings of the ACM SIGCOMM*. August 2014.

[30] Lakshminarayanan, Karthik; Caesar, Matthew; Rangan, Murali; Anderson, Tom; Shenker, Scott and Stoica, Ion. *Achieving convergence-free routing using failure-carrying packets*. *ACM SIGCOMM Computer Communication Review*, volume 37(4):241–252, 2007.

[31] Limacher, Lukas. *Implementation for the master thesis work "source meta-information authentication along adaptive network paths for policy enforcement"*. `https://github.com/LukasLimacher/source-authentication`.

[32] Liu, Xin; Yang, Xiaowei; Wetherall, David and Anderson, Thomas. *Efficient and secure source authentication with packet passports*. In *Proceedings of the 2Nd Conference on Steps to Reducing Unwanted Traffic on the Internet - Volume 2*, SRUTI'06, pages 2–2. USENIX Association, Berkeley, CA, USA, 2006. URL `http://dl.acm.org/citation.cfm?id=1251296.1251298`.

[33] Martins, Ernesto QV and Pascoal, Marta MB. *A new implementation of yen?s ranking loopless paths algorithm*. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, volume 1(2):121–133, 2003.

[34] Moy, J. *OSPF Protocol Analysis*. RFC 1245 (Informational), July 1991. URL `http://www.ietf.org/rfc/rfc1245.txt`.

[35] Moy, J. *OSPF Version 2*. RFC 1247 (Draft Standard), July 1991. URL `http://www.ietf.org/rfc/rfc1247.txt`. Obsoleted by RFC 1583, updated by RFC 1349.

[36] Moy, John T. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0201634724.

[37] Naous, Jad; Walfish, Michael; Nicolosi, Antonio; Mazières, David; Miller, Michael and Seehra, Arun. *Verifying and enforcing network paths with icing*. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 30:1–30:12. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-1041-3. doi:10.1145/2079296.2079326. URL `http://doi.acm.org/10.1145/2079296.2079326`.

[38] Perrig, Adrian; Canetti, Ran; Song, Dawn and Tygar, J. D. *Efficient and secure source authentication for multicast*. In *In Network and Distributed System Security Symposium, NDSS ?01*, pages 35–46. 2001.

[39] Qi, Yan. *An implementation of k-shortest path algorithm (cpp version)*. `https://github.com/yan-qi/k-shortest-paths-cpp-version`.

[40] Rogaway, Phillip and Shrimpton, Thomas. *Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance.* In *Fast Software Encryption*, pages 371–388. Springer, 2004.

[41] Savage, Stefan; Collins, Andy; Hoffman, Eric; Snell, John and Anderson, Thomas. *The end-to-end effects of internet path selection.* In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 289–299. ACM, 1999.

[42] Savage, Stefan; Wetherall, David; Karlin, Anna and Anderson, Tom. *Practical network support for ip traceback.* In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 295–306. ACM, New York, NY, USA, 2000. ISBN 1-58113-223-9. doi:10.1145/347059.347560. URL `http://doi.acm.org/10.1145/347059.347560`.

[43] Seth, Sameer and Venkatesulu, M Ajaykumar. *TCP/IP architecture, design and implementation in Linux*, volume 68. John Wiley & Sons, 2009.

[44] Snoeren, Alex C.; Partridge, Craig; Sanchez, Luis A.; Jones, Christine E.; Tchakountio, Fabrice; Kent, Stephen T. and Strayer, W. Timothy. *Hash-based ip traceback. SIGCOMM Comput. Commun. Rev.*, volume 31(4):3–14, August 2001. ISSN 0146-4833. doi:10.1145/964723.383060. URL `http://doi.acm.org/10.1145/964723.383060`.

[45] Song, Dawn Xiaodong and Perrig, Adrian. *Advanced and authenticated marking schemes for ip traceback.* In *INFOCOM*, pages 878–886. 2001. URL `http://dblp.uni-trier.de/db/conf/infocom/infocom2001-2.html#SongP01`.

[46] Yaar, Abraham; Perrig, Adrian and Song, Dawn. *Siff: A stateless internet flow filter to mitigate ddos flooding attacks.* In *In IEEE Symposium on Security and Privacy*, pages 130–143. 2004.

[47] Yen, Jin Y. *Finding the k shortest loopless paths in a network. Management Science*, volume 17(11):712–716, 1971. doi:10.1287/mnsc.17.11.712. URL `http://dx.doi.org/10.1287/mnsc.17.11.712`.

[48] Zhang, Fuyuan; Jia, Limin; Basescu, Cristina; Kim, Tiffany Hyun-Jin; Hu, Yih-Chun and Perrig, Adrian. *Mechanized network origin and path authenticity proofs.* In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 346–357. 2014. doi:10.1145/2660267.2660349. URL `http://doi.acm.org/10.1145/2660267.2660349`.

Index