University of Amsterdam
System & Network Engineering
Research Project

P4 VPN Authentication

Authentication of VPN Traffic on a Network Device with P4

Jeroen Klomp

Supervisors: Ronald van der Pol & Marijke Kaat

July 16, 2016

Abstract

This research explores the authentication of network traffic via P4 in the context of the CoCo VPN service. CoCo is a prototype of a user-initiated multi-domain VPN service that is intended to be used for computationally intensive applications, like eScience. The prototype leverages SDN such that most functionality can be implemented directly in network devices, but supporting authentication in this manner proves challenging. The P4 language provides more flexibility than current SDN techniques by allowing reprogramming of the data plane, possibly enabling secure authentication via SDN.

During the project the feasibility of authenticating network traffic with P4 has been researched and a proof of concept has been built that implements the main aspects. These aspects include distinguishing sessions and protecting against replay attacks. Since the cryptographic means required for a secure authentication scheme are currently not available in P4 and cannot be implemented using the P4 language itself, a message authentication code is simulated.

The result of the proof of concept is that the flexibility of the P4 language shows promise. However, to be implemented in a secure manner cryptographic means need to be supported by the P4-enabled devices. For incorporation into the CoCo VPN service more work is necessary to determine the best way to extend the architecture of the prototype.

Contents

1 Introduction

For the Community Connection (CoCo) project a user-configurable, on-demand virtual private network (VPN) service is being developed. Its aim is to enable user-initiated setup and tear down of multipoint-to-multipoint VPN instances in a multi-domain environment.[1] The use case of the user-initiated VPN service is to facilitate eScience[1] in a secure, highly scalable, fast and very easy to use and support manner.

Currently, the prototype of the VPN service still has a couple of limitations; encryption needs to be provided at the application layer where necessary and authentication is considered to be suboptimal (e.g., provided by a whitelist of the source IP address, VLAN ID or interface). To make the solution more secure it is an option that traffic from the user side be verified for authenticity at the provider edge. This proves to be a problem with the OpenFlow protocol due to inflexibility with regard to the parsers and actions that an OpenFlow switch can support.

A possible solution to the limitations of OpenFlow herein are programmable data planes. An example of a language that makes this possible is P4 (which stands for Programming Protocol-Independent Packet Processors).[2] It allows for reconfiguration of a network device by running P4 programs. This approach enables a higher degree of flexibility than OpenFlow, for instance, it allows matching on arbitrary headers instead of from a predefined set. Through P4 it might be possible for a switch to authenticate VPN traffic.

1.1 Related work

Research has been done on using software defined networking (SDN), and more specifically OpenFlow, for authentication purposes. However, these research projects focus on users authenticating at a centralised system which then uses the OpenFlow protocol to grant access to resources,[3, 4] using policies on top of SDN controllers to ease the management of user authentication[5] or instead focus on securing the OpenFlow protocol and architecture itself.[6]

These research projects assume that after authentication has taken place traffic can be securely matched on switch port, Media Access Control (MAC) address or Internet Protocol (IP) address. This assumption does not hold true for the CoCo VPN use case because the verification needs to be done at the provider which has no control over the network of its customers.

Since its release to the general public in 2014, P4 has been subjected to several case studies in order to assess its capabilities and state. Sivaraman et al. designed a data-center switch[7] and Dang et al. implemented Paxos in P4.[8] The outcomes of these research projects are that P4 shows promise but that there is still a need for improvements in regards to the modularity of code, specification of primitives and the availability of fixed-function blocks (i.e. libraries possibly provided by manufactures). Other remarks were the unintuitive

[1]Computationally intensive research efforts carried out in distributed environments; facilitating a DNA sequencer as a service is one of the use cases.

approach of programming with tables, need for better error handling and the desire for multi-packet processing.

1.2 Research questions

This project revolves around the question “how the CoCo VPN traffic can be authenticated on a network device using the P4 language”. The following sub-questions are used to answer the main question:

- What kind of protocol will provide secure authentication of CoCo VPN traffic?
- What facilities does P4 have to enable the authentication of network traffic?
- How feasible is it to use P4 for authentication of network traffic on a network device?
- How can authentication via P4 be incorporated into the CoCo VPN architecture?

1.3 Report outline

The rest of the report is structured as follows:

- in Chapter 2 the architecture of the CoCo VPN service is explained and an overview of the P4 language is given;
- Chapter 3 lists the requirements of the authentication scheme, researches possible approaches, including algorithms and protocols, and proposes an implementation direction;
- in Chapter 4 a simplified authentication protocol is defined, the protocol is implemented in the P4 and the program is discussed using code snippets;
- Chapter 5 explains how the proof of concept is tested and demonstrates the P4 program;
- in Chapter 6 the gained results and insights are summarised and used for providing recommendations;
- finally, in Chapter 7 and Chapter 8 concluding thoughts are given and possible new topics for research and follow-up tasks are proposed.

2 Background

In this part of the report information about the architecture of the VPN service and its expected authentication use case is presented. Afterwards an overview of the P4 language is given.

2.1 CoCo VPN service

The CoCo VPN service is a site-to-site VPN that spans multiple administrative domains and can be initiated by users themselves. The service uses *CoCo portals* which handle interaction with users and *CoCo agents* to calculate and establish the forwarding paths of the virtual networks. This is done dynamically by reconfiguring the infrastructure within a domain via OpenDaylight SDN controllers, which use the OpenFlow protocol. The Border Gateway Protocol (BGP) is used to distribute the VPN and end-point information between the agents across the different domains and within the core of the network Multiprotocol Label Switching (MPLS) is used to encapsulate and forward the packets over their respective dynamically constructed paths. The way the paths are calculated and constructed is not important during this project, and to not needlessly deviate from the important aspects it is not further addressed. In Figure 2.1 an overview of the VPN architecture is shown.

Of importance is the way users will be identified by the system and how this information can be used for authentication purposes. A high-level overview of the steps that are taken when a user constructs a VPN is as follows: 1) the user uses the CoCo portal to sign-up for the VPN service, 2) cryptographic material is exchanged to provide in the means of confidentiality and integrity, 3) the user authenticates at the CoCo portal to construct virtual networks, and 4) the user's network traffic is authenticated by the VPN service (i.e. the provider) in order to get access to resources on other networks).

Currently, the security measures of the VPN service are still in the design phase. It is envisioned that the cryptographic material will consist of asymmetric keys such that individual users can be distinguished. When a user wishes to get access to remote resources via the VPN service it will first need to contact the CoCo portal (either via a browser or via an application programming interface (API)) so that a (symmetrical) session key can be established. This session key can then be used by a program on the user's device (from now on referred as the client) to mark the traffic destined for the VPN service in such a way that the VPN service is able to verify the authenticity of the traffic using the same key.

At the moment it is not envisioned that the authentication needs to be implemented on an end-to-end basis. Instead, once traffic is verified by the VPN service and has passed additional access checks (is this user allowed to access this VPN instance) the traffic is considered secure and is not subject to further scrutiny at the network level in regard to the VPN service (it may still be checked for other purposes like network management). This implies a trust relationship between the service providers where each promises to verify the authenticity of its users to enable the service to function securely.

This means that there is no explicit session between the client and the network device that checks the authenticity of the network traffic destined for the VPN service. Instead, one

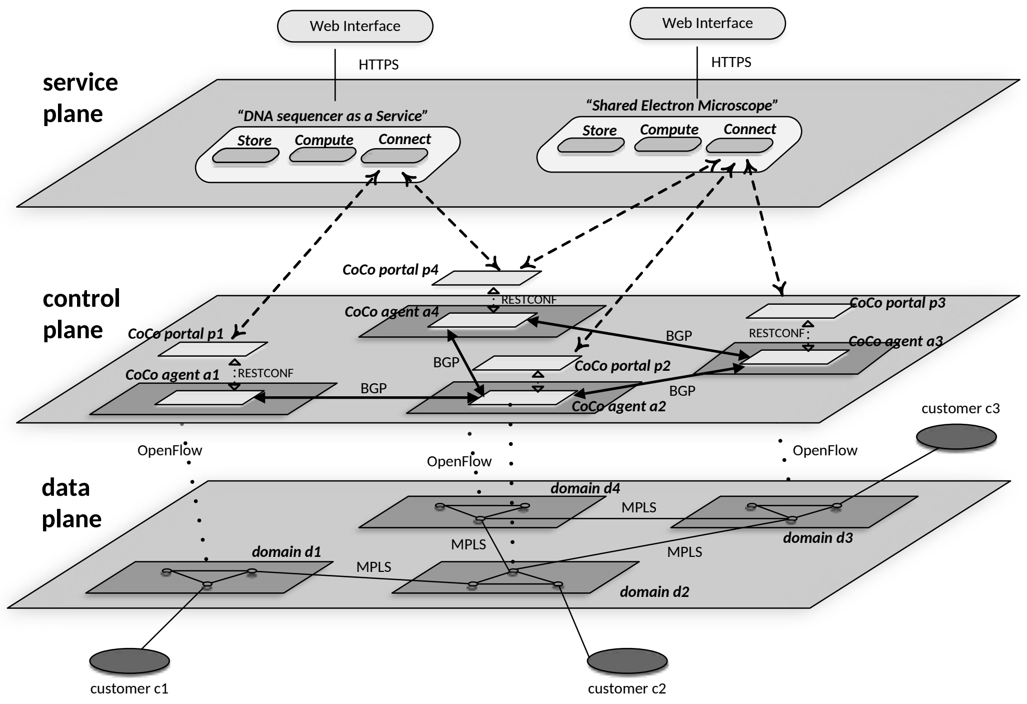

Figure 2.1: CoCo VPN architecture (adapted from Van der Pol et al.)[1]

could see the phase where the session key is established between the client and the CoCo portal as the beginning of the multi-point to multi-point connection. The network device used to verify the authenticity needs to have access to this session key, thus a mechanism that distributes the key from the CoCo agent to the network device is necessary. It is conceivable that the session key will not be valid indefinitely, but instead that after a predetermined amount of time a new session key is established. This will allow more control over access to resources and decrease the chance of malignant use. This means that the key distribution mechanism needs to have a means to either express the validity constraints of the key material or to explicitly invalidate previously distributed material once it expires.

In Figure 2.2 a simple illustration shows the procedures depicted above (the illustration is simplified; only shows a unidirectional flow e.g., the return traffic is not shown but the steps would be the same; and does not necessarily reflect the path the packets take e.g., communication between customer 1's user and the CoCo VPN portal is likely to go through the same provider edge (PE) device that in a later phase verifies the authenticity of the traffic, but this is not necessary).

This research focuses on the authentication of network traffic on a network device but is not concerned with the intricacies of designing all the components necessary for authenticating CoCo users. Therefore, it is assumed that the prerequisites for user authentication (e.g., a system for distinguishing users, creating sessions and exchanging key material) are available.

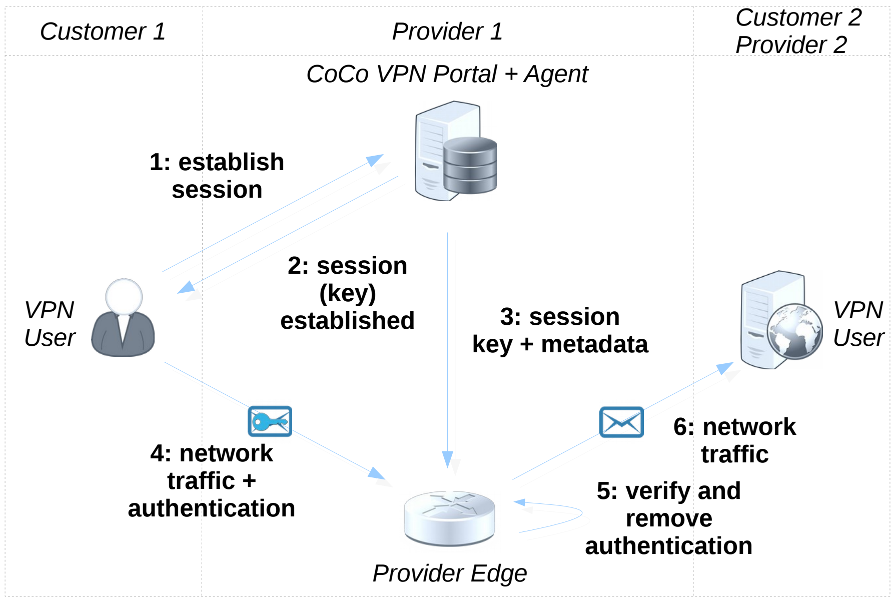

Figure 2.2: Expected CoCo VPN use case

1. The user authenticates at the CoCo VPN service, most likely using a secure channel that involves previously obtained key material. Then an attempt to establish a session is initiated. This likely involves the user stating its intents (e.g., accessing specific resources).
2. The CoCo VPN service verifies the intents of the user (e.g., via an access control list (ACL) access to specific VPN instances or even endpoints is checked). If the checks return positive a session is established which results at least in a session identifier, a shared session key and an algorithm (and possibly other session attributes) used for network authentication.
3. The CoCo VPN service then supplies any network device that serves as an entry point for the VPN service (i.e. the PE) with the necessary information needed to verify the network flows. Possibly, the user information only gets distributed to the PEs that face that user.
4. The user generates traffic destined for a specific endpoint that is reachable via the VPN. The packet is supplied with information such that its integrity and authenticity can be verified and sent directly to the destination. This likely involves using the session key and network packet as input to a digest algorithm and storing the result in the form of an authentication code accompanied with auxiliary information, like the session identifier, inside the packet.
5. The PE identifies packets destined for the service (e.g., via a prefix check or by looking for a specific protocol) and uses the session identifier to lookup the shared key in order to carry out the same cryptographic procedure to verify the authenticity. Since the use case does not involve distributing the session key to other providers or the remote endpoint, the PE removes the authentication information.
6. The PE sends the authenticated network packet without the authentication information via the VPN towards its endpoint.

2.2 P4 overview

As mentioned in the introduction, P4 allows one to program the packet forwarding pipeline of P4-enabled network devices. While the same holds for OpenFlow to a certain extent, P4 allows for much more granularity because it is not restricted to populating a set of well-known tables with entries that match and act on a limited set of protocol header types. It does this by defining an abstract switch model that allows the specifics to be defined by a programmer via P4 applications. The applications are written in an implementation-independent manner and are then compiled to specific P4 targets. These are devices that support or can run P4.[9]

P4 is protocol independent which means a programmer can define its own headers, which the switch running the P4 program will be able to parse. After the parser has extracted the headers as defined by the program, the match and action tables determine the control flow within the pipeline. First the tables are created by the programmer, which defines the header fields used for matching packets and describes the actions that can be taken in case of a hit. Then the tables are populated by entries that hold the value to be matched and once a packet matches, the action listed by that entry will be taken. The tables are reconfigurable at runtime which can be done via an API in an OpenFlow-like fashion, or even via OpenFlow itself (Figure 2.3 shows how OpenFlow and P4 fit into the SDN model).

Figure 2.4 shows the main components of the P4 abstract switch model. Packets are parsed according to the headers described by the program. The result is a parsed representation of the packet that can be used by the P4 program, primarily via match and action tables. These tables can be applied both at the ingress and egress pipeline. There are subtle differences between these stages e.g., in regards to dropping packets and setting the destination (which for brevity will not be elaborated upon in this section). Finally, at the end of the egress pipeline packets are deparsed (i.e. the parsed representation is serialised back to packet format) and put on the wire.

The explanation given in this section only scratches the surface of the P4 forwarding model and its capabilities. P4 is specified in the P4 Language Specification which goes into much more detail and can be consulted for more information.[10] The language is still in development and currently hardware support for P4 is in its early stages. However, there is a software implementation, which was used during the project. In Section 4.2 the way the P4 program is constructed is described and during the process more details about the software switch and the P4 language are given.

