A New Authenticator

An Alternative to Binary Authentication Using Traffic Shaping

JIA GUO

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:299

A New Authenticator

An Alternative to Binary Authentication Using Traffic Shaping

Jia Guo
jiag@kth.se

20 November 2010

Master's Thesis

Examiner and Supervisor

Prof. Gerald Q. Maguire Jr.

Department of Communication Systems
School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden

Abstract

This thesis is part of a larger project on non-binary alternatives to authentication; in contrast to the binary authentication used in IEEE 802.1X and IEEE 802.11i. This thesis project seeks to define, implement, and evaluate a non-binary wireless access authentication mechanism. It introduces a new authenticator that implements such a new non-binary authentication mechanism.

In today's wireless local area networks it is generally not possible to continue a multimedia roaming session smoothly, because of the long delay caused by authentication – during which no traffic other than authentication traffic is permitted. In the best cases this high delay results in a long communication interruption interval without media, while in the worst cases the session is aborted by the higher layer application as the application believes that the connectivity is lost. Thus introducing a more appropriate authentication mechanism enables mobile users who move into a new wireless local area network cell to continue to send and receive packets with greatly reduced handover latency (in comparison to existing authentication mechanisms). This new authentication mechanism potentially enables seamless roaming for users of conversational multimedia services (for example, a voice over IP call could continue despite a movement from one cell to another).

This thesis demonstrates that it is possible to allow unauthenticated users to immediately begin to communicate, while simultaneously limiting their traffic. These limitations in traffic are implemented by traffic shaping. Additionally, using traffic shaping also offers a number of new possibilities – such as offering different qualities of service, allowing negotiation for different maximum bandwidths, etc.

Sammanfattning

Detta examensarbete är en del av ett större projekt om icke-binära alternativ till autentisering, i motsats till den binära autentiseringen används i IEEE 802.1X och IEEE 802.11i. Denna avhandling syftar till att definiera, genomföra och utvärdera en icke-binär trådlös åtkomst autentiseringsmekanism. Denna avhandling presenterar en ny dosa som implementerar en ny icke-binär autentiseringsmekanism.

Införa en mer ändamålsenlig mekanism för autentisering möjliggör mobila användare som flyttar in i ett nytt trådlöst lokalt nätverk cell att fortsätta att skicka och ta emot paket med kraftigt nedsatt överlämnandet fördröjning (i jämförelse med befintliga autentiseringsmekanismer). Denna nya autentiseringsmekanism potentiellt möjliggör sömlös roaming för användare av konversation multimedia tjänster (exempelvis kan en röst över IP-samtal fortsätter trots en rörelse från en cell till en annan). Tyvärr, i dagens lokala trådlösa nätverk fortsätter smidigt ett multimedium session är i allmänhet inte är möjligt, på grund av det långa dröjsmålet väntar på autentisering - då ingen trafik än autentisering är tillåten. I bästa fall kan detta hög fördröjning resultera i ett lång meddelande avbrott intervall utan medier, medan det i värsta fall sessionen avbryts av högre lager tillämpning som tillämpningsprogram anser att anslutning är förlorat.

Denna avhandling visar att det är möjligt att tillåta autentiserade användare som omedelbart börja kommunicera, samtidigt som begränsar deras trafik. Dessa begränsningar i trafiken genomförs av trafikformning. Dessutom använder trafikformning erbjuder också ett antal nya möjligheter - såsom att erbjuda olika kvaliteter av service, vilket gör förhandling för olika maximal bandbredd, osv.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

AID	association identifier
AP	access point
BSS	Basic service set, including STA and AP
BSSID	6 bytes long, MAC of AP in infrastructure mode; random number in IBSS
CFP	contention-free period
DHCP	dynamic host configuration protocol
DS	distribution system
EAP	extensible authentication protocol
EAPOL	EAP over LAN
ESS	extended service set, including several BSSs
IANA	Internet Assigned Numbers Authority
IAPP	Inter-Access Point Protocol
IBSS	independent BSS
ISP	inteternet service provider
MAC	media access control
Mbps	million bits per second
MPDU	MAC Protocol Data Unit
NAS	network access server
PACP	Port access control protocol
PAE	Port access entity
PHY	physical layer
PPP	point-to-point protocol
RTS	Request to Send
SSID	service set ID, an arbitrary string as the AP’s name
STA	station
UDP	user datagram protocol
VoIP	voice over IP
WEP	wire equivalent privacy
WISP	wireless inteternet service provider
WLAN	wireless local area network
WPA	Wi-Fi Protected Access

1 Introduction

Wi-Fi™ technology has been incoroporated by manufacturers in various mobile devices, such as laptops, personal digital appliances, and emerging WiFi phones, but users of this technology face a major practical hurdle - roaming[1]. After getting used to communicating through these devices inside a room, typically only a few meters from the access point, the users find that they can not easily go beyond this "room" (or nearby locations) because to do so the users will have to authenticate with a new access points, resulting in a kind of wireless "zoo". Hence manufacturers are trying to enable users to roam between different Wi-Fi access points (APs).