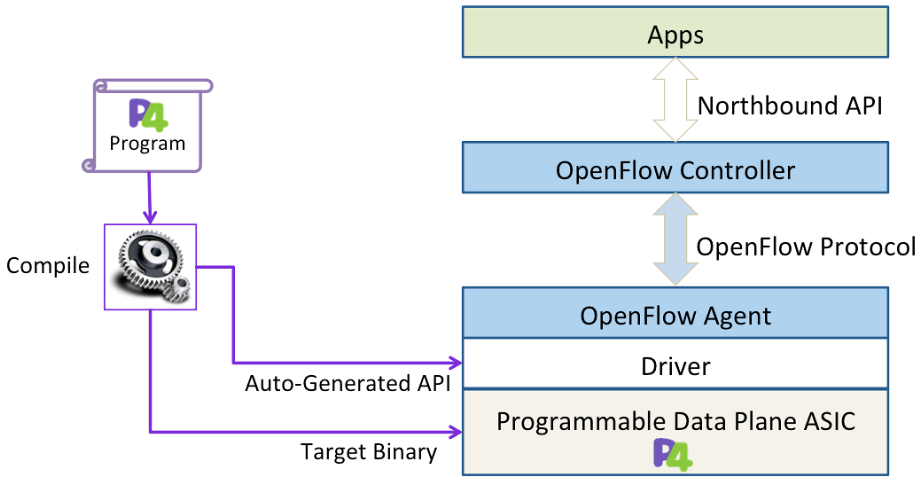

Figure 2.3: P4 in relation to OpenFlow (adapted from the P4 Language Consortium)[9]

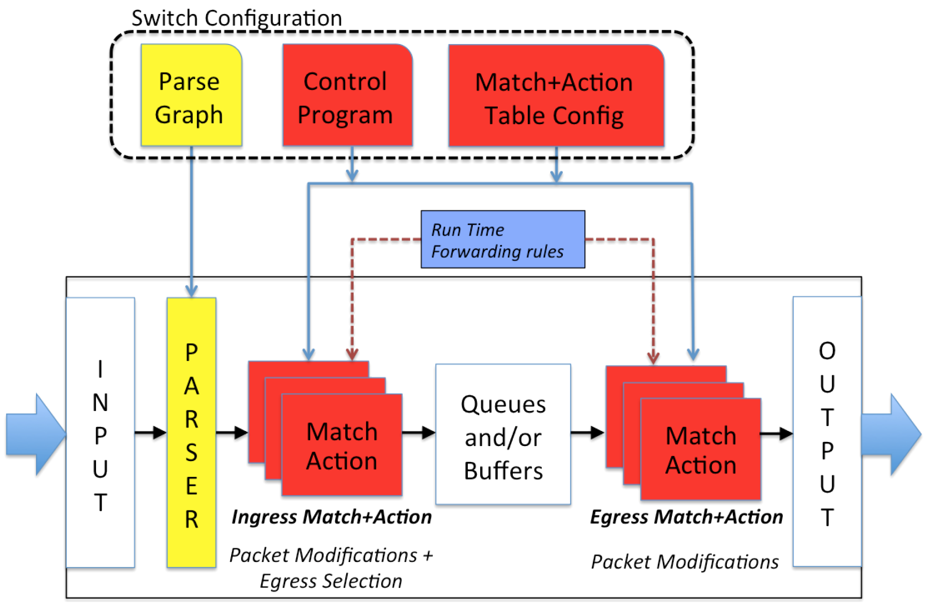

Figure 2.4: P4 abstract forwarding model[10, p. 6]

3 Authentication protocol

In this chapter the requirements for network authentication are researched and possible authentication schemes and protocols to be incorporated into the CoCo VPN service are explored.

3.1 Protocol requirements

A protocol used for authentication of network traffic provides protection against forgery of packets. This means a recipient can be sure the packet originates from the sender, and that the packet has not been altered by an intermediary (i.e. its integrity can be verified). In this section the requirements for such a protocol are described in a high-level manner.

Message authentication is typically done in roughly two different ways: using message authentication codes (MACs) or via digital signatures. Because signatures use prohibitively slow asymmetric cryptography they are not used for authentication of network traffic, instead MACs (using symmetric cryptography) are widely deployed for authentication of network traffic.[11, p. 10] The MAC algorithm is used with a key and the message as input and results in information, which the receiver of the message verifies by carrying out the same operation and comparing the received value with its own calculated value. The algorithm can involve relatively costly computations (in regard to generic network traffic) in order to guarantee sufficient security properties and results in additional bits to be transferred. Costly computations and increase of packet overhead could degrade network performance and possibly even create opportunities for a Denial of Service (DoS). Therefore, the used algorithm should provide sufficient protection against forgery while keeping the computational and packet overhead to a minimum. In Section 3.2 more information about MAC algorithms is given and recommendations for suitable algorithms in regard to the CoCo VPN service are made.

Once a method for establishing the authenticity of messages is available and a forger is no longer able to spoof messages, one more important provision needs to be made: forgers must not be able to retransmit authentic messages without this being noticed, i.e. the receiver needs to be able to detect replay attacks. This can be solved by including information inside the message which ensures it can be only sent once e.g., a unique number, or by using a per-packet key. In case of network traffic authentication this can be provided by including a sequence number or a timestamp inside the packet. These values are constructed to monotonically increase and receivers, that keep track of the values, only accept authenticated packets if the value is greater than the previous. Since in practice values are finite and network protocols are constrained in terms of field width, one needs to account for the situation where the anti-replay value overflows and possibly wraps-around to a lesser value. When this behaviour is left unspecified the receiver might either discard valid packets or accept invalid (old) packets. These considerations demand a detailed specification of the rules implementations should follow. In Section 3.3 an authentication protocol, that governs authentication of network packets, is studied so that important aspects are identified and possibly a candidate for the VPN service can be selected.

Another requirement is that ideally no new possibilities for tracking users are introduced. This suggests that the session identifiers be rotated regularly. Since this information is different for every new session, is expected to be only of local significance (meaning between a customer and its provider) and is removed before it enters the paths between the providers that comprise the VPN, the impact on users privacy is ought to be low. Still it would be prudent to regularly refresh this information. Examples of how this could be implemented is that the session and related key material are pre-emptively invalidated (e.g., using a timer) or explicitly removed once they expire. In Section 3.4 a decision is made on how session and key expiration should be implemented.

3.2 Authentication algorithm overview

In this section a non-exhaustive overview of possible message authentication algorithms is given such that a preliminary selection can be made for possible incorporation into the P4 CoCo VPN authentication solution.

As stated in Section 3.1, MACs are used for authentication of network traffic. MACs can be subdivided into authentication codes that are built from block ciphers and authentication codes that leverage hash functions. The former uses a secret key to encrypt a fixed-length representation of the original message with a symmetric encryption algorithm. The latter uses a secret key mixed with the original message as input to a hash function.

There are several methods for mixing the key with the message, a secure construction known as hash-based message authentication code (HMAC) is typically used. HMACs are described in RFC 2104 and provide a way to use readily available cryptographic hash functions in combination with a secret key to provide secure message authentication, while incurring only a minor performance degradation over the original hash function.[12] Keys should be at least as long as the hash digest output while the resulting MAC can be truncated to decrease the overhead the MAC incurs. A typical recommendation is to truncate the output to at most half the output size of the algorithm (a value that stems from the birthday attack), while ensuring enough bits are available to keep the probability of collisions low.[12, p. 5]

The following algorithms are typically used for HMACs: MD5, and the Secure Hash Algorithm (SHA) 1 and 2. MD5 and SHA-1 are deprecated or in the process of being deprecated when used for providing signatures to certificates. However, in terms of message authentication algorithms they are still considered to provide a sufficient level of security. While the vulnerabilities do not affect MACs, or at least to lesser extent, it is discouraged to include these weaker algorithm in newly designed protocols since better alternatives are available.[13, p. 4] However, since the application of the CoCo VPN features very high bandwidths and requires implementation and computation in embedded devices, a compromise in terms of security strength might be deemed acceptable.

As stated before, besides using HMACs block cipher algorithms can too be employed to provide message authentication. RFC 3566 describes `AES-XCBC-MAC` which solves the forgery problems of using Advanced Encryption Standard (AES) in cipher block chaining (CBC) mode that arise when messages of varying lengths are authenticated.[14] Another block cipher mode of operation suitable for MAC purposes is GMAC. GMAC uses Galois/Counter mode (GCM) to provide an efficient and high performance MAC via parallelisation. Its low latency and operational overhead make it suitable for protecting network traffic against forgery.[15, 16]

More recently Poly1305 has been standardised in RFC 7539 and has been incorporated into browsers and OpenSSH. It provides high-speed message authentication and is relatively straightforward to implement. It can be used in concordance with ChaCha20 to form an alternative to AES while offering better performance in absence of hardware acceleration.[17, 18]

Another possibility is SipHash. SipHash is a relatively new hash function which differentiates itself from other hash function due to its low complexity, high performance and optimisation for short messages.[19] Other hash functions tend to be optimised for longer messages which incurs a computational overhead on the relatively large amount of short messages network traffic is composed of. Furthermore, SipHash does not require multiple keys, it does not rely on AES and no nonces need to be added to messages. The low-cost computations result in an efficient hash function that promises to be hash-flood resistant while still providing reasonable randomness.[20] These properties make it particularly suitable as a function for software hash structures. An obligatory note to make is that SipHash is not meant to be collision-resistant (which means it has less desirable properties for signatory purposes, yet promises to be secure enough for MAC purposes under the assumption of a secret key of sufficient length).[21]

In Table 3.1 an overview of (some of the variants of) the aforementioned algorithms is given. Where applicable operation in HMAC mode is assumed.

Algorithm	**Hash size**	**Truncated size**[a]	**Nonce size**	**Total size**
MD5	128 bit	64 bit	-	64 bit 8 byte
SHA-1	160 bit	80 bit	-	80 bit 10 byte
SHA-256	256 bit	128 bit	-	128 bit 16 byte
SHA-512	512 bit	256 bit	-	256 bit 32 byte
AES	128 bit	-	-	128 bit 16 byte
Poly1305	128 bit[b]	-	128 bit	256 bit 32 bytes
SipHash	64 bit	-	-	64 bit 8 byte

Table 3.1: Overview of possible authentication algorithms

[a]standardised/advised truncation size
[b]requires two 128-bit keys

In terms of overhead, efficiency and implementation effort SipHash appears to be the best choice. The security properties are not of the same level as SHA, but likely to be sufficient for the CoCo use case which requires authentication only between the customer and provider. Its hash-flood resistance makes it easier to incorporate into a network device whereas otherwise dedicated circuits or fail-safe mechanisms need to be provided to surmount possible DoS attacks. However, it is unlikely that the algorithm is high on the list of cryptographic algorithms to be implemented by vendors that develop P4-enabled devices; P4 does not have facilities for defining authentication algorithms in P4 itself,[10, p. 33] instead these functions need to be provided by external objects via standardised or target-specific libraries.[10, p. 33] It is more likely that algorithms like SHA and AES will find their way to P4 targets, possibly facilitated by dedicated functions in hardware, since these algorithms are standardised by a well-known institute, are considered secure and are widely deployed.

In case it is not possible to use an algorithm that is impervious to flood attacks and the hardware is not capable of providing dedicated functions for message authentication, setting a safe upper bound to the computation could be attempted (e.g., by limiting the amount of VPN sessions, traffic or endpoints). P4 is capable of metering traffic, but this will make the P4 program more complex and depending on the implementation (individual) users could be subjected to a DoS.[10, p. 46] Furthermore, it could pose difficult to determine the value at which traffic needs to be discarded since this is likely to be target and model dependent. Nevertheless, counting the amount of authentication packets is advisable since it gives insight into the usage of the service. Like metering, P4 has facilities for keeping traffic counters (on a packet and/or byte level), how these are reported back to a controller is not part of the specification. It is conceivable that a runtime-API, like OpenFlow, will provide the means to either pull or push these values to the controller at definable intervals.

3.3 Authentication protocol overview

Once one or multiple MAC algorithms are available, a method is required to put the MAC on the wire in a standardised manner. In the next sections a glimpse into one of such protocols, Internet Protocol Security (IPsec), is given. It is by no means meant to be an exhaustive study of protocol; only the relevant aspects for the CoCo VPN service are taken into account.

3.3.1 IPsec overview

At the network layer security can be provided by Internet Protocol Security (IPsec). It is a comprehensive solution to transparently offer authentication and encryption for applications. IPsec is considered to be complex and over-engineered, which mainly stems from the comprehensive Internet Key Exchange (IKE) key exchange protocol.[22, p. 360, 23, p. 1] There are also reports of incompatibility, which could be attributed to the complexity of the protocol.[24]

3.3.2 Authentication Header and Encapsulating Security Payload

As mentioned earlier IPsec provides in authentication and encryption of IP traffic. Once an IKE Security Association (SA) is established — a session is created and cryptographic material has been exchanged — authentication can be provided by the Authentication Header (AH) and both authentication and encryption can be provided by the Encapsulating Security Payload (ESP). The AH is infamous since authentication can be provided solely by ESP. A difference is that AH can also provide integrity over the immutable fields of the IP header (fields that should not change across hops), whereas ESP can only provide this by also encapsulating the original IP header at the expense of higher overhead.

In terms of header structure AH and the ESP feature strong similarities. They contain a 32-bit `Security Parameters Index (SPI)` which is an arbitrary value used to identify the

SA (since the SA is used to agree upon and exchange of cryptographic means and material, this value ultimately maps to a specific key used for authenticating and/or decrypting the traffic). IPsec allows nodes to share SAs e.g., for multicast traffic.

The 32-bit `Sequence Number` is used for replay prevention. It is implemented as a monotonic counter which initialises to zero and is incremented by one for every packet sent. Packets with a sequence number lower than that of the latest received valid packet are dropped and for packets to be considered valid the counter needs to be within a sliding window. The sequence number does not cycle (wrap around back to zero); once the maximum value is reached no more packets will be accepted and a new SA needs to be established. Generation of the sequence numbers is required but verification is optional. An extensions to the 32-bit sequence number is offered to facilitate high-bandwidth applications. This 64-bit Extended Sequence Number (ESN) consists of a 32-bit low-order part (which is put on the wire) and a 32-bit high-order part (which only exists internally in the endpoints). Depending on the version of the specification the ESN needs to be explicitly negotiated or is implicitly enabled.[25, p. 8]

A variable-length `Integrity Check Value (ICV)`, aligned to 32-bit boundaries for IPv4 and 64-bit boundaries for IPv6, contains the value used for verifying the authenticity of the packet. IPsec defines the fields used for the calculation of the ICV and standardises several algorithms for this process. The algorithm used by a session is negotiated in the IKE phase and part of the SA.[25, p. 11] RFC 4307 specifies the following algorithms for authentication purposes to be used in IPsec: *AUTH_HMAC_MD5_96*, *AUTH_HMAC_SHA1_96* and *AUTH_AES_XCBC_96* (96 means truncated to 96 bits).[26, p. 4] RFC 4543 also allows for *AUTH_AES_GMAC* which allows for efficient implementation in hardware and allows for tens of gigabits of throughput and higher (as mentioned earlier in Section 3.3).[15] RFC 4868 specifies the following algorithms, which improve upon their predecessor SHA-1, for authentication purposes: *AUTH_HMAC_SHA2_256_128*, *AUTH_HMAC_SHA2_384_-192* and *AUTH_HMAC_SHA2_512_256* (128, 192 and 256 are the truncated lengths and are chosen according to the formula "nnn/2" in accordance with the birthday bound for each algorithm).[27, pp. 6,18] To ensure against possibly-uncovered weaknesses of the de-facto encryption standard AES RFC 7634 selects the *ChaCha20* stream cipher with the *Poly1305* authenticator to be used as a standby cipher (a cipher which can be used as a fallback).[28]

An 8-bit `Next Header` indicates which IP protocol follows the AH or ESP header (the original payload) and when necessary padding is used to make sure certain fields start or end at the correct boundary.