Seamless Wi-Fi roaming is promising since it is beneficial for all. End-users want consistent service experience across hotspots without knowledge of the mechanics of wireless access. They also want simple and safe login process no matter which authentication method is used. They may even hope that hotspots from public, home, or even enterprise can be integrated, so that a temporary, basic but urgent communication need can be satisfied despite the different service providers. On the other hand, for service providers and enterprises, Wi-Fi roaming could attract more customers and, as a result, generate new revenue streams. They also want to ensure an accurate and timely billing system. Network operators may even hope roaming users can automatcially utilize the most appropriate network - so as to reduce traffic jam and network device redundency while maximizing user satisfaction.

Unfortunately, realizing such benefits in today's wireless local area networks (WLANs) is generally not possible due to the lack of roaming support. For example, a voice over IP (VoIP) user will not experience a seamless handover as they move from one WLAN subnet to another, because of the long delay waiting for authentication – during which no traffic other than authentication data is permitted. This is because current authentication mechanism (using IEEE 802.1X [33] or IEEE 802.11i) is a *binary* authentication process. Here "binary" means that authentication **must be completed *before*** the user is able to send any non-authentication related traffic.

To elimiate this bottleneck, Professor Gerald Q. Maguire Jr., proposed a thesis project to take a deep look into the IEEE 802.1X protocol, study the actual need for authentication from the perspectives of both customers and service providers, and consider existing AP products and roaming solutions; then propose an innovative solution that allows both user authentication and normal network service *to **proceed in parallel*** during handover, thus leading to seamless Wi-Fi roaming. This thesis project is part of a larger project on non-binary alternatives to authentication, which consists of three sub-projects: (1) a new authenticator which utilizes traffic shaping to limit the amount of traffic from the supplicant – both *before* and *after* the supplicant is authenticated (the subject of this thesis), (2) to add bandwidth specifications to the responses from the authentication server [9], and (3) a new supplicant which can take advantage of the flexibility offered by this new authenticator and the ability to ask for different amounts of bandwidth (or potentially other parameters to be applied by the traffic shaper) [10].

[1] Wi-Fi is a trademark of the Wi-Fi Alliance, while initially focused on interoperability of IEEE 802.11b devices the group has expanded their scope in recent years..

This chapter first gives an overview of the architecture of a common wireless access point. Then it introduces our design and summaries the problems to be addressed. Solutions to these problems will be covered in detail in the remainder of this thesis.

1.1 Introduction to the Authenticator and our model

An authenticator is traditionally used by a network operator to decide if a supplicant should be allowed to utilize the network's resources. Figure 1-1 shows an example of a traditional binary authentication mechanism, where the authenticator controls a switch that initially connects the supplicant *only to the authenticator*. Figure 1-2 shows the proposed non-binary authentication mechanism. In this approach the binary switch is replaced by a traffic shaper. A continuous traffic shaping process replaces the binary authentication process, while at the same time maintaining traffic flows and redirecting authentication messages to the authenticator.