The main differences (relevant for the CoCo VPN architecture) between the AH and ESP are that ESP is also meant to provide confidentiality and contains its payload (most likely in encrypted form) within the ESP header, whereas AH's payload follows the AH header. For clarity the format of the AH is shown in Figure 3.1 including the length of possible (non-truncated) MAC's.

3.3.3 Suitability of IPsec for the CoCo VPN service

IPsec fulfils in every requirement of the CoCo VPN service. However, since at the moment only authentication is desired implementing ESP is deemed unnecessary. Instead AH could be applied between the VPN client and the PE. The protocol must account for network address translation (NAT) between the VPN client and the CoCo portal, although since the VPN service is envisioned to be used in national research and education networks

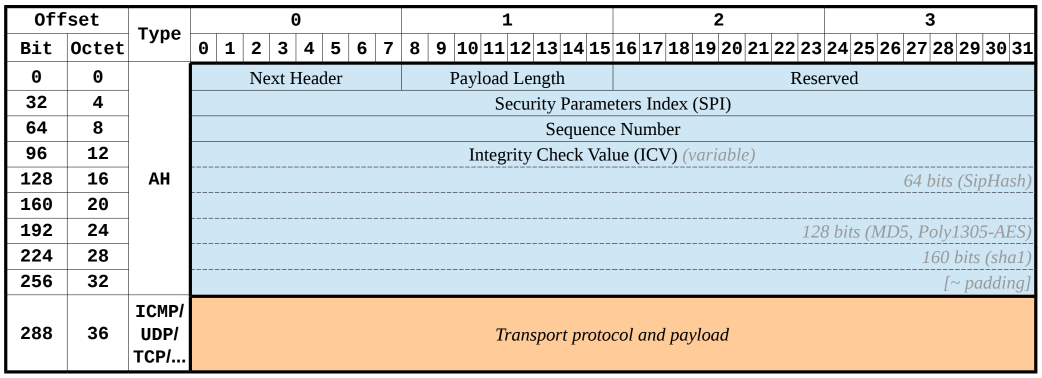

Figure 3.1: IPsec AH protocol header

(NRENs) — which are typically expeditious in the deployment of new network protocols like IPv6 — the extent of this concern first needs to be determined. The design also needs to account for mobile clients e.g., one envisioned use case is possibly roaming researchers supplied with notebooks. IPsec has facilities for NAT traversal and has been enhanced with IKEv2 Mobility and Multihoming Protocol (MOBIKE) which enables mobility of IKE SAs, however it requires using tunnel mode which has an increased overhead compared to transport mode.[29] Technically, it is possible to use part of the SPI to encode options to alter behaviour on the intermediary device that authenticates the traffic such that even when using the AH the source IP address could be considered a mutable field. Unfortunately, this would likely result in violations of the IPsec specification, a course of action generally best refrained from.

Using IPsec has the advantage that the VPN service can use the extensive amount of specification and tools available. Furthermore, since it is a network layer protocol no applications need to be adapted and it could eventually be used to also provide confidentiality. Since key exchange takes place in user space and the VPN service is deployed in an environment assumed to be (largely) overseen by the creator of the service, a form of IPsec-light could be created that minimises the downsides of IPsec. An approach that could be taken is to fork an open-source IPsec implementation to create a lightweight alternative at the cost of sacrificing compatibility with implementations of IPsec that do follow the (complete) specification. Unfortunately, this could cause problems for systems that are already using IPsec and it likely still means a lot of complexity could be inherited, risking technical debt.

When pursuing solely authentication of network traffic a more simple solution might be preferred. When confidentiality is also desired a comprehensive suite like IPsec could have merits. A consideration to be aware of is the requirement of the CoCo VPN service to provide authentication of VPN traffic on intermediary devices instead of on end-to-end basis. This could make putting IPsec to use for both authentication and encryption more advantageous (providing confidentiality while distributing the keys to many devices sounds contradictory). A solution could be to use different keys for authentication and confidentiality purposes; the authentication keys are shared between the domains and the two (or possibly more) endpoints, while the keys used for confidentiality are known only by the endpoints.

3.4 CoCo authentication protocol

In Section 3.3 it is determined that using a readily available authentication protocol has advantages over creating a custom protocol, however it also shows that it might cause too much inflexibility and complexity. In theory one could divert from the specification and only implement the necessary parts in a way that suits the use case, but this is generally seen as a bad practice.

In case no available protocol suits the use case appropriately then a new protocol can be created at the costs of designing, implementing and maintaining it. This is likely to be the best approach for the CoCo VPN service due to its relatively unique requirements (non-end-to-end authentication of traffic in a multi-domain setting). For this research a preliminary design of such a protocol is proposed.

3.4.1 CoCo protocol header

The preliminary CoCo header has been created with the following considerations in mind, the header should:

- be lightweight: since the VPN is meant to facilitate computational intensive research activities;
- be flexible: behaviour of the device processing the header should be adjustable, e.g., via flags;
- facilitate authentication: via a variable length field several hashing algorithms should be supported;
- ideally also support home and mobile users: it is envisioned that the some users of the VPN service are likely to roam and be on networks using NAT;
- and possibly be extendible: it is likely that end-to-end encryption cannot be facilitated by the applications in all occasions, thus the need for encryption provided by the VPN service might arise. The CoCo authentication header could then be extended to also facilitate this need. Extendibility should either provide in the flexibility to enable the required functionality without changing the header format or by ensuring future changes can be made to the format.[1]

Figure 3.2 shows the proposed CoCo authentication header. The header is heavily influenced by IPsec AH but provides additional flexibility. For instance it is implemented on top of User Datagram Protocol (UDP), which makes it easier to traverse middle boxes using NAT and firewalls. Initially it had flags to alter the processing of the header, which for example could be used to relax security checks to accommodate NAT environments. However, during the design it became apparent that supporting such corner cases gets tricky very quickly (especially when not all the required use cases and their security requirements are clear). Therefore, it has been decided to remove provisions like these and allow them to be added where necessary at a later stage.

The CoCo header follows a normal UDP header from which the source and destination ports are used to signal the presence of the CoCo header. In the proposal port number

[1]Since this project solely comprises authentication this design goal is not taken into consideration in the proposed protocol.

49344 from the set of dynamic and private ports is used, solely because its hexadecimal representation is `0xc0c0`; since the port numbers are not used by an actual application and the way the authentication is generated by the client is not yet defined, it has no other criterion than being a relatively uncommon and easy to spot combination. In case the authentication scheme is extended to also include end-to-end use cases then these port numbers can become more important since a daemon needs to listen on a certain socket. Technically more information could be encoded into this identifier (e.g., the CoCo authentication protocol version number) but while this might lower the overhead per packet it could also make maintenance of the specification and the implementations more cumbersome.

The first field of the actual CoCo authentication header is the 16-bit `CoCo Identifier` field. Since the UDP source port number discussed above could be subject to mutation by a middlebox that identifier might not be sufficient, hence the CoCo identifier field. Part of the identifier (e.g., the last four bits) could be used for encoding the version of the authentication header and might be used by the parser to determine how to parse the header in case changes are made to the protocol. This method saves bits at the cost of slightly more complexity. A dedicated version number field is likely to be unnecessary since the protocol is meant to be used in a semi-private environment, and will be implemented in hardware that can be reprogrammed. On the other hand it could be prudent to still include it because when a provider serves multiple customers, upgrading the protocol format might be easier this way (otherwise coordinated efforts could be required).

From this point the CoCo header follows the format of the IPsec AH header since no other facilities are required. Where necessary the SPI can be used to encode additional functionality, i.e. certain bits could be used to define distinct modes that change the way the network device verifies the authenticity of the packet. At the moment the following modes are envisioned: `NAT` mode could be used to signal the presence of a NAT device (which could mean that the source IP address and the source port number are not included in the calculation of the authentication code) and `Relaxed` mode could be used to signal that no fields of the IP header are included in the calculation of the authentication code. This will reduce the amount of sessions that can be defined (e.g., with two modes 31 bits are left for identification when the reserved IPsec values[30] are not taken into account).

Alternatively the `Header Length` field could be halved where one half is used for flags or modes and the other half encodes the length per four octets plus (since the minimal length of the header is four rows of 32 bit the maximum integrity check value that can be represented by a four bit Header Length field is 672 bits which is sufficient for today's hashing protocols).

In theory the length field is not necessary since the algorithm used could be also encoded into the SPI and thus the length of the ICV field can be deduced. However, this could make parsing the header more complex. Encoding information in other values is best to be refrained from when information can also be conveyed by transferring it in the form of metadata as part of the session. The downside to this approach is that it results in a more stateful approach.

The `Sequence Number` field follows the same rules as the IPsec AH. It is initialised at zero and stops at its highest value. After that a new session needs to be established. The sequence number is still a 32-bit value where IPsec has the option to negotiate a 64-bit extended sequence number to allow high performance applications to send more data without renegotiating an SA. Since a 32-bit value can be used to transfer about 6 TB using an maximum transmission unit (MTU) of 1500 bytes it likely suffices (transferring 6 TB over a 10 Gb link takes about 1.3 hours and while using jumbo frames the session can be prolonged

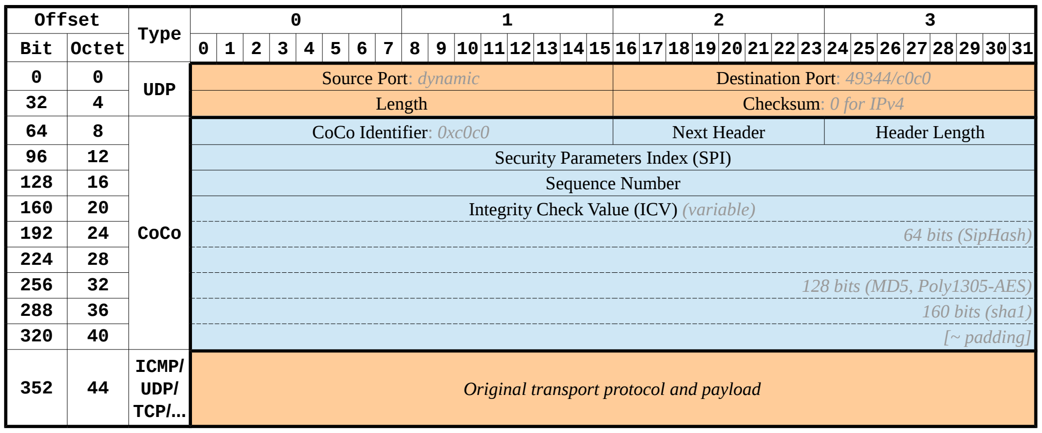

Figure 3.2: Custom CoCo protocol header

to about 8.5 hours in the ideal case). In the scenario where a 100 Gb link gets saturated with packets of 1000 bytes (a more realistic average that still is on the high end) a new session needs to be established every 5 minutes.

Even with session pre-establishment and a smooth roll-over this could become a problem. When such extreme use cases are envisioned (a category a DNA sequencer could be considered part of) it might be better to immediately move to 64-bit sequence numbers at the cost of a 32-bit overhead every packet. Also when multi-domain end-to-end authentication is considered 64-bit sequence numbers are advisable because it puts less stress on the part of the CoCo architecture that handles session establishment (a subsystem which will then comprise a more complex chain and thus has more points of failure).

Another option is to use a similar approach as with the ESN of IPsec. IPsec allows for a 64-bit sequence number of which only 32 bit are sent on the wire while the endpoints keep track of the 64-bit value. While this is a good approach implementing it in P4 might be more complicated.

When a sequence number is about to reach its maximum value the client could negotiate a new session before its old session has expired. To ensure a smooth roll-over during a short period of time both sessions will be considered valid.

3.4.2 CoCo authentication protocol behaviour

The flow chart in Figure 3.3 depicts the actions taken by the PE device. First, a packet hits the PE and enters the forwarding pipeline of the device. At a certain point in this pipeline e.g., after functions of higher priority like ACLs and error checking have been applied, the PE needs to determine whether the packet is destined for the CoCo VPN service. This check can consist of looking for a specific header or verifying whether the destination of the packet is a VPN endpoint (and maybe even its source too). If the packet is not destined for the VPN then the normal processing path can continue which ultimately will result in forwarding of the packet (assuming the switch has the necessary routing information).

Once the packet is determined part of the CoCo VPN service the session identifier is used to look-up the authentication key. The figure depicts this as a single action but could consist of separate steps. In case of the CoCo VPN service, it is expected that every session identifier maps to a single key and both are supplied to the switch simultaneously. Thus, if the session identifier of the packet does not match with a table the key is also not available and vice versa. In this case the packet must be dropped.

If the session material is known by the PE, the sequence number is checked for validity. For replay protection a per-session sliding window is used and only sequence numbers within that window are accepted. If the number attached to the packet is lower or higher than expected that packet is dropped. This step is best carried out before the MAC gets verified since it minimises the amount of work the PE has to do (in case sequence numbers are invalid) and also lowers the chance of introducing DoS opportunities.

In case the sequence number check passes, the MAC is computed using the session key and compared to the value carried by the packet. Computation of the MAC should follow a similar method as with the computation of checksums; the MAC field of the header is set to zero during the computation. Fields included in the computation should be at least the authentication header and its payload, but could also include parts of the network protocol to ensure the source and destination addresses are authentic. Depending on the use case the fields included could be different per session, which should be signalled out-of-band, via protocol flags or by encoding options in other values. Only if the MACs are equal the packet is considered authentic and is granted access to the VPN paths. Before forwarding the packet, the latest valid sequence number is saved and its authentication header removed. This step includes modifying the IP header to account for the change in protocol type, packet size and header checksum. If the MACs do not match the packet is dropped.

The actions can be enhanced in several ways. For instance, at several steps statistics could be gathered for monitoring and troubleshooting purposes. This could include packet count and bytes sent (per-session and in total, separately for packets that succeeded or failed authentication). These statistics could be transferred periodically to the controller in a push or pull fashion. Similarly, a metering function could be used that limits the amount of traffic the VPN service or an individual session is allowed to use. Such a function could also be used to lower the probability or the impact of a DoS attack, however it needs to be implemented such that no new attacks are introduced (e.g., by limiting the total amount of VPN traffic, malicious users could deny service to legitimate users).

In the flow chart packets are dropped silently instead of reporting the error to the sender. The rationale behind this is that it keeps the P4 implementation simple. Instead it moves the complexity to the client which should implement a function to reset a stale or out-of-sync session. In the case no return traffic is received after a certain amount of time (or packets) the client should establish a new session. The result is that only bidirectional flows are supported. It is expected that this will not cause a problem because bidirectional flows are anticipated. In practice a message could be used to indicate the authentication failed, however for security considerations it should not mention why in detail. Also, timing attacks possibly leading to key recovery should be taken into account when implementing a feedback method.