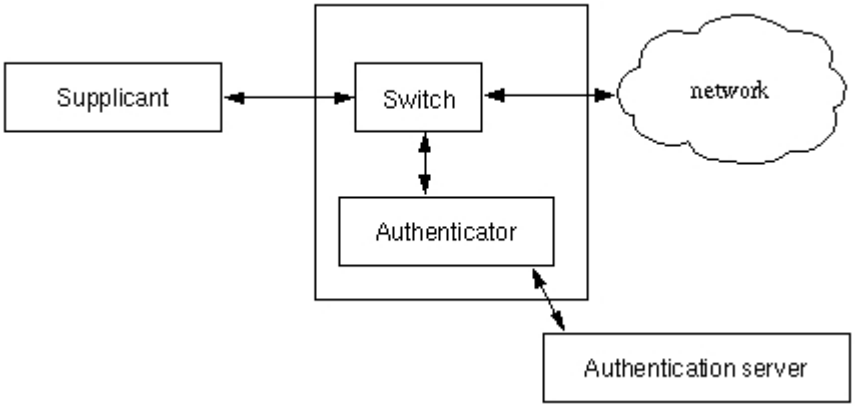

Figure 1-1: Traditional binary authentication

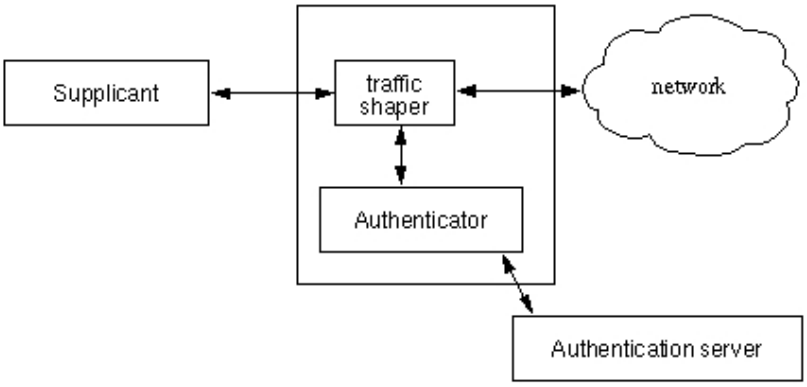

Figure 1-2. Non-binary authentication using a traffic shaper

For the remaineder of this thesis, unless explicitly stated, we assume that the network operator is a internet service provider. (ISP).

Unlike a binary switch, the traffic shaper allows traffic to continue to flow through an AP. Thus customers can carry on their business during a handover. However, end users crossing zones (cells operated by different operators) should be reminded that they are in a new Wi-Fi zone, therefore authentication and payment should be made. To achieve that, traffic shaping can limit the bandwidth of unauthenticated users, thus affect the degree of communication comfort. Ideally, the AP should be able to apply a blacklist and ban malicious users. As with existing APs, rather than doing authentication on its own, APs rely on remote authentication servers (RADIUS Server) for authentication service and relay all authentication messages between the authentication servers and the clients.

Although it looks like a simple change from a binary switch to a traffic shaper, things are not as simple as they seem. The new architecture must accommodate the following requirements: (1) allow the network to provide continuous services to supplicants (clients requesting authentication), so that customers will not have their traffic cut-off during a handover; (2) be acceptable to existing supplicant devices and authentication servers, so that both customers and service providers can preserve their investment and avoid a requirement for new software installation; (3) be compatible with the most commonly used authentication protocol IEEE 802.1X; and (4) leverage existing APs by upgrading software rather than installing new hardware. The following chapters will explain how to achieve these goals in a cost-effective manner.

1.2 Problem Statement

In addition to forwarding traffic (the basic function of an AP), the other function of a traditional AP is to authenticate end users. Thus traditionally the AP blocks all traffic to/from an unauthenticated user, except for traffic related to the authentication process. IEEE 802.1X defines a port based binary authentication scheme. Support for IEEE 802.1X is often implemented in the AP's software. Therefore, one way to implement the proposed new AP is to modify a traditional AP's software.

A traditional AP's work flow can be summaried as: (1) discover a new wireless device trying to access the network; (2) cut off the supplicant's incoming and outgoing traffic; (3) trigger a new authentication process whose traffic will flow between the supplicant, the authenticator, and the authentication server; and (4) if the supplicant is successfully authenticated then allow traffic to flow to-from it, otherwise block this device's traffic. In the case of our non-binary authentication AP, no traffic cut-off will occur, which makes for seamless handovers. To avoid traffic becoming free for all devices, we need to **encourage** devices to authenticate. How can we ensure that the AP only provides access to legitimate users (generally the ISP's subscribers)? This is achieved by using an intelligent traffic shaper. This traffic shaper initially limits the bandwidth of the new device to very low throughput. The device now has the possibility to send/receive traffic in parallel with its attempt to authenitcate itself. If the authentication is successful then the traffic shaper is informed to increase the throughput to this device (or perhaps to decrease the delay which this device's traffic has been experiencing). If the authenitcation was not successful (including authentication failure and free communication timed out), the traffic shaper will cut off the unauthorized user's traffic and block the traffic until it has successfully authenticated itself.

The diffrence in behavior between the proposed AP and a traditional AP is similar to the difference between purchasers using a credit card versus using a debit card. In the case of a purchase with a credit card, the credit card issuer guarantees that the merchant will be paid and bills the customer on a monthly basis for all of their consumption with this credit card; while in the case of a purchase with a debit card the customer's account is debited for the amount of the expenses at the time of the procurement. Just as the credit card issuer takes some risk that their customers may not pay their bill, the network operator takes a risk with the new AP that the users will use the AP to access the network with no intention of paying. A credit card issuer keeps track of customers who do not pay and does not give them credit in the future or charges them much higher for their outstanding debt.

Considering the above features of the new AP, we can further specify the problems that need to be solved from a technical point of view:

1. How to discover a new supplicant and tell this device's traffic from other traffic?
2. When and how to trigger a new authentication process?
3. How to distinguish authentication messages from other traffic?
4. How to relay the authentication packets between the supplicant and the authentication server?
5. How to define the limited time period *before* requiring authentication?
6. What is the risk of allowing traffic before successful authentication and how to mitigate this risk?

Keeping in mind the above technical problems, this thesis demonstrates a feasible way to seamlessly perform Wi-Fi roaming, while limiting the traffic of unauthenitcated devices. We assume that these limitations are enforced by a traffic shaper for business or policy reasons.

1.3 Limitations

This thesis project seeks to design an architecture for a non-binary wireless access authentication mechanism, to analyze potential technical barriers, and to propose solutions.

For the purpose of this thesis, the authenticator is only concerned with controlling the ability of a supplicant to access a network – potentially limiting this supplicant to a specific maximum bandwidth. Therefore, the thesis will explicitly consider the case of a WLAN supplicant associating with a WLAN access point. Additionally, the thesis assumes that a RADIUS server will be used as the authentication server.

These limitations should not be overly constraining – thus changing to a DIAMETER [47] or other authentication server should not require significant changes, but such changes are explicitly outside the scope of this thesis. Similarly applying this new authenticator to the case of an authenticator for a port controlled Ethernet switch should also not require a major effort, but also lies outside the scope of this thesis.

1.4 Organization of this thesis

Chapter **Error! Reference source not found.**will provide the background for both the hardware and software that will be necessary for understanding the rest of the thesis. Chapter 2 will also lay the foundation for the system architecture and give the design reason of the proposed solution, why we choose a specific hardware / software solution, how the non-binary authentication should be performed, and what further modifications should be made. Chapter 3 presents the proposed design. Chapter 4 4describes the evaluation of a prototype of the proposed solution. This is followed by a chapter that gives some conclusions and suggests future work. The appendixes contain the relevant source code used in Chapter 3.

2 Background

This chapter provides the reader with the background information necessary to understand the rest of the thesis. It begins by introducing the relevant IEEE standards. The RADIUS authentication, authorization, and accounting protocol will also be introduced and the message sequences required for a device to authenticate itself and receive authorization from a wireless local area network (WLAN) access point (AP) will be described in detail. Following this the *hostapd* module and the LINUX network filtering mechanisms will be described. This module and these filtering mechanisms will later be used as the basis for the solution proposed by this thesis to implement non-binary authentication.

2.1 IEEE 802.11 and 802.1X standards

The IEEE 802 family of standards deal with Local Area Network (LAN) and metropolitan area network communications, including the Ethernet family, token ring, wireless LANs (WLANs), wireless personal area networks, wireless metropolitan area network, bridging, and virtual bridged LANs[29]. These standards concern the data link layer and physical layer. Among these standards, those for WLAN were developed by the IEEE 802.11 working group.

The first WLAN standard (referred to as a base standard) IEEE 802.11 was ratified in 1997. It defined the physical layer (abbreviated PHY), the data link layer protocol, and frame format for WLAN. Many amendments came afterwards, aiming at higher data rates or enhanced security mechanisms. IEEE 802.11a [30] works in the 5 GHz band and offers data rates of up to 54 million bits per second (Mbps). IEEE 802.11b [31] works in 2.4 GHz band and offers data rates of up to 11 Mbps. As IEEE 802.11a and IEEE 802.11b use different portions of the radio spectrum, they are not compatible. Thus a new standard 802.11g [32] was defined with the high data rates of IEEE 802.11a, but backward compatibility with 802.11b and operating in the same frequency band as IEEE 802.11b. In order to merge these amendments (802.11a, b, d, e, g, h, i, j) with the base standard, IEEE 802.11-2007 [5] was approved on March 8, 2007. The most recent amendment is IEEE 802.11n published in October 2009, which adds multiple-input multiple-output (MIMO) and some other new features. **Error! Reference source not found.** shows the IEEE 802.11 protocol stack.