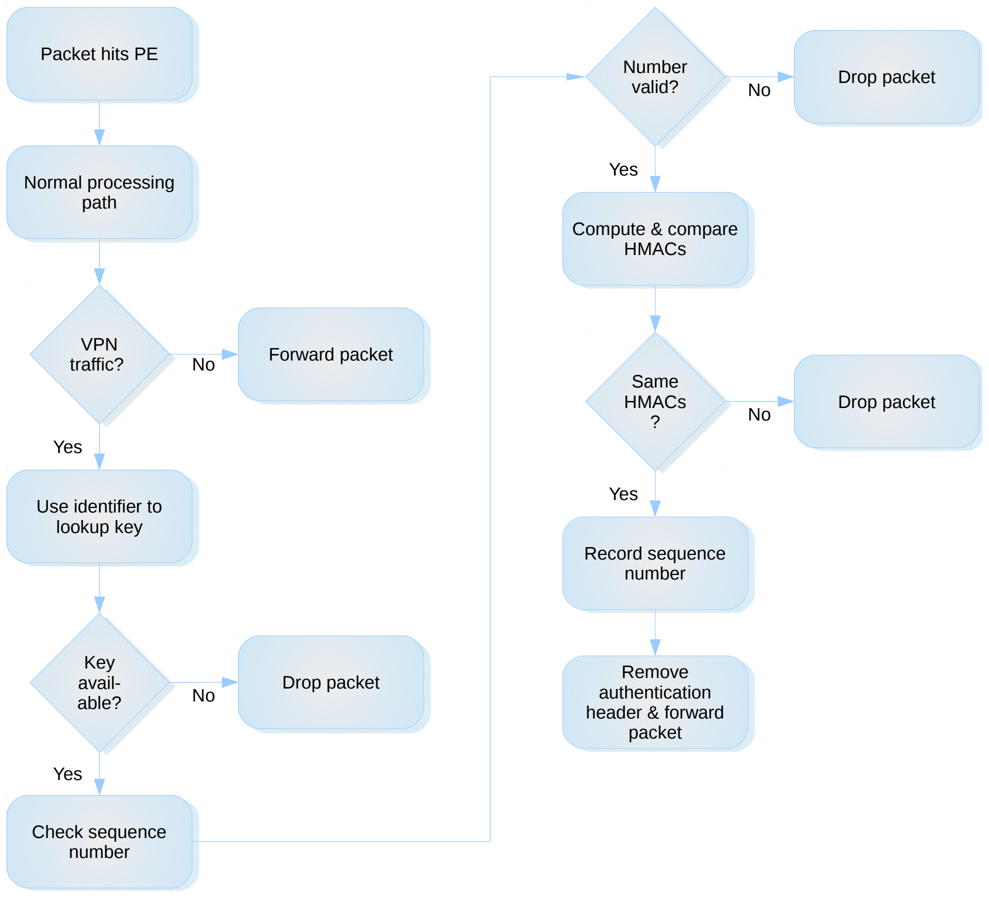

Figure 3.3: PE authentication actions flow chart

4 Implementation in P4

A proof of concept has been created to test whether the functionality required for authentication of network traffic can be implemented using P4. In this chapter this proof of concept is discussed. First a simplified authentication protocol is described, then the aspects of the P4 program constructed during the project are detailed step by step.

4.1 Simplified authentication protocol

The Generic Routing Encapsulation (GRE) has been used for the simplified authentication protocol. It provides in all the facilities required for the simplified authentication scheme: session identifiers can be stored in the `Key` field, sequence numbers used for replay protection can be stored in the `Sequence Number` field and the `Checksum` field can be used for simulating the MAC. Furthermore it intrinsically allows for encapsulation of other packets and is widely supported by hosts and network tools.

GRE was first specified in RFC 1701 and updated in RFC 2784. It can be used for encapsulating IP packets (or even complete Ethernet frames) into other IP packets so that they can be tunnelled across the Internet. Where the RFC 1701 specification provides in several flags (e.g., `Checksum Present` bit and `Key Present` bit) RFC 2784 removed most of them (only the checksum flag was kept) and a new GRE version number was introduced for PPTP.[31, 32] PPTP is specified in RFC 2637 and uses an enhanced GRE header (which still includes the flags because it predates RFC 2784).[33] In RFC 2890 the Key and Sequence Number Extensions are specified which basically adds these flag bits back and elaborates on their use (the bits are no longer referred to as flags but are used in this report interchangeably).[34] Where the original specification states that the Key field may be used for authentication purposes RFC 2890 states that it may be used for "identifying individual flows within the tunnel".

The history and intricacies of GRE are not important. What is important is that by setting flags (e.g., the `Checksum Present` and `Sequence Number Present` bits) GRE's optional fields can be enabled. Since both sides of the communication channel are controlled (the client used to send the GRE packets and the P4 software switch) there is no need to strictly adhere to the specification. This flexibility allows using the mentioned fields for authentication purposes.

In Figure 4.1 the GRE header is given and shows how it is used for authentication purposes. Since the specification reflects their use closely the `Key` field holds the 16-bit session identifier and the `Sequence Number` stores the 16-bit sequence number. Their semantics are kept as simple as possible which means the session identifiers are not used to encode additional options and the sequence numbers are not used to represent a larger number internally (i.e. as with IPsec's ESN). However, each session identifier does have its own sequence number to keep track of and a sliding window is used (set to an arbitrary size of 125 packets).

The `Checksum` field however is used to hold the MAC and its semantics do not adhere to the checksum as specified by the GRE specification. A normal GRE checksum contains the one's complement of the one's complement sum of all the 16 bit words in the GRE header

Offset		Type	0								1								2								3							
Bit	Octet		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	GRE	Flags: *CKS*				Reserved0									Version			Protocol Type: *0000 (possibly GRE keepalive)*															
32	4		Checksum: *MAC (CRC16)*																Reserved1/Offset: *key (not on wire)*															
64	8		Key: *session identifier*																															
96	12		Sequence Number																															
128	16	ICMP	*ICMP echo request with random payload*																															

Figure 4.1: CoCo protocol header

and its payload.[32, p. 3] To simulate a MAC the checksum's value is altered by storing the MAC key in the `Offset` field. Prior to sending the packet the key in the Offset field is removed. On the wire the checksum will be wrong according to the GRE specification but both endpoints know that they need to add a key to the Offset field to obtain the right result. What key is used depends on the session identifier.

The idea behind the authentication scheme is that an authentication header is inserted between the network and transport layers. At the very least the transport layer and its payload must be protected by the MAC. Since the authentication header is removed after the P4 switch successfully verifies the packet to be authentic, and the scheme does not require packets to be tunnelled, the GRE payload has no need for an additional IP header. Because it is practical to test with Internet Control Message Protocol (ICMP) traffic (i.e. no software is needed on the remote host) pings are used as the payload.

4.2 P4 program

In the following sections different aspects of the developed P4 program are given. First the way the program was developed is discussed, then a high-level overview of the program is shown and finally the actual P4 code is given.

4.2.1 P4 development environment

Since P4 hardware support is in its early stages a P4 software switch is used as provided by the P4 Language Consortium.[1] There are two different software switches. It was not clear which software switch suited the authentication proof of concept best, so it was planned to use the first software switch. In case problems would arise the second switch would be tried.

The first software switch (called the behavioral model) is written in C. For every software target, the source code gets pulled into the target's working directory and is together with the P4 program compiled into an executable. This means that it is easy to make changes to the software switch's code but also requires a full compilation every time the P4 program is changed.

To be more flexible a second software switch (called the behavioral model 2) is being developed in C++. It does not require recompilation of the software target when a change is made to the P4 program.[35] In practice it appears that this does result in less recompilations but is a bit harder to get started with and to make adaptations to the software switch; some source code of the software switch can be/needs to be local (i.e. reside in the working

[1]The starting point for working with P4 code is https://github.com/p4lang

directory) while other source code stays at its default location. Also, it was a bit unclear how to use the behavioral model 2 in the P4factory (an all-in-one P4 development and experimentation environment)[2] so the behavioral model 2 including its dependencies were installed manually.[3] Work on the behavioral model 1 seems to have stalled while the behavioral model 2 is in active development.

The software switches implement version 1.0 of the P4 language. However, it is possible to enable version 1.1 of the language using the behavioral model 2.[4] Version 1.1 of the P4 language is currently in draft form so during the project only version 1.0 was used.

4.2.2 Program overview

The flow chart in Figure 4.2 depicts the actions taken by the P4 software switch in a high-level manner. The figure is very similar to Figure 3.3 but the actions reflect the P4 packet processing pipeline more closely. For instance, it shows that the authentication functionality is located at the egress pipeline. The figure is included to provide clarity before the details and the actual code are presented in the following section.

4.2.3 P4 code

In this section snippets of the P4 code are given (for brevity, clarity and aesthetic reasons the sequence the snippets are shown in may be different from the actual P4 program and comments may have been added or removed; however, the code itself is left unchanged). The full P4 program and auxiliary code are made available via a git repository[5] and as an archive attached to the digital version of this report.

At first the program was made using the behavioral model 1, but due to issues stated in a later section, development switched to the behavioral model 2. For this reason all the code shown is meant to be run on the behavioral model 2 unless stated otherwise (although syntactically the code should be the same).

The `simple_router` target[6] is used as the basis of the P4 authentication target. Technically only layer 2 functionality is required, however using a target with routing functionality allows for integration with Mininet[7] which makes testing the P4 program easier.

[2]https://github.com/p4lang/p4factory

[3]https://github.com/p4lang/behavioral-model

[4]p4c-bm can be used to generate JSON representations of the P4 program that are written with P4 version 1.1 in mind: https://github.com/p4lang/p4c-bm

[5]https://github.com/JcKlomp/rp2-p4-authentication

[6]https://github.com/p4lang/p4factory/tree/master/targets/simple_router

[7]Via Mininet instant virtual networks can be created on a Linux system: http://mininet.org/

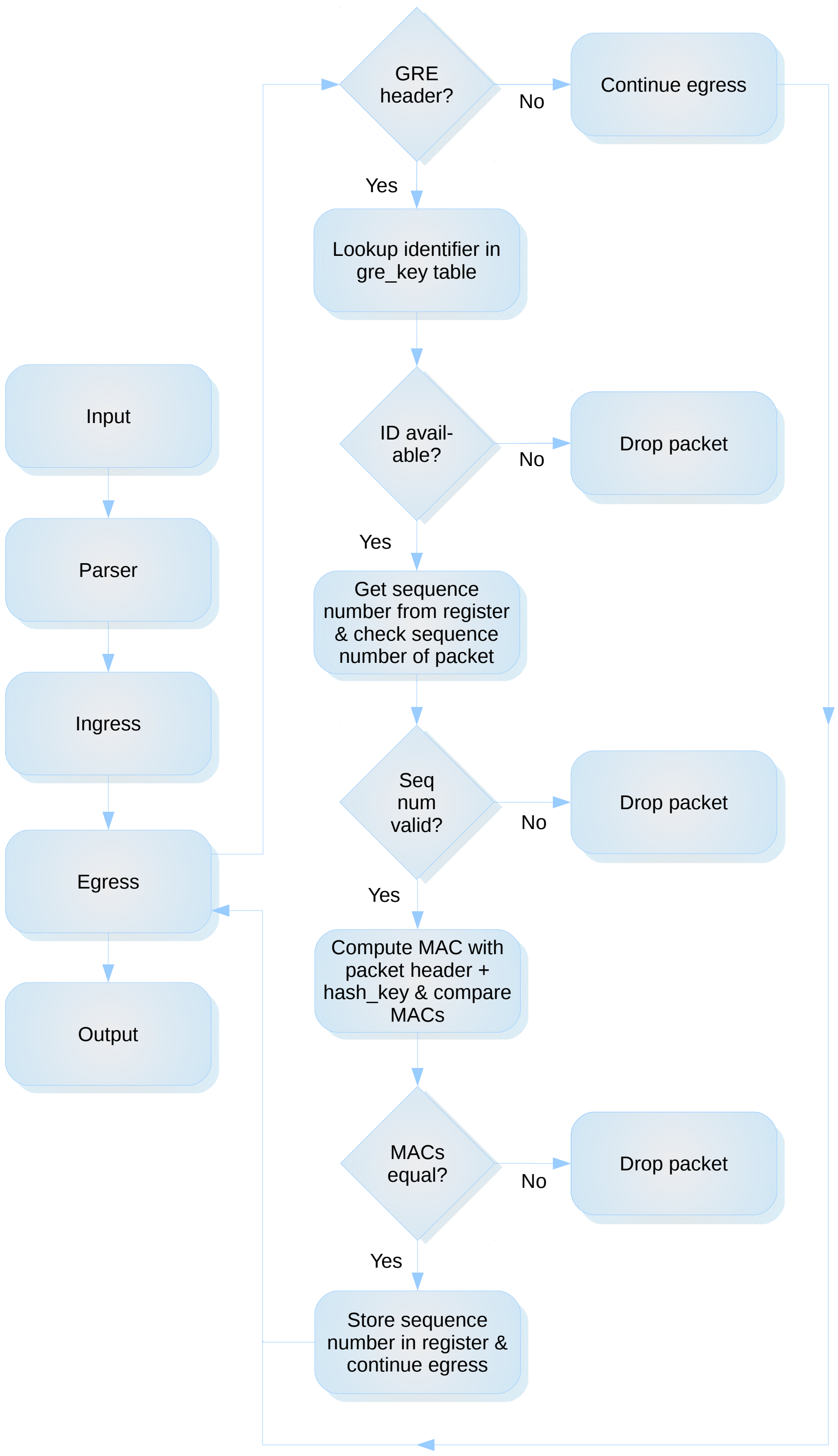
GRE header?
No
Continue egress
Yes
Lookup identifier in gre_key table
Input
ID avail-able?
No
Drop packet
Parser
Yes
Get sequence number from register & check sequence number of packet
Ingress
Seq num valid?
No
Drop packet
Egress
Yes
Compute MAC with packet header + hash_key & compare MACs
Output
MACs equal?
No
Drop packet
Yes
Store sequence number in register & continue egress

Figure 4.2: P4 program actions flow chart

GRE header

First the GRE header type called `gre_t` is declared (see Listing 4.1). Via the `fields` attribute the fields of the header are named and followed by their respective width in bits.[8] The GRE flags are declared as a compound field instead of individually bit by bit. This is because using the behavioral model 2 these fields are reset to zero after the egress phase is ends.[9] Declaring these fields as one compound field works around that problem. Although the switch should never forward a packet containing a GRE header (since that header must be stripped off) this action was taken during testing the checksum functionality and now only exists for historical reasons.

The declared GRE header does not comprise the full functionality of the GRE protocol. For instance the P4 implementation does not contain any optional fields. In P4 optional fields can be implemented using a variable-length field.[10] Only one variable-length field can be declared per header and variable-length fields have many other constraints. For instance, variable-length fields cannot be used for metadata (headers that are declared similarly as normal headers but are not deparsed after egress thus never sent) and cannot be used for checksum calculations. To actually use a variable-length field it needs to be parsed further until it has been split into fixed-sized data types.[11] It is likely that the actual authentication header used for the CoCo VPN service will allow MACs of differing length, therefore using the variable-length fields needs to be investigated further.

Listing 4.1: GRE header

```
header_type gre_t {
  fields {
    flags : 16;  // flags as compound field
    protocolType : 16;
    checksum : 16;  // MAC
    offset: 16;
    key : 32;  // session ID
    sequenceNumber : 32;  // replay protection
  }
}

header gre_t gre;
```

Similarly to the GRE header, a GRE metadata header type is declared called `gre_-metadata_t`. It is used to hold auxiliary data used during the verification of the authentication of the packet. Most important are the `index`, `hashKey`, `prevSequenceNumber` and the `computedHash` fields.

The `validKey` field is used in the control block for conditionally applying the authentication-related tables. Technically it is possible to use another field that would also

[8]This follows the P4 language 1.0 specification. Version 1.1 deviates from this notation and allows for more control over the data types used for the fields.

[9]Using the debugger (p4dbg.py) it was verified that these fields contain the right value up until the egress phase. After that the packet is deparsed and sent. For an unknown reason some of the fields that compose the compound field were set to zero on the wire. This problem was absent using the behavioral model 1.

[10]In P4 1.0 this is done by setting the field to width `*` and in P4 1.1 the `varbit` type is used. A header that contains a variable-length field can be assigned a maximum length to limit its length.

[11]An example of parsing variable-length fields is given in the `TLV_target`: https://github.com/p4lang/tutorials/tree/master/examples/TLV__parsing

be set in case the session identifier is known, but this way the semantics of the fields are more clear. The validKey, index and hashKey fields are set by the GRE match+action table (described later). The prevSequenceNumber and computedHash values are respectively retrieved from a register and computed. Both are used for comparison with respectively the GRE checksum and sequenceNumber fields and used to determine whether the packet is valid. The emptyChecksum is never assigned an actual value and used solely as a work around for setting the checksum field to zero during the computation of the computedHash field. The reason for this work around is that using the #define directive directly to define a field sometimes results in a wrong value (i.e. using the behavioral model 2, defining the value '0x0000' resulted in the value '0x0' of bit width '0', after compiling the program to JSON representation).

Listing 4.2: GRE metadata header

```
header_type gre_metadata_t {
  fields {
    validKey : 1;  // set when the session ID is known by the switch
    index : 16;  // index of register that holds latest valid sequence number
    hashKey : 16;  // holds the MAC key
    prevSequenceNumber : 32;  // holds the latest valid sequence number
    computedHash : 16;  // holds the locally computed MAC (simulated via checksum)
    emptyChecksum : 16;  // never assigned; always 0x0000
  }
}

metadata gre_metadata_t gre_metadata;
```

GRE parsing

After the headers required for GRE are declared they need to be parsed. This is done by adapting the IP parser. After the GRE header has been parsed control of the program continues in the ingress pipeline (see Listing 4.3). The Ingress pipeline of the constructed P4 program is not different from that of the simple_router target. It was attempted to check for a valid GRE header at this point, but this led to the switch crashing. In theory it should be able to verify the GRE header at this point and either forward or drop these packets (although at the ingress pipeline packets cannot truly be dropped; only the egress_spec can be set to the drop port which can be overridden by subsequent tables in ingress).[10, p. 58] Therefore, it was decided to move all the GRE control flow to the egress pipeline.

Listing 4.3: GRE parsing

```
#define IP_PROT_GRE 0x2f

parser parse_ipv4 {
  extract(ipv4);
  return select(ipv4.protocol) {
    IP_PROT_GRE : parse_gre;
    default : ingress;
  }
}

parser parse_gre {
  extract(gre);
  return ingress;
}
```

GRE control flow

The P4 program is of quite modest size, but if there is a most comprehensive part then it will be the egress pipeline (see Listing 4.4). First it checks whether the packet traversing the pipeline contains a valid GRE header via the `valid(gre)` conditional. If this is the case then the `gre_key` table is applied. The descriptions of the tables and actions will follow in later sections.

When the session identifier hold by the GRE `Key` field results in a match in the `gre_key` table an action is taken that sets the `validKey` metadata field to 1. Via the conditional `gre_metadata.validKey == 1` checking the authenticity of the packet either continues or the packet is dropped by applying the `gre_drop3` table. It can be seen that there are three different tables used for dropping packets. The reason for this is that it is not possible to reference multiple times to the same table and tables are necessary for applying actions. The P4 specification hints that this is a per-target limitation, in this case likely imposed to prevent recursion.[10, p. 74] A possible advantage of using multiple tables e.g., for dropping traffic, is that separate counters can be used to keep track of the amount of dropped packets for specific reasons (although it is not inconceivable this could be provided too by a single table using different entries). Another advantage is that (at least in the software switch) the name of the applied tables and actions taken are included in the logs which helps development.

The next control block is preceded with the `((gre.sequenceNumber > gre_-metadata.prevSequenceNumber) and (gre.sequenceNumber < gre_metadata.prevSeq-uenceNumber + 125))` if-else statement and is used to check whether the sequence number the packet carries is expected. The number needs to be higher than the latest stored number, but not higher than the window of 125 packets (which includes the current/latest valid packet so the window actually fits 124 packets). This way the authentication scheme tolerates a small amount of packet loss while keeping the opportunity for a brute force attack low (which could lead to unwanted access of resources and possibly result in a DoS).

When the sequence number is considered valid the checksum (the simulated MAC) can finally be computed which is done via the `gre_compute_hash` table. Then the checksum of the GRE header and the computed checksum are compared for equality. If they are equal the packet is considered authentic. Subsequently, the (higher) sequence number is stored in a register via the gre_update table and the GRE header is removed via the gre_remove table. At this point the GRE part of the egress pipeline is complete and the send_frame table from the simple_router target is applied so that the packet can be forwarded properly.

Listing 4.4: GRE control flow

```
control egress {
  if (valid(gre)) {
    apply(gre_key);
    if (gre_metadata.validKey == 1) {
      if ((gre.sequenceNumber > gre_metadata.prevSequenceNumber) and
  ↪  (gre.sequenceNumber < gre_metadata.prevSequenceNumber + 125)) { // sliding window
        apply(gre_compute_hash);
        if (gre.checksum == gre_metadata.computedHash) {
          apply(gre_update); // store new sequence number
          apply(gre_remove); // remove gre header
        }
        else {
          apply(gre_drop); // not a valid hash
        }
      }
      else {
        apply(gre_drop2); // not a valid sequence number
      }
    }
    else {
      apply(gre_drop3); // not a valid key (identifier)
    }
  }
  apply(send_frame); // normal forwarding functionality
}
```

Tables

As shown in the previous section the P4 program uses several tables. Via the tables entries can match packet fields which results in the execution of the specified action, possibly with specific parameters. In terms of the GRE authentication scheme only the gre_key table is used for matching entries. The other tables are used to execute default actions.

Listing 4.5 shows the gre_key table. It reads the GRE Key field of the current packet and looks for an exact table entry match.[12] If the table has a hit (an entry matches) then the action supplied with that entry is taken, which for the gre_key table means that the set_gre_key_valid is executed with the index and the hash_key supplied as parameters (as stated earlier, the index is used to look up the right sequence number in a register and the key is used as input to the simulated MAC). If no match is found then the default set_gre_key_invalid action is taken. This action explicitly sets the validKey metadata field to 0. This is done so that the control flow is correctly directed to the gre_drop3 table, otherwise if the packet is dropped in a normal manner, the packet will erroneously traverse the control blocks after the gre_metadata.validKey == 1 check, even though the validKey field is not set to 1.

[12]How entries are matched is determined by the field_match_type option which can be exact, ternary, lpm, index, range or valid.

Listing 4.6 shows how entries are added to the `gre_key` table and how default table actions are set at runtime.

Listing 4.5: gre_key table

```
table gre_key {
  reads {
    gre.key : exact;
  }
  actions {
    set_gre_key_valid;
    set_gre_key_invalid;
  }
}
```

Listing 4.6: Adding table entries and actions

```
table_set_default gre_key set_gre_key_invalid
table_add gre_key set_gre_key_valid 123456789 => 0 0xabcd
```

As stated earlier the other tables are not used for matching entries but purely exist for executing actions (Listing 4.7). A peculiarity is the `force_drop` action. This is a work around for dropping packets containing a GRE header and works by truncating the packets to zero length. Why the behavioral model 2 was able to drop non-GRE packets normally but not GRE packets is not clear.

Listing 4.7: Other GRE tables

```
table gre_compute_hash {
  actions {
    compute_gre_hash;
  }
}

table gre_update {
  actions {
    update_gre_sequence_number;
  }
}

table gre_remove {
  actions {
    remove_gre;
  }
}

// all the gre_drop tables are constructed the same
table gre_drop {
  actions {
    //_drop; // doesn't work properly
    force_drop;
  }
}
```

Register actions

In P4 actions can be subdivided into primitive and compound actions. Compound actions are declared as functions which contain one or multiple primitive actions that will be executed sequentially. Primitive actions are the actual actions applied to the packets. “P4 supports an extensible set of primitive actions”, but “not all targets will support all actions. Target switches may have limits on when variables are bound and what combinations of parameter types are allowed.”[10, p. 52] From this point, actions can refer both to compound and primitive actions. Unless stated otherwise, in the context of a table action a compound action is implied (like `set_gre_key_valid`) which might be referred to as a function for clarity, and from the context of a compound action a primitive action is implied (like `modify_field`).

The actions will be described following the happy path (it is assumed no errors occur) and where appropriate the deviations from it are given. First, when the packet matches the `gre_key` table the `set_gre_key_valid` action is taken (Listing 4.8). This function accepts the `idx` and `key` parameters. First the `modify_field` action is executed to set the `validKey` field to 1, indicating that the packet has a valid session identifier. Then the `key` variable passed as a parameter to the function is stored in the `hashKey` metadata field. The reason for this action is to allow the key to be used in the calculation of the checksum. Also the `idx` variable is stored in the `index` metadata field. This is handy in case the register gets updated in a subsequent action. After these actions are carried out another function is called, `get_gre_sequence_number`, which is used to retrieve the latest valid sequence number from a register. Technically this function is unnecessary since the primitive action could have been used directly (this call is done purely for historical reasons and keeps the naming consistent).

In case no entry matches the `set_gre_key_invalid` function is executed which simply sets the `validKey` field to zero to prevent the packet from traversing wrong tables. Once this issue is properly debugged the default table action can be set back to the `force_drop` (or even the regular `drop` action once that issue is resolved).

Listing 4.8: gre_key table actions

```
action set_gre_key_valid(idx, key) {
  modify_field(gre_metadata.validKey, 1);
  modify_field(gre_metadata.hashKey, key);
  modify_field(gre_metadata.index, idx);
  get_gre_sequence_number();
}

action set_gre_key_invalid() {
  modify_field(gre_metadata.validKey, 0);
}
```

In P4 counters, meters and registers can be used to maintain state across multiple packets. Counters can be used for tracking the amount of packets and/or bytes matched in the table. They can be automatically applied to individual table entries or incremented by one by manually issuing the *count action*. Meters can measure data rate and expose it via a three-color marking algorithm. Registers provide a more general-purpose stateful memory and allow values to be read and written by actions.[10, p. 49] Since a sliding window needs to be implemented registers appear to fit the purpose best.

In P4 stateful memories are organised into named arrays of *cells* and a cell e.g., an individual register, is referenced by its array name and index.[10, p. 46] Stateful memories can either be used in *direct access* or *indirect access*. Direct access means that the register is bound to one table and every table entry has a dedicated cell. Entries cannot reference other cells nor can the register be referenced from other tables. Indirect access means that any entry can reference any cell in a global manner.[10, p. 46] A bit further in the specification another distinction seems to indicate that stateful memories can be optionally declared as *direct* or *static*. The direct attribute appears to conform to the direct access type and if it is not used then the stateful memory needs to be referenced via name and index, which appears to be conform to the indirect access type. However, static means that the resource is dedicated to a single table while being referenced by name and index, which appears to be a constraint of the indirect access type but this is not clearly stated in the specification.

On itself this is not a problem but these concepts are described only for the counter and meter memories, not for registers. When declaring the register it was assumed that the behaviour of the register in this regard would be the same or similar as that of the other memory types. Unfortunately it appears that the compiler insists the register be declared static, which again on itself is not a problem, however it is referenced from actions of two tables (reading and writing the register) yet no failure arises. There also was a discrepancy between the behaviour of the behavioral model 1 and 2 switches; where software switch 1 complained about the register having a wrong table declaration software switch 2 did not.

Another peculiarity is how to actually work with the registers. The specification mentions that "Although registers cannot be used directly in matching, they may be used as the source of a `modify_field` action allowing the current value of the register to be copied to a packet's metadata and be available for matching in subsequent tables."[10, p. 50] Afterwards this sounds straightforward and like the proper course of action, but instead the `register_read` and `register_write` primitive actions, available in the simple_switch target, were used during development.[13] The reason for this is that the runtime_CLI[14] uses the same functions for manipulating registers (and simply oversight). Currently, these actions are not part of the P4 specification, but it is expected they will be added in a future revision. At this point the behavioral model 1 was not used any more because its runtime_CLI does not have the register functions, which made testing the functionality hard. During the project no indication was found that this software target allows manipulation of these stateful memories.

Reading the sequence number from the register involves the `get_gre_sequence_number` function which executes the `register_read` action. In this case the action takes the `prevSequenceNumber` field as the destination, the register `sequence_number_reg` is read-out at the index specified in the `index` field (Listing 4.9). The `update_gre_sequence_number` is given at this point because it is very similar to the way the register is read.

The rationale behind the instance count of the `sequence_number_reg` declaration is that this way it could be possible to use the session identifier as the index. However, adding such a number as a table action parameter proved impossible. Possibly the more precise data type declaration method of version 1.1 of the P4 language will enable this optimisation (less state would need to be stored by the controller and the switch).

[13]https://github.com/p4lang/behavioral-model/blob/master/targets/simple_switch/primitives.cpp

[14]A simple command line tool that can be used to program the tables at runtime and show information about the functioning of the switch.

Listing 4.9: GRE register actions and declaration

```
action get_gre_sequence_number() {
  register_read(gre_metadata.prevSequenceNumber, sequence_number_reg,
  ↪   gre_metadata.index);
}

action update_gre_sequence_number() {
  register_write(sequence_number_reg, gre_metadata.index, gre.sequenceNumber);
}

register sequence_number_reg {
  width: 32;
  static: gre_update;
  instance_count: 65536;  // cannot add 32 bit number via table action parameter, so
  ↪   session ID cannot be used as index
}
```

Checksum action

In Listing 4.10 the code used for MAC simulation is shown. First the `gre_checksum_list` field list is defined. A field list is used to specify a sequence of header fields to be handed to checksum and hash-value generators. The output of these calculation objects can be used by the P4 program in the form of an integer. The underlying algorithm of the function cannot be expressed in the P4 language, but instead needs to be provided by target.[10, p. 33] This means that algorithms like cyclic redundancy checks (CRCs) and hash functions are outside the scope of the P4 language and the algorithms supported may vary between targets. The language does allow interfaces to the functions such that they can be configurable at runtime. Examples given in the P4 Language Specification are a configurable seed value and specifying parameters like coefficients.

Currently for the simulated MAC, supplying the key to the function is done by putting the key in an unused field of the GRE header: instead of the empty `offset` field the `hashKey` metadata field is used for the calculation. However, a proper key supply facility is advisable because this would allow the underlying algorithm to clearly distinguish the message from the key, such that they can be mixed together by the algorithm itself.[12, p. 3] Whether the interfaces specified by the P4 language will allow such a use case (e.g., inputting a 128-bit key) has not become clear during this research.

The `gre_checksum_list` field list is referenced by the `gre_checksum` field list calculation object. It defines the algorithm used and the properties of the resulting value. In this case the same checksum algorithm used by the IPv4 header is utilised. Next the field list calculation can be used to verify and update the checksum of a header via the `gre.checksum` calculated field declarations. The *verify* option happens during parsing and can result in a parse exception, which can be handled by the P4 program. Parse exceptions are currently not implemented in the software switches (the behavioral model 2 issues a warning at compilation time) which means this option has no effect. The verify option, once implemented, would not provide the required functionality because to be able to dynamically change the way the checksum is computed per session the session key needs to be put into a metadata field first, an action that takes place after parsing the packets. The *update* option can be used to update the checksum field of a header prior to deparsing it (preparing the packet to be sent) at the end of the egress pipeline. This function is currently implemented, however it does not provide the functionality necessary for the simplified authentication scheme; packets need to be dropped if the computed checksum does not match.