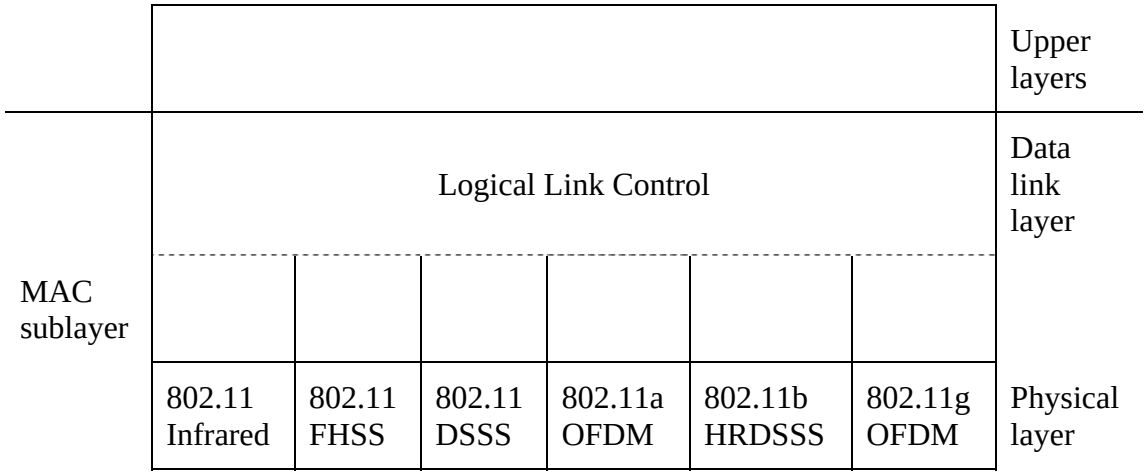

Figure 2-1. IEEE 802.11 Protocol Stack

For security, IEEE 802.11 defined Wired Equivalent Privacy (WEP). Unfortunately, this scheme turned out to be vulnerable to malicious tampering with messages and to replay attacks [11, 12, 23, 24]. Thus 802.1X was proposed for stronger security.

We assume that our mobile devices communicate according to the IEEE 802.11 standard. When the device enters a cell in a new domain it will perform conditional authentication following IEEE 802.1X. To understand the details of the handover procedure (for a device to change from using one cell to access the network to using another cell to access the network) it is important for us to be familiar with the details of these two protocols.

2.1.1 IEEE 802.11 Concepts

IEEE 802.11 [18] was defined to be a "wireless Ethernet"[2]. The standard refers to wireless stations (STAs) as the devices that utilize a IEEE 802.11 interface to communicate. The standard defines two modes of communication: a infrastructure mode and an *ad hoc* mode. In infrastructure mode each STA communicates via an access point (AP). Multiple APs can be connected into a LAN and interconnected with other networks. In ad hoc mode the STAs communicate directly between themselves. In all cases each IEEE 802.11 device competes for access to the media using a media access control (MAC) protocol. As we are only concerned with infrastructure mode, we will not consider *ad hoc* mode further.

We focus on APs that have an IEEE 802.11 interface and an IEEE 802.3 interface. These APs act as a bridge when forwarding packets from one interface to the other. The LAN interface receives and transmits IEEE 802.3 frames, while the wireless interface receives and transmits IEEE 802.11 frames. All protocols and data from higher layers are carried within the frame's body. In order to understand the details of the non-binary authentication we have to consider the interworking between these two kinds of frames in order to support the IEEE 802.1X authentication process.

Table 2-1 shows three different types of MAC frames in IEEE 802.11. Data frames are used for data transmission (i.e., sending traffic for higher layer protocols). Control frames are used for media access control. Management frames transmit management information, but are not forwarded to upper layers. Figure 2-2 shows the 802.11 frame format. As can be seen, the 802.11 frame format is more complicated than the IEEE 802.3 frame format.

Table 2-1. Types of IEEE 802.11 MAC Frames

Control Frame	RTS, CTS, ACK
Data Frame	
Management Frame	Beacon
	Probe Request, Probe Response
	Assoc Request, Assoc Response
	Reassoc Request, Reassoc Response
	Disassociation
	Authentication
	Deauthentication
	Announcement traffic indication frame

[2] Ethernet is a wired LAN physical and MAC layer specification that was the precusor to IEEE 802.3.