Instead the `modify_field_with_hash_based_offset` primitive action is used.[15] It is issued from the `compute_gre_hash` function via the `gre_compute_hash` table. This primitive action can be used to apply a field list calculation (`gre_checksum`) and use the result to generate an offset value which can be used by the program. The P4 program uses this function to compute and store the simulated MAC in the `computedHash` metadata field so that it can be used to compare the checksums (as shown in Listing 4.4). The description of the primitive function is a bit unclear, but the understanding is that the base (0) and the size (65536) are used to determine the minimum and maximum value of the result (initially the size was left zero but this led to compiler errors).[10, pp. 53,57]

One might have noticed that this MAC simulation does not secure the packet against spoofed IP addresses. Technically it would be trivial to include fields of the IP header in the calculation, but it was decided not to since it made calculating the checksum at the client-side (the node that sends the authenticated packets to the switch) more complicated (this process is described in Chapter 5).

Listing 4.10: GRE checksum

```
field_list gre_checksum_list {
  gre.flags;
  gre.protocolType;
  gre_metadata.emptyChecksum;
  gre_metadata.hashKey; // dynamic hash_key via offset field
  gre.key; // session identifier
  gre.sequenceNumber;
  payload;
}

field_list_calculation gre_checksum {
  input {
    gre_checksum_list;
  }
  algorithm : csum16;
  output_width : 16;
}

// normal checksum facility not used
/*
calculated_field gre.checksum  {
  verify gre_checksum if (valid(gre));
  update gre_checksum if (valid(gre));
}
*/

action compute_gre_hash() {
  modify_field_with_hash_based_offset(gre_metadata.computedHash, 0, gre_checksum,
   ↪  65536);
}
```

[15]P4 also provides the `generate_digest` primitive action, which as its name indicates allows one to generate digests of packets, however it appears to be meant to send this digest to a controller for further processing.[10, p. 60]

Removal of GRE header action

After the packets have been verified to be authentic the GRE authentication header can be removed. This is done by the `remove_gre` function via the `gre_remove` table (Listing 4.11). This function uses the `remove_header` action to mark the GRE header as invalid which will prevent deparsing the header instance at egress. The result is that the GRE header is popped and valid parts higher-up the stack are copied to lower positions.[10, p. 55] Since the packet is altered the IPv4 header needs to reflect those changes. First the protocol is set to ICMP (during the project only ICMP was used as a payload) and the size of the GRE header needs to be subtracted from the total packet length.

Besides being easier to debug, there is no good reason to make updating the sequence number and removing the GRE header two separate steps. For an actual implementation this optimisation can be considered. Also, for the actual authentication scheme this code needs to be more sophisticated. For instance, different protocols need to be supported and the header is likely to support variable lengths. It is expected that these requirements will not result in (overly) complex code but might necessitate changes like removing the GRE header after the alterations to the IP header have been made (because referencing invalid fields results in undefined behaviour).[10, p. 32]

Because the IPv4 header has been changed, the header checksum has become invalid. Prior to sending the packet the switch needs to update the IPv4 header checksum. The code is similar to that shown in Listing 4.10 and is part of the base `simple_router` target, so it is not shown here.

Listing 4.11: remove_gre action

```
#define IP_PROT_ICMP 0x01

action remove_gre() {
  remove_header(gre);
  modify_field(ipv4.protocol, IP_PROT_ICMP); // static proto following gre header
  add_to_field(ipv4.totalLen, -16); // reduce length (size of gre header)
}
```

5 Proof of concept

In this chapter the functionality of the constructed P4 program is shown. First the way the program is tested is given, then the simplified authentication scheme implemented in P4 is demonstrated.

5.1 Test setup

The P4 program is run via a P4 software switch (the behavioral model 2) in a Mininet environment. Via Mininet a virtual network that comprises two hosts is created and connected to the P4 switch. The switch is actually functioning as a router that allows communication between the two /24 subnets. Then a Python script using Scapy[1] modules is used to send a packet containing the simplified GRE authentication header. Scapy is positioned on the link that connects H1 with the switch. Packets are sent to H2 via the switch. Assuming the packets are authenticated correctly, H2 should reply and the response captured by Scapy.

First an IP packet is created with the source and destination address set to H1 and H2 respectively. A GRE header is added that contains the session identifier and sequence number. An ICMP layer follows the GRE header and the packet is concluded with three random hexadecimals that comprise the ICMP `Data` field (used for identifying the packet and testing forgery). Scapy calculates the checksum including the key by putting the MAC key in the GRE `Offset` field. After the checksum has been calculated the key is removed from the `Offset` field and the packet is sent.

If the packet is considered authentic by the P4 switch, it removes the GRE header and then sends it to H2 which will respond with a regular ICMP echo reply packet. Since the P4 switch is actually a router that forwards normal IP packets it will forward the packet to H1 at which point Scapy will also receive it.

The packets are put on the wire via Scapy's layer 2 `sendp` function. Via a sniffer the payloads of the packets sent and received are compared and if they are equal the response is considered valid. The reason for sending the packet at layer 2 is that the Linux namespace facing the user (the root namespace) does not contain the IP addresses nor routing information necessary for reaching the hosts (which live in the Mininet host namespaces). Also the packets sent and received are asymmetric (GRE packets are sent and normal ICMP packets are received). Therefore Scapy cannot match the response correctly. The solution to use a sniffer in a separate thread proved the most straightforward.

Figure 5.1 depicts the test scenario (routing information is left out for brevity). The notation of the messages sent approaches Scapy's method for creating packets (for clarity a simplified notation is used in the figure).

[1]Scapy is an interactive packet manipulation tool which among others, allows for easily crafting and forging of packets: http://www.secdev.org/projects/scapy/

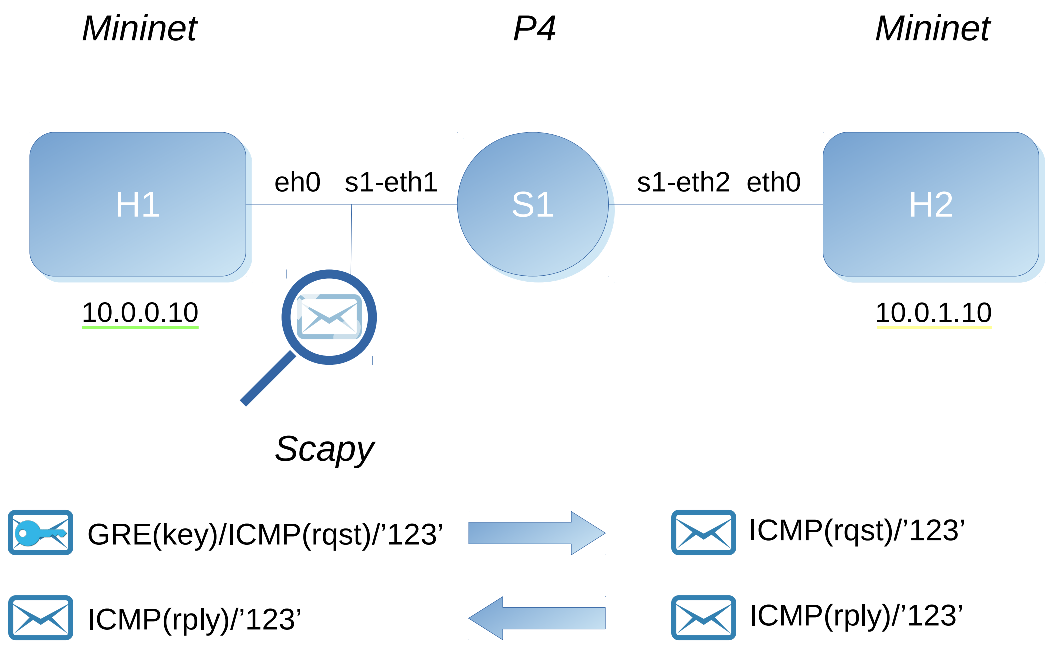

Figure 5.1: PoC test scenario

5.2 Demonstration

In this section the simplified authentication scheme implemented in the P4 program is demonstrated.[2]

5.2.1 Reachability test

First a regular ping is tried to verify network reachability between H1 and H2. Listing 5.1 shows the command in Mininet returned successfully and the sniffer on the left-hand side of the switch confirms this.

[2]A screencast of the demonstration has been made. It is published at https://raw.githubusercontent.com/JcKlomp/rp2-p4-authentication/master/demonstration/demo.webm

Listing 5.1: Reachability test

```
# Mininet
mininet> h1 ping -c1 h2
PING 10.0.1.10 (10.0.1.10) 56(84) bytes of data.
64 bytes from 10.0.1.10: icmp_seq=1 ttl=63 time=2.75 ms

--- 10.0.1.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.752/2.752/2.752/0.000 ms

# Sniffer H1 - S1-ETH1
# Src         => Dst        Prot Type Data     ID          Seq Chksum
1 10.0.0.10 => 10.0.1.10 ICMP RQST 1234567
2 10.0.1.10 => 10.0.0.10 ICMP RPLY 1234567
```

5.2.2 Session identifier 1

Via the `runtime_CLI` P4 command line tool the state of the P4 switch is checked (Listing 5.2). There are two entries in the `gre_key` table: 075bcd15 (123456789) and 3ade68b1 (987654321). These are the session identifiers, both are assigned separate keys (abcd and 1234 respectively) and separate indices (used for the registers). Via the `register_read` function both registers are read-out and shown to be initialised to zero.

Listing 5.2: P4 runtime state

```
# gre_key table entries
RuntimeCmd: table_dump gre_key
==========
TABLE ENTRIES
**********
Dumping entry 0x0
Match key:
* gre.key               : EXACT     075bcd15
Action entry: set_gre_key_valid - 00, abcd
**********
Dumping entry 0x1
Match key:
* gre.key               : EXACT     3ade68b1
Action entry: set_gre_key_valid - 01, 1234
==========
Dumping default entry
Action entry: set_gre_key_invalid -

# sequence_number_reg register values
RuntimeCmd: register_read sequence_number_reg 0
sequence_number_reg[0]=  0

RuntimeCmd: register_read sequence_number_reg 1
sequence_number_reg[1]=  0
```

In Listing 5.3 three packets are sent. First the `packet_send` function is used to create a packet with `123456789` as the session identifier, `0xabcd` as the MAC key and `123` as the sequence number. These parameters result in a valid packet, the switch authenticates and forwards it, and H2 responds with an ICMP echo reply containing the same random payload (frame 4). Next the same parameters are used to send another packet, however this time the remote host fails to reply (frame 6 should have been a reply from H2, instead it shows the next packet sent by Scapy). This is because the sequence number no longer falls within the sliding window. After the sequence number gets increased by one the number is correct and the packet is considered valid.

Afterwards, the registers are checked and show that the sequence number of the session identifier used to sent the packets has been increased, while the other register has not changed.

Listing 5.3: Testing packets with session identifier 123456789

```
# Sending 3 packets
> packet_send 123456789 0xabcd 123
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  123, checksum: 0xcf45, payload: 683

response: 683

> packet_send 123456789 0xabcd 123
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  123, checksum: 0xcf45, payload: 662

no response

> packet_send 123456789 0xabcd 124
#### sending 1 packet ####
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  124, checksum: 0xcf44, payload: 360

response: 360

# Src          => Dst         Prot Type Data     ID          Seq Chksum
3 10.0.0.10 => 10.0.1.10 GRE  ICMP 683     0x075bcd15 123 0xcf45
4 10.0.1.10 => 10.0.0.10 ICMP RPLY 683
5 10.0.0.10 => 10.0.1.10 GRE  ICMP 662     0x075bcd15 123 0xcf45
6 10.0.0.10 => 10.0.1.10 GRE  ICMP 360     0x075bcd15 124 0xcf44
7 10.0.1.10 => 10.0.0.10 ICMP RPLY 360

# Register values
RuntimeCmd: register_read sequence_number_reg 0
sequence_number_reg[0]=  124

RuntimeCmd: register_read sequence_number_reg 1
sequence_number_reg[1]=  0
```

5.2.3 Session identifier 2

Next the other session identifier known by the switch is tested (Listing 5.4). First a packet is sent using `987654321`, `0x1234` and `125` as parameters. This packet is dropped because the sequence number lies outside the window. Then the sequence number is lowered to `124` and this time a response is received successfully. After the window has been updated `125` is now considered valid. Next the sequence number is incremented by 125 resulting in a packet with sequence number `250`. This again fails because once more the number falls outside the window. After lowering the sequence number to `249` the packet is accepted.

Listing 5.4: Testing packets with session identifier 987654321

```
# Sending 5 packets
> packet_send 987654321 0x1234 125
sending packet: identifier: 987654321 (0x3ade68b1), hash key: 0x1234, sequence number:
 ↪  125, checksum: 0x99be, payload: 783

no response

> packet_send 987654321 0x1234 124
sending packet: identifier: 987654321 (0x3ade68b1), hash key: 0x1234, sequence number:
 ↪  124, checksum: 0x99bf, payload: 454

response: 454

> packet_send 987654321 0x1234 125
sending packet: identifier: 987654321 (0x3ade68b1), hash key: 0x1234, sequence number:
 ↪  125, checksum: 0x99be, payload: 928

response: 928

> packet_send 987654321 0x1234 250
sending packet: identifier: 987654321 (0x3ade68b1), hash key: 0x1234, sequence number:
 ↪  250, checksum: 0x9941, payload: 324

no response

> packet_send 987654321 0x1234 249
sending packet: identifier: 987654321 (0x3ade68b1), hash key: 0x1234, sequence number:
 ↪  249, checksum: 0x9942, payload: 831

response: 831

# Src         => Dst        Prot Type Data    ID         Seq Chksum
 8 10.0.0.10 => 10.0.1.10 GRE  ICMP 783     0x3ade68b1 125 0x99be
 9 10.0.0.10 => 10.0.1.10 GRE  ICMP 454     0x3ade68b1 124 0x99bf
10 10.0.1.10 => 10.0.0.10 ICMP RPLY 454
11 10.0.0.10 => 10.0.1.10 GRE  ICMP 928     0x3ade68b1 125 0x99be
12 10.0.1.10 => 10.0.0.10 ICMP RPLY 928
13 10.0.0.10 => 10.0.1.10 GRE  ICMP 324     0x3ade68b1 250 0x9941
14 10.0.0.10 => 10.0.1.10 GRE  ICMP 831     0x3ade68b1 249 0x9942
15 10.0.1.10 => 10.0.0.10 ICMP RPLY 831
```

5.2.4 Session identifier 3

New sessions can be added to the switch at runtime.[3] In Listing 5.5 first an attempt is made at sending a packet with the identifier `0x00000042`. Without this session being known by the switch, it fails. After the entry has been added to the `gre_key` table the next attempt succeeds.

Listing 5.5: Testing packets with session identifier 0x00000042

```
# Sending the packet without the session identifier configured
> packet_send 0x00000042 0x0024 123
sending packet: identifier: 66 (0x42), hash key: 0x24, sequence number: 123, checksum:
 ↪  0x4f1e, payload: 263

no response

# Configure new session by adding table entry
RuntimeCmd: table_add gre_key set_gre_key_valid 0x00000042 => 2 0x0024
Adding entry to exact match table gre_key
match key:           EXACT-00:00:00:42
action:              set_gre_key_valid
runtime data:        00:02  00:24
Entry has been added with handle 2

# Sending the packet again
> packet_send 0x00000042 0x0024 123
#### sending 1 packet ####
sending packet: identifier: 66 (0x42), hash key: 0x24, sequence number: 123, checksum:
 ↪  0x4f1e, payload: 725

response: 725

# Src         => Dst       Prot Type Data    ID         Seq Chksum
16 10.0.0.10 => 10.0.1.10 GRE  ICMP 263     0x00000042 123 0x4f1e
17 10.0.0.10 => 10.0.1.10 GRE  ICMP 725     0x00000042 123 0x4f1e
18 10.0.1.10 => 10.0.0.10 ICMP RPLY 725
```

5.2.5 Forging of packets

Packets can be forged by the Scapy script in different ways. Two methods are shown. The first method is to use the wrong MAC key. In Listing 5.6 the keys `0xabce`, `0x1234` and `0xabcd` are used of which only the right key results in the packet being forwarded by the switch.

Listing 5.6: Forging the key

```
# Register holds same value previously recorded
RuntimeCmd: register_read sequence_number_reg 0
sequence_number_reg[0]=  124

# Sending packet with key set to 0xabce
> packet_send 123456789 0xabce 125
```

[3] Actually, all sessions are configured at runtime.

```
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabce, sequence number:
 ↪  125, checksum: 0xcf42, payload: 887

no response

# Sending packet with key set to 0x1234
> packet_send 123456789 0x1234 125
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0x1234, sequence number:
 ↪  125, checksum: 0x68dd, payload: 341

no response

# Sending packet with key set to 0xabcd
> packet_send 123456789 0xabcd 125
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  125, checksum: 0xcf43, payload: 288

response: 288

# Src         => Dst        Prot Type Data    ID         Seq Chksum
20 10.0.0.10 => 10.0.1.10 GRE  ICMP 341     0x075bcd15 125 0x68dd
21 10.0.0.10 => 10.0.1.10 GRE  ICMP 288     0x075bcd15 125 0xcf43
22 10.0.1.10 => 10.0.0.10 ICMP RPLY 288
```

The other method is to append data to the payload after the simulated MAC has been calculated. This is done by adding the 1 parameter. Listing 5.7 shows that when the payloads are forged the packets are not be accepted.

Listing 5.7: Forging the payload

```
> packet_send 123456789 0xabcd 126 1
forging payload '618' with '248'             appended
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 618248

no response

> packet_send 123456789 0xabcd 126 1
forging payload '481' with '769'             appended
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 481769

no response

> packet_send 123456789 0xabcd 126
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 476

response: 476

# Src         => Dst        Prot Type Data    ID         Seq Chksum
23 10.0.0.10 => 10.0.1.10 GRE  ICMP 618248  0x075bcd15 126 0xcf42
24 10.0.0.10 => 10.0.1.10 GRE  ICMP 481769  0x075bcd15 126 0xcf42
25 10.0.0.10 => 10.0.1.10 GRE  ICMP 476     0x075bcd15 126 0xcf42
26 10.0.1.10 => 10.0.0.10 ICMP RPLY 476
```

5.2.6 Peculiarities

During the development and testing of the P4 program several odd phenomena were observed. Some seem to stem from the software switch's quirks that were worked around during the project. Others were mainly unexpected and counter-intuitive manifestations that happen when working with the lower layers of the network stack and using protocols in ways unintended. Two of them are highlighted here.

The first peculiarity is that even though the payloads are randomised, the checksums stay the same when the GRE header is left unchanged (Listing 5.8). The cause for this strange behaviour is found in the ICMP header: changes to the ICMP payload are cancelled out by the ICMP checksum; when the ICMP data payload value is increased (e.g., from '1' to '2' resulting in the hexadecimal value '3200' instead of '3100')[4] the ICMP checksum "corrects" the difference by being decremented with the same difference (e.g., 'c6ff' becomes 'c5ff'). The result for the GRE checksum is that the changes to the ICMP layer become an invariant. Listing 5.9 shows this phenomenon by sending different packets via Scapy, capturing the frames and calculating the checksums (the GRE header starts at offset 0x22 and the ICMP header at 0x32).

Listing 5.8: Random payloads with same checksums

```
> packet_send 123456789 0xabcd 126
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 906

no response

> packet_send 123456789 0xabcd 126
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 465

no response

> packet_send 123456789 0xabcd 126
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 736

no response

> packet_send 123456789 0xabcd 126
sending packet: identifier: 123456789 (0x75bcd15), hash key: 0xabcd, sequence number:
 ↪  126, checksum: 0xcf42, payload: 282

no response

# Src         => Dst        Prot Type Data     ID          Seq Chksum
27 10.0.0.10 => 10.0.1.10 GRE  ICMP 906      0x075bcd15 126 0xcf42
28 10.0.0.10 => 10.0.1.10 GRE  ICMP 465      0x075bcd15 126 0xcf42
29 10.0.0.10 => 10.0.1.10 GRE  ICMP 736      0x075bcd15 126 0xcf42
30 10.0.0.10 => 10.0.1.10 GRE  ICMP 282      0x075bcd15 126 0xcf42
```

[4]The values are effectively padded with zeros because the checksum calculation uses 16-bit words.

Listing 5.9: Detailed view of the equalising effect of the ICMP checksum

```
# Send packet with ICMP payload 1
send(IP()/GRE(key_present=1,seqnum_present=1,chksum_present=1)/ICMP()/'1')

# Capture packet
0000   ff ff ff ff ff ff 00 00 00 00 00 00 08 00 45 00   ..............E.
0010   00 2d 00 01 00 00 40 2f 7c 9f 7f 00 00 01 7f 00   .-....@/|.......
0020   00 01 b0 00 00 00 4f ff 00 00 00 00 00 00 00 00   ......O.........
0030   00 00 08 00 c6 ff 00 00 00 00 31                  ..........1

# Calculate GRE checksum
b000+0000+0000+0000+0000+0000+0000+0000+0800+c6ff+0000+0000+3100 = 1+AFFF = ones(B000)
 ↪   = 4FFF

# Send packet with ICMP payload 2
send(IP()/GRE(key_present=1,seqnum_present=1,chksum_present=1)/ICMP()/'2')

0000   ff ff ff ff ff ff 00 00 00 00 00 00 08 00 45 00   ..............E.
0010   00 2d 00 01 00 00 40 2f 7c 9f 7f 00 00 01 7f 00   .-....@/|.......
0020   00 01 b0 00 00 00 4f ff 00 00 00 00 00 00 00 00   ......O.........
0030   00 00 08 00 c5 ff 00 00 00 00 32                  ..........2

b000+0000+0000+0000+0000+0000+0000+0000+0800+c5ff+0000+0000+3200 = 1+AFFF = ones(B000)
 ↪   = 4FFF

# Send packet with ICMP payload 4213212131312121 and ICMP identifier 1
send(IP()/GRE(key_present=1,seqnum_present=1,chksum_present=1)/ICMP(id=1)/'42132121313121
 ↪   21')

0000   ff ff ff ff ff ff 00 00 00 00 00 00 08 00 45 00   ..............E.
0010   00 3c 00 01 00 00 40 2f 7c 90 7f 00 00 01 7f 00   .<....@/|.......
0020   00 01 b0 00 00 00 4f ff 00 00 00 00 00 00 00 00   ......O.........
0030   00 00 08 00 63 72 00 01 00 00 34 32 31 33 32 31   ....cr....421321
0040   32 31 33 31 33 31 32 31 32 31                     2131312121

b000+0000+0000+0000+0000+0000+0000+0000+0800+6372+0001+0000+3432+3133+3231+3231+3331+3331
 ↪   +3231+3231 = 2+AFFE = ones(B000) = 4FFF
```

Another unexpected result is that the sniffer positioned at the right-hand side of the switch (not shown in the listings above) stops capturing when a GRE packet is sent that cannot be verified correctly. Tcpdump shows that the software switch literally presents an empty Ethernet frame to the interface. Apparently truncating the packet to zero in order to drop it in case of a GRE header has an unexpected side effect. This results in tshark exiting when it sees such a malformed Ethernet frame, but only when tshark (or Wireshark) is listening on a single interface. Further inspection shows that an error is printed via the kernel ring buffer. To work around this problem tshark is automatically restarted but this action is visible via the frame counter.

During publication of the final code, different behaviour was seen: an error message gets printed that the packet size is too short and the sniffer does no longer stop capturing because the erroneous frame is no longer presented. A possible explanation for this differing in behaviour is that an updated dependency (e.g., the Mininet package) now accounts for frames smaller than an allowed minimum.

Listing 5.10: Truncation quirk

```
# Sniffer S1-ETH2 - H2
# Src         => Dst        Prot Type Data    ID         Seq Chksum
 1 10.0.0.10 => 10.0.1.10 ICMP RQST 725
 2 10.0.1.10 => 10.0.0.10 ICMP RPLY 725
 1 10.0.0.10 => 10.0.1.10 ICMP RQST 288
 2 10.0.1.10 => 10.0.0.10 ICMP RPLY 288
 1 10.0.0.10 => 10.0.1.10 ICMP RQST 476
 2 10.0.1.10 => 10.0.0.10 ICMP RPLY 476

# Tcpdump output
12:31:52.266216 [|ether]

# Kernel ring buffer error message
protocol 0003 is buggy, dev s1-eth2

# Alternative error message
lt-simple_route: packet size is too short (0 < 14)
```

6 Discussion and recommendations

In this chapter the insights obtained during the project are discussed. First the architecture of the CoCo VPN service and the authentication protocol is looked into. Then the proposed authentication scheme and protocol are examined in the light of the P4 proof of concept. Furthermore, the state of the P4 language and its ecosystem are reflected upon.

6.1 CoCo architecture and authentication scheme

During the project it became apparent that the CoCo VPN architecture is still in an early phase. Currently, the user interaction and routing aspects have been developed in the form of a prototype. However, user authentication was considered out of scope up to now. The user agent is not defined nor is clear what the cryptographic means will comprise. For instance, it has not been determined how user identities will be dealt with in a multi-domain setting or how the key material will be distributed. This makes reasoning about the required authentication scheme and its protocol a difficult task. As a result no clear and definite recommendations can be given during this project in that regard. Instead we need to constrain ourselves to guidelines and recommendations that need further exploration.

In terms of complexity it appears wise to keep the authentication scheme simple. By confining most of the complexity in the CoCo controller the implementation and maintenance of the P4 program is likely to be more straightforward. Especially for the early generations of P4 hardware it is conceivable that the lack of maturity may result in less than ideal conformance to the specification, possibly giving rise to compatibility issues between targets (e.g., requiring maintenance of separate versions of the same program). It is expected that by keeping the P4 components simple, development and troubleshooting will be easier, leading to a system that performs better.

A thing that is likely to complicate the implementation and result in a complex system is the scope of the system. Currently it is envisioned that VPN users may be mobile and use the system from home. For the authentication scheme this means that it needs to be able to identify the user using different network identifier and possibly differentiate users that use the same identifier due to NAT. This has implications for the authentication protocol which need to be considered carefully. It might be better to refrain from supporting too many use cases, otherwise a too complex system could be the result. For instance, using tunnels for home users instead of a purely routed solution could lower the requirements of the scheme and alleviate the burden of developing and maintaining it.

Another argument for keeping the solution simple is that its intended use case is high performance computing. This consideration led to the idea to pre-provision the P4 switch with all session information and key material. This way session roll-overs could be handled in the middle of transfers without any performance degradation. The downside is that this solution has scalability issues. Especially when authentication information is distributed across different domains the constrained amount of memory of the network devices could pose a problem.

In the current envisioned use case it is expected that the key material can be kept local to the domain, but if end-to-end authentication is desirable this is no longer an option. In that case it might be a solution to forward packets that contain an unknown session identifier to the CoCo controller in an OpenFlow manner (i.e. the controller inspects packets with unknown identifiers and installs an entry on-demand). As an optimisation it could be decided to only provision the switches with the key material of sessions that are expected to flow through the switch. The rest of the sessions are only installed on-demand when a flow hits the P4 switch. This does have implications for the complexity of the system and the performance characteristics. Further research into P4 solutions that make this possible and careful consideration are advised.

During the project the concept of sending a packet with an authentication header, which is removed by the network after it has been verified, has been successfully implemented in P4. However, even though the providers should be trustworthy (the architecture provides no confidentiality), this scheme will likely lead to a suboptimal system. The concern stems from the asymmetry of the flows: packets with an IPsec AH or UDP authentication header are sent and the response could be an ICMP or Transmission Control Protocol (TCP) packet. This is likely to pose a problem for middleboxes (e.g., a firewall or NAT device) and could impede monitoring and troubleshooting. A solution could be to not remove the header used for authentication or restore it when it exits the last PE on its path. Possibly a generic or local session identifier could be used such that authentication information can still be kept local and the scheme kept simple.

Implementation of the CoCo client (the user agent that adds the authentication information) and the authentication protocol are other parts of the architecture that need to be studied further. If an existing protocol is used (like the IPsec AH) then the accompanied software could be used too, possibly as a basis, to implement the CoCo client. However, this could lead to violations of the leveraged protocol. Customisation of the protocol and software could lead to compatibility issues. If the VPN service gets deployed across more than a hand full of organisations this could be an indefensible design decision. If IPsec AH is selected as the authentication protocol it is likely best to refrain from customising the standard. However, a subset could be defined to ease the implementation efforts.

In case a lower-layer protocol is used (like the IPsec AH) or created, mobility and home use are harder to support than is likely the case with a higher-layer protocol (like UDP). However, higher-layer protocols are likely to impose a higher overhead and could be more difficult to implement in P4 because of their generic use case; an identifier needs to be used (e.g., a specific port number) or part of the payload needs to be parsed in order to determine whether the packet contains a CoCo authentication header. Further insight in the CoCo use cases and research into the flexibility of parsing headers in P4 is necessary to determine the best method.

Likewise, more research is necessary to decide upon the most suitable authentication method. Especially the cryptographic means (e.g., hashing algorithm) needs to be considered in terms of provided security but also feasibility of P4 target support and runtime requirements. AES and SHA are likely candidates, however it would be very interesting to compare these algorithms with SipHash. Due to its high performance, low complexity and suitability for (short) network packets, SipHash might be a better choice. However, the algorithm appears less versatile in terms of cryptographic purposes and since it is not part of cipher suites it is not a likely candidate for widespread inclusion in the P4 targets.

Besides complications in terms of firewalls and NAT, other possible issues have been identified. For instance, IP fragmentation would necessitate that the P4 switch identifies, buffers and reassembles fragmented packets so that the MAC can be computed correctly. It is unlikely that P4 will provide the facilities necessary to do this; in P4 programming the parser is limited to defining fields and extracting headers of a single packet, it does not standardise control over the queues[10, p. 76] which might otherwise be used for buffering fragments, and in IPv6 routers do not fragment packets lowering the need for such a facility).[36, p. 18] Thus, it is likely that this functionality would need to be provided by the target making its availability uncertain. Another problem for computing the MAC is the switch's packet forwarding mode; in cut-through mode the switch starts forwarding the packet before it has been completely received. Since it is expected that the PE is run in layer 3 mode this is unlikely to pose a problem.

Another issue that might arise is that multiple PE switches might be deployed in a load-balance or back-up scenario. Since state about the sessions (the sequence numbers) is kept locally on the switch a split-brain problem might arise. A solution might be to use large windows and periodically transfer the state to a controller. Alternatively, each switch could directly inform other switches, either after each packet or a when a predetermined threshold is reached. Still, it means that state needs to be transferred and failure modes require more attention. For example, the client could consider the established session invalid after it has not received a reply of the other VPN endpoint after a certain amount of attempts. On the other hand, the controller should ensure that the switches are in the right state; state of the switches must be tracked and stale session information should be removed.