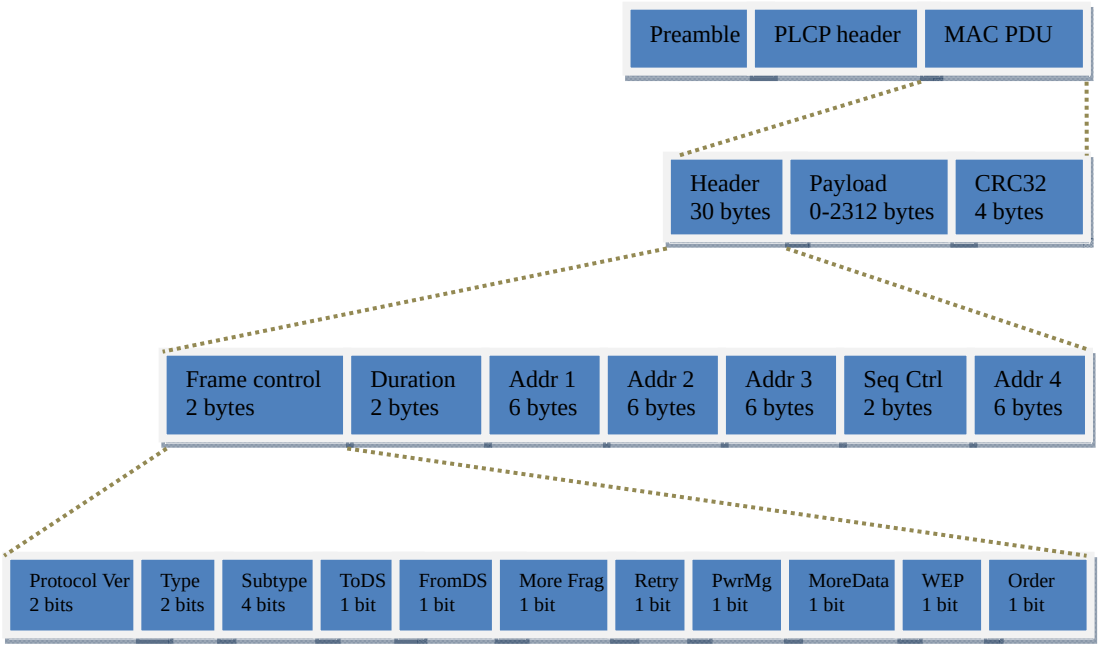

Figure 2-2. IEEE 802.11 Frame Format [19]

Considering the various fields shown in Figure 2-2 in a top to bottom order:

Preamble is PHY dependent, and includes to sub-fields: **Synch** and **SFD**. Synch is 80-bit long and is used by the PHY circuitry to select the appropriate antenna (if diversity is used), for steady-state frequency offset correction, and synchronization with the received packet. The Start Frame Delimiter (SFD) consists of a 16-bit binary pattern 0000 1100 1011 1101 which indicates the start of the PLCP header [34].

PLCP header is always transmitted at 1 Mbit/s and contains information used by the PHY layer to decode the frame. It consists of **PLCP_PDU Length Word**, **PLCP Signaling Field,** and a **Header Error Check Field**. PLCP_PDU Length Word indicates the number of bytes in the frame. The PLCP Signaling Field indicates the supported date rate, encoded in 0.5 MBps increments from 1 Mbit/s to 54 Mbit/s. The Header Error Check Field is a 16 Bit CRC error detection field [34].

Considering the MAC layer protocol data unit (PDU), the first field in the header is the frame control field. It consists of two bytes. The first two bits indicate the **Protocol Version** with a value of 00. **Type** and **Subtype** fields work together to specify the frame type. Table 2-2 shows the meaning on the various 6 bit combinations of Type and Subtype.

Table 2-2. Type and Subtype of Frame Control [34]

Type Value b3 b2	Type Description	Subtype Value b7 b6 b5 b4	Subtype Description
00	Management	0000	Association Request
00	Management	0001	Association Response
00	Management	0010	Reassociation Request
00	Management	0011	Reassociation Response
00	Management	0100	Probe Request
00	Management	0101	Probe Response
00	Management	0110 - 0111	Reserved
00	Management	1000	Beacon
00	Management	1001	ATIM
00	Management	1010	Disassociation
00	Management	1011	Authentication
00	Management	1100	Deauthentication
00	Management	1101..1111	Reserved
01	Control	0000..1001	Reserved
01	Control	1010	PS-Poll
01	Control	1011	RTS
01	Control	1100	CTS
01	Control	1101	ACK
01	Control	1110	CF End
01	Control	1111	CF End + CF-ACK
10	Data	0000	Data
10	Data	0001	Data + CF-Ack
10	Data	0010	Data + CF-Poll
10	Data	0011	Data + CF-Ack +CF-Poll
10	Data	0100	Null Function (no data)
10	Data	0101	CF-Ack (no data)
10	Data	0110	CF-Poll (no data)
10	Data	0111	CF-Ack + CF-Poll (no data)
10	Data	1000..1111	Reserved
11	Reserved	0000..1111	Reserved

ToDS is set to 1 if the frame is addressed to the AP for forwarding to the distribution system (DS), including the case when the destination station is in the same cell, i.e., within the Basic service set (BSS), but is not the AP itself. The bit is set to 0 in all other frames. **FromDS** is set to 1 when the frame is coming from the DS. Table 2-3 illustrates the 4 possible combinations of ToDS and FromDS with their corresponding meaning.

Table 2-3. ToDS and FromDS

ToDS	FromDS	Meaning
0	0	Data transmitted between two stations within the same BSS
1	0	Data sent to DS
0	1	Data received from DS
1	1	Wireless distribution system frame sent from one AP to other AP

More (**Fragments) Flag** is set to 1 when there are additional fragments belonging to the same frame following the current fragment. **Retry** is set to 1 indicating that this fragment is a retransmission of a previously fragment in order for the receiver to recognize duplicate transmissions that may occur when an acknowledgement packet is lost. **Power Management** indicates the Power Management mode that the station will be in after the transmission of this frame. This is used when the station changes from Power-Save to Active or vice versa; 1 means the station will operate in power-save mode and 0 means active. All frames from an AP have a power management value equal to 0; as the AP will remain active at all times. **More Data** is used by the AP to notify a STA which is operating in power-save mode that there are more frames buffered for transmission to this station. Given this information a STA may continue polling the AP for these buffered packets or the STA may to change to active mode. The **WEP** flag indicates if the frame body was encrypted by the WEP algorithm. **Order** indicates if this frame is being sent using the Strictly-Ordered service class. The Strictly-Ordered Service Class is defined for users that cannot accept a change of ordering between unicast frames and multicast frames (ordering of unicast frames to a specific address is always maintained) [34].