To accommodate very high-performance applications it is advisable to use 64-bit sequence numbers. This can be done either by allotting a 64-bit field for this purpose at the cost of higher header overhead, or via the same concept as the Extended Sequence Numbers of the IPsec protocol at the cost of more complexity. Implementing the latter in P4 is likely to be feasibly using an extra register per tracked session, but remains a future research topic. Likewise, encoding options into the session identifier is likely possible via a longest-prefix or masked match, or options could be supplied via metadata by the controller. With these topics unresearched the best method for indicating session options remains to be determined.

Because the CoCo VPN architecture needs to differentiate users it has to include methods for exchanging key material to support authentication. And since the service is intended to support eScience — which will likely involve sensitive information — there is no good reason to not develop it with end-to-end encryption in mind (besides to keeping the system simple). Solving confidentiality at the network layer is a complex task, but once the key distribution system is in place adding encryption at layer 3 is likely to be more successful than adapting every application with a Transport Layer Security (TLS) subsystem.

If it is decided to implemented end-to-end authentication and encryption it might be a good idea to use separate keys for authentication and encryption. The authentication keys could then be known by the network while the encryption keys could be only known by the endpoints. That way the network can authenticate the VPN traffic and limit misuse early, while the end users can be ensured of the confidentiality of their traffic. However, the ideal situation where the network does not need to trust its users and vice versa would require a more sophisticated cryptography scheme.

6.2 P4 language

The experiences achieved during the project show that P4 language is in a fairly good shape considering its relatively immature state. The software targets are still in development, and while there were several quirks or even bugs, it can be considered reasonably stable. It is to be expected that while the software switch is in active development, its stability will vary over time and that the issues found during the project are of a volatile kind.

Even for an inexperienced programmer, like the author of this report, the P4 Language Specification is remarkably clear and fairly unambiguous. Since the language is being developed and version 1.1 of the specification is currently in draft form, it is inevitable that inconsistencies and functionality gaps exist. Examples are the description of stateful memories and the unspecified `register_read` and `register_write` primitive actions.

Via the proof of concept the main requirements for implementing authentication of network traffic in P4 are shown to be feasible. The 1.0 version of the P4 Language Specification imposes several impracticalities, mainly in terms of the bit width of the data type allowed in action parameters. This limitation should be overcome with version 1.1 of the language, which allows for more expressive declaration of data types. At this moment there are no suitable cryptographic means e.g., algorithms and key input functions, for ensuring that authentication will be secure. To certain extent it is expected that the P4 hardware targets can and will be enabled to provide in these means. First, the software targets and possibly the P4 compiler need to be enhanced such that a secure authentication scheme can be tested realistically.[1]

Of great assistance were the many software targets, available in the git repository, which provided practical code examples. What is a bit confusing is that there are software targets that are named the same, but consist of differing code. Possibly, development has shifted to newer versions and the older versions are still available. Another confusing aspect is that every software target can have different auxiliary tools and ways of interacting with it. For instance, not all targets in the P4factory appear to be enabled for usage with the second software switch and the targets of the behavioral model 2 have their supported primitive actions defined separately.

[1]Possible starting points for adding the cryptographic means to the P4 source code are as follows:

- the behavioral model 1: `p4c-behavioral/p4c_bm/templates/src/checksums.c` and `p4c-behavioral/p4c_bm/templates/src/checksums_algos.h`
- the behavioral model 2: `behavioral-model/src/bm_sim/checksums.cpp` and `behavioral-model/src/bm_sim/calculations.cpp`;
- the compiler: `p4c-bm/p4c_bm/gen_json.py`;
- and the P4 high-level intermediate representation: `p4-hlir/p4_hlir/frontend/dumper.py`

7 Conclusion

During the project, requirements for adding authentication to the CoCo VPN service have been explored and a P4 program comprising a simplified authentication scheme has been developed. Through a proof of concept the mechanisms required for authentication of network traffic via the P4 language are shown to be feasible; sessions can be manipulated at runtime and via a simulated MAC (a 'keyed CRC') multiple sessions can be independently authenticated. However, the necessary cryptographic means cannot be implemented using the P4 language, but instead require support of the P4 target. Immaturity of the language and per-target limitations are likely to impose hurdles for a successful deployment, especially for the first generation(s) of P4 hardware. A possible mitigation of the expected problems is to keep the authentication scheme and the P4 program simple. An example is to pre-provision the switch that authenticates the traffic with all the established sessions and related key material. This enables the high-performance applications of the VPN service but imposes limits to the scalability.

As a more general conclusion, adding authentication of network traffic to the CoCo VPN architecture knows many caveats. NAT, IP fragmentation and packet forwarding modes need to be taken into account when designing the authentication scheme. The most cost-effective solution to these issues is to only support a limited amount of problematic use cases within the protocol or even the VPN itself, while tunnelling can be considered at the cost of performance. The most appropriate authentication protocol depends on the use case and the deployment scale. While the IPsec AH provides in all the required means, creating a custom protocol is achievable through P4 and might offer a more flexible solution.

During the project, an authentication scheme consisting of asymmetrical flows was used as a 'reference design'. While implementing this scheme in P4 is shown feasible and has a few inherently favourable characteristics in terms of complexity and scalability, it might cause problems for actual deployments. The main concern of the scheme is that the removal of the authentication header will likely make the flows harder to firewall and monitor. As a solution end-to-end authentication, or another form that results in symmetrical flows, is encouraged. Unfortunately, this has the potential to result in a more complex system, therefore the final solution should be designed with these trade-offs in mind.

8 Future research

As future work the following topics are proposed:

- During the project it has become apparent that the cryptographic means necessary cannot be implemented directly into P4 itself. Instead the targets should provide in these functions. For a follow-up project it would be interesting to implement a MAC algorithm in the software switch so that a more realistic proof of concept can be created. This work could also include the comparison and selection of the most appropriate algorithm to be used for the authentication scheme and to be implemented in actual targets. During the project a preliminary selection of algorithms has been made. This selection includes the SipHash algorithm, which does not seem to have been subjected to extensive cryptanalysis.
- The authentication scheme envisioned for this project requires enhancements of the CoCo VPN architecture. The CoCo agent needs to implement a P4-facing API while the CoCo portal needs to allow session establishment with a to-be-developed CoCo client. Development of this authentication scheme and its subsystems will likely require further research of the P4 capabilities. The intended use cases of the VPN service could also be reconsidered to increase the chances of a successful implementation and deployment. For instance, offering high performance and a high level of security while supporting home and mobile use cases could result in a system that is too complex. Either restrictions or solutions for these use cases could be topics of future research. Once user authentication on a network device is in place, end-to-end authentication and encryption are desirable features. Even though these will introduce more complexity, designing the CoCo VPN architecture should be done while keeping these in mind. Whether and how these features can be supported best could be part of future research.

Bibliography

[1] Ronald van der Pol et al. "Assessment of SDN technology for an easy-to-use VPN service". In: *Future Generation Computer Systems* 56 (2016), pp. 295–302 (cit. on pp. 4, 7).

[2] Pat Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95 (cit. on p. 4).

[3] Vainius Dangovas and Feliksas Kuliesius. "SDN-driven authentication and access control system". In: *The International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*. Society of Digital Information and Wireless Communication. 2014, p. 20 (cit. on p. 4).

[4] Jianfeng Zou et al. "Design and implementation of secure multicast based on SDN". In: *Broadband Network & Multimedia Technology (IC-BNMT), 2013 5th IEEE International Conference on*. IEEE. 2013, pp. 124–128 (cit. on p. 4).

[5] Hyojoon Kim and Nick Feamster. "Improving network management with software defined networking". In: *Communications Magazine, IEEE* 51.2 (2013), pp. 114–119 (cit. on p. 4).

[6] Dongting Yu et al. "Authentication for resilience: the case of SDN". In: *Security Protocols XXI*. Springer, 2013, pp. 39–44 (cit. on p. 4).

[7] Anirudh Sivaraman et al. "DC. p4: programming the forwarding plane of a data-center switch". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM. 2015, p. 2 (cit. on p. 4).

[8] Huynh Tu Dang et al. "Paxos Made Switch-y". In: *arXiv preprint arXiv:1511.04985* (2015) (cit. on p. 4).

[9] Nick McKeown and Jen Rexford. *Clarifying the differences between P4 and OpenFlow*. P4 Language Consortium. URL: http://p4.org/p4/clarifying-the-differences-between-p4-and-openflow/ (visited on 07/08/2016) (cit. on pp. 9, 10).

[10] *The P4 Language Specification*. Version 1.1.0. The P4 Language Consortium. Jan. 27, 2016. 124 pp. URL: http://p4.org/wp-content/uploads/2016/03/p4_v1.1.pdf (cit. on pp. 9, 10, 14, 27, 28, 31–35, 48).

[11] J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. O'Reilly Media, 2002. ISBN: 9780596551971. URL: https://books.google.nl/books?id=IIqwAy4qEl0C (cit. on p. 11).

[12] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (Informational). Updated by RFC 6151. Internet Engineering Task Force, Feb. 1997. URL: https://tools.ietf.org/html/rfc2104 (cit. on pp. 12, 33).

[13] S. Turner and L. Chen. *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. RFC 6151 (Informational). Internet Engineering Task Force, Mar. 2011. URL: https://tools.ietf.org/html/rfc6151 (cit. on p. 12).

[14] S. Frankel and H. Herbert. *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec*. RFC 3566 (Proposed Standard). Internet Engineering Task Force, Sept. 2003. URL: https://tools.ietf.org/html/rfc3566 (cit. on p. 12).

[15] D. McGrew and J. Viega. *The Use of Galois Message Authentication Code (GMAC) in IPsec ESP and AH*. RFC 4543 (Proposed Standard). Internet Engineering Task Force, May 2006. URL: https://tools.ietf.org/html/rfc4543 (cit. on pp. 12, 15).

[16] Stefan Lemsitzer et al. "Multi-gigabit GCM-AES architecture optimized for FPGAs". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2007, pp. 227–238 (cit. on p. 12).

[17] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 7539 (Informational). Internet Engineering Task Force, May 2015. URL: https://tools.ietf.org/html/rfc7539 (cit. on p. 13).

[18] Daniel J Bernstein. "The Poly1305-AES message-authentication code". In: *Fast Software Encryption*. Springer. 2005, pp. 32–49 (cit. on p. 13).

[19] *SipHash: a fast short-input PRF*. May 27, 2016. URL: https://131002.net/siphash/ (visited on 06/13/2016) (cit. on p. 13).

[20] Christian Heimes. *PEP 456 – Secure and interchangeable hash algorithm*. Sept. 27, 2013. URL: https://www.python.org/dev/peps/pep-0456 (visited on 06/13/2016) (cit. on p. 13).

[21] Jean-Philippe Aumasson and Daniel J Bernstein. "SipHash: a fast short-input PRF". In: *Progress in Cryptology-INDOCRYPT 2012*. Springer, 2012, pp. 489–508 (cit. on p. 13).

[22] M.S. Merkow and J. Breithaupt. *Information Security: Principles and Practices*. Prentice Hall Security Series. Pearson Prentice Hall, 2006. ISBN: 9780131547292. URL: https://books.google.nl/books?id=fTAkAQAAIAAJ (cit. on p. 14).

[23] Niels Ferguson and Bruce Schneier. "A cryptographic evaluation of IPsec". In: *Counterpane Internet Security, Inc* 3031 (2000) (cit. on p. 14).

[24] Paul Wouters. *Libreswan - Interoperability*. Nov. 4, 2015. URL: https://libreswan.org/wiki/Interoperability (visited on 06/12/2016) (cit. on p. 14).

[25] R. Kent. *IP Authentication Header*. Dec. 2005. URL: https://tools.ietf.org/html/rfc4302 (cit. on p. 15).

[26] J. Schiller. *Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2)*. Dec. 2005. URL: https://tools.ietf.org/html/rfc4307 (cit. on p. 15).

[27] S. Kelly and S. Frankel. *Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec*. May 2007. URL: https://tools.ietf.org/html/rfc4868 (cit. on p. 15).

[28] Y. Nir. *ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec*. Aug. 2015. URL: https://tools.ietf.org/html/rfc7634 (cit. on p. 15).

[29] Ed. P. Eronen. *IKEv2 Mobility and Multihoming Protocol (MOBIKE)*. June 2006. URL: https://tools.ietf.org/html/rfc4555 (cit. on p. 16).

[30] *Security Parameters Index (SPI) Parameters*. IANA. URL: https://www.iana.org/assignments/spi-numbers/spi-numbers.xhtml (visited on 06/21/2016) (cit. on p. 18).

[31] S. Hanks et al. *Generic Routing Encapsulation (GRE)*. RFC 1701 (Informational). Internet Engineering Task Force, Oct. 1994. URL: https://tools.ietf.org/html/rfc1701 (cit. on p. 22).

[32] D. Farinacci et al. *Generic Routing Encapsulation (GRE)*. RFC 2784 (Proposed Standard). Updated by RFC 2890. Internet Engineering Task Force, Mar. 2000. URL: https://tools.ietf.org/html/rfc2784 (cit. on pp. 22, 23).

[33] K. Hamzeh et al. *Point-to-Point Tunneling Protocol (PPTP)*. RFC 2637 (Informational). Internet Engineering Task Force, July 1999. URL: https://tools.ietf.org/html/rfc2637 (cit. on p. 22).

[34] G. Dommety. *Key and Sequence Number Extensions to GRE*. RFC 2890 (Proposed Standard). Internet Engineering Task Force, Sept. 2000. URL: https://tools.ietf.org/html/rfc2890 (cit. on p. 22).

[35] *Behavioral Model Repository*. GitHub. URL: https://github.com/p4lang/behavioral-model (visited on 07/04/2016) (cit. on p. 23).

[36] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. Internet Engineering Task Force, Dec. 1998. URL: https://tools.ietf.org/html/rfc2460 (cit. on p. 48).

Acronyms

ACL

access control list 8, 19

AES

Advanced Encryption Standard 12–15, 47

AH

Authentication Header 14–18, 47, 50

API

application programming interface 6, 9, 14, 51

BGP

Border Gateway Protocol 6

CBC

cipher block chaining 12

CoCo

Community Connection ii, 4–8, 11–20, 23, 26, 46–48, 50, 51

CRC

cyclic redundancy check 33, 50

DoS

Denial of Service 11, 13, 14, 20, 28

ESN

Extended Sequence Number 15, 19

ESP

Encapsulating Security Payload 14, 15

GRE

Generic Routing Encapsulation 22, 23, 26–30, 33, 35, 36, 43, 44

HMAC

hash-based message authentication code 12, 13

ICMP

Internet Control Message Protocol 23, 35, 36, 39, 43, 47

ICV

Integrity Check Value 15, 18

IKE

Internet Key Exchange 14–16

IP

Internet Protocol 4, 14–16, 18, 20, 23, 27, 33–36, 48, 50

IPsec

Internet Protocol Security 14–19, 22, 47, 48, 50

MAC

Media Access Control 4

MAC

message authentication code 11–15, 20, 22, 23, 26, 28, 29, 33, 34, 36, 39, 41, 42, 48, 50, 51

MOBIKE

IKEv2 Mobility and Multihoming Protocol 16

MPLS

Multiprotocol Label Switching 6

MTU

maximum transmission unit 18

NAT

network address translation 15–18, 46–48, 50

NREN

national research and education network 15

PE

provider edge 7, 8, 15, 19, 20, 47, 48

SA

Security Association 14–16, 18

SDN

software defined networking ii, 4, 6, 9

SHA

Secure Hash Algorithm 12–15, 47

SPI

Security Parameters Index 14, 16, 18

TCP

Transmission Control Protocol 47

TLS

Transport Layer Security 48

UDP

User Datagram Protocol 17, 18, 47

VPN

virtual private network ii, 4–8, 11, 12, 14–17, 19, 20, 26, 46–48, 50, 51