Following the Frame Control field is the **Duration / ID** field. This field can serve as an association identifier (AID) in Power-Save Poll messages for a station operating in power save to retrieve frames that are buffered for it at the AP. In all other frames, this field contains a duration value to update the Network Allocation Vector (NAV). Table 2-4 explains the meaning on bit of the 2 bytes long Duration / ID.

When bit 15 is zero, bits 14-0 represent the remaining duration of a frame exchange sequence after the frame in which the duration value is found. This value is used to upgrade the NAVs of other stations, preventing a station receiving this field from beginning a transmission that might cause corruption of the ongoing frame exchange sequence [35]. During a contention-free period (CFP) Duration / ID's value is set to 32768. In PS-Poll frames AID is a 16-bit field that contains an arbitrary number assigned to the station by the AP when it associates with a BSS. The numeric value in the least significant 11 bits (bits 0..10) are used by the mobile station to identify which bit in a traffic indication map [36] information element indicates that the access point has frames buffered for the mobile station [35].

Table 2-4. Duration / ID [35]

Bit 15	Bit 14	Bits 13-0	Usage
0	0..32767		Duration (after this frame, calculated in μs)
1	0	0	Fixed value (32768) during CFP
1	0	1..16383	Reserved
1	1	0	Reserved
1	1	0..2007	AID in PS-Poll frames
1	1	2008..16383	Reserved

In Address Fields, a frame may contain up to 4 Addresses depending on the ToDS and FromDS bits defined in the Control Field. The four possibilities are shown in Table 2-3.

Table 2-5. Meaning of Addresses According to To / FromDS [34]

To DS	From DS	Address1	Address 2	Address3	Address 4
0	0	DA	SA	BSSID	N/A
0	1	DA	BSSID	SA	N/A
1	0	BSSID	SA	DA	N/A
1	1	RA	TA	DA	SA

Address-1 is the Recipient Address. If ToDS is set, then this is the address of the AP; if ToDS is not set, then this is the address of the end-station.

Address-2 is the Transmitter Address. If FromDS is set, then this is the address of the AP; if it is not set, then it is the address of the Station.

Address-3 is in most cases the remaining, missing address. If FromDS is set to 1, then this is the original source Address; if ToDS is set, then this is the destination address.

Address-4 is used in the special case where a wireless wistribution system is used, and the frame is being transmitted from one AP to another, in this case both the ToDS and FromDS bits are set, so both the original destination and the original source Addresses are missing.

The Sequence Control field, located between Address-3 and Address-4, is used to indicate the order of different fragments belonging to the same frame, and to recognize duplicate packets. It consists of two subfields: Fragment Number and Sequence Number, which indicate the frame and the number of the fragment in the frame.

The last part of the MAC PDU, following the payload, is CRC. This contains a 32-bit field used as a Cyclic Redundancy Check.

Error! Reference source not found. shows the sample payload of Probe Request and Probe Response.

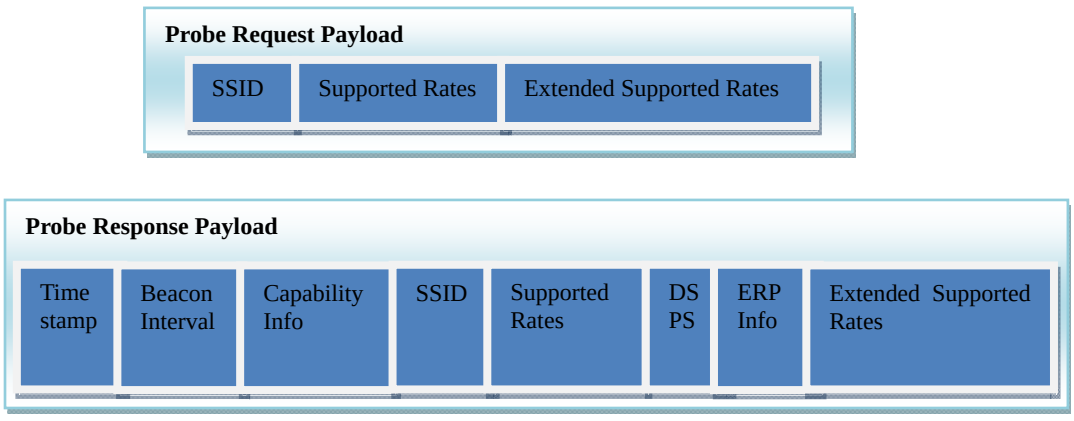

Figure 2-3: Payloads of Probe Request and Response [19]

A supplicant connecting to an AP needs to perform the following procedures:

1. Scanning

   There are two types of scanning: active and passive. In active scanning, the STA sends a Probe Reqest, and AP replies with Probe Resp.onse. In passive

scanning, the STA simply listens to beacon frames.

2. Association

   Association only occurs in infrastructure mode and is logically equivalent to connecting to a wired network.

3. Authentication

   There are three types of authentication methods: open, shared-key, and IEEE 802.1X. In the **open** authentication method the STA and AP exchange an authentication frame. The **shared-key** authentication method is based on WEP. The STA sends an authentication frame, then the AP replies with a challenge in clear text, to which the STA replies with an encrypted challenge. The AP decrypts this challenge text and matches it with the original clear text. If they match, then the AP will send an authentication frame with a status code of success. The third authentication method is **802.1X**, which is used in Wi-Fi Protected Access (WPA). Figure 2-4 shows the handshaking progress for WLAN authentication.

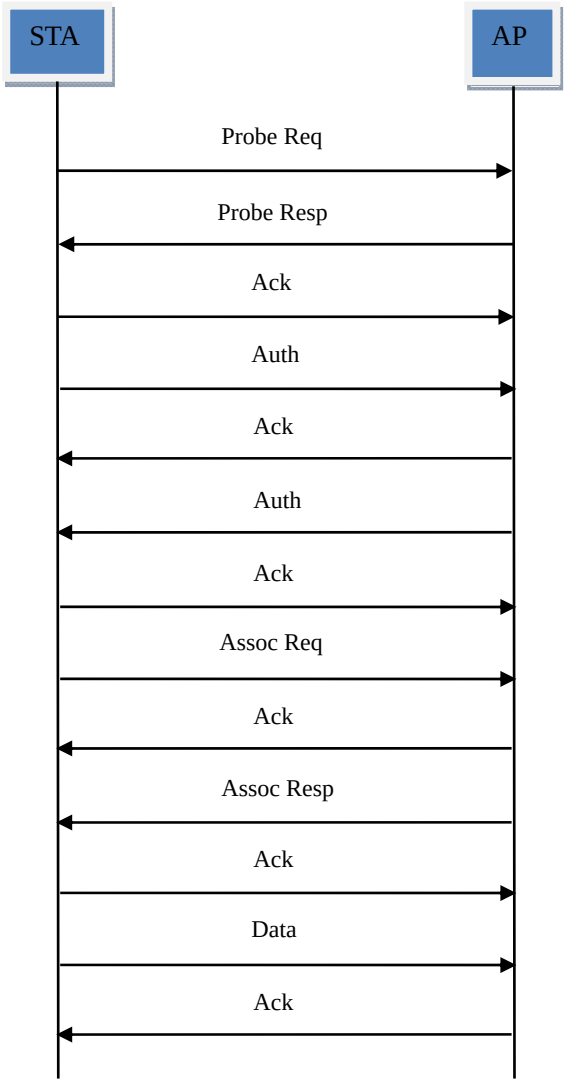

Figure 2-4. IEEE 802.11 handshake between AP and STA [19]

2.1.2 IEEE 802.11i

The initial IEEE 802.11 standard defined WEP to protect wireless networks. WEP uses RC4 with 40-bit keys, a 24-bit initialization vector (IV), and CRC32 to protect against packet forgery [19]. All these choices proved to be insufficient: the key space is too small to protect against attacks, RC4 key scheduling is insufficient, the IV space is too small and IV reuse makes attacks easier, there is no replay protection, and non-keyed authentication does not protect against bit flipping packet data. [23, 24, 25]

Because the security framework of IEEE 802.11 proved to be insecure, Task group I (Security) of the IEEE 802.11 working group [21] worked to address the flaws. The result was the IEEE 802.11i amendment to the IEEE 802.11 standard. This amendment was approved in June 2004 and published in July 2004.

Wi-Fi Alliance [22] adopted the 3.0 draft version of IEEE 802.11i in order to quickly establish a subset of the proposed security improvements. This subset is called Wi-Fi Protected Access (WPA). WPA capability is a mandatorily requirement for the interoperability testing and certification done by the Wi-Fi Alliance. WPA uses the Temporal Key Integrity Protocol (TKIP) to replace WEP. TKIP was selected as a compromise between strong security and the requirement by many vendors to be usable on existing hardware [19]. WPA uses RC4 for encryption, but with per-packet RC4 keys. Moreover, it adds replay protection and keyed packet authentication mechanism, based upon a message integrity check.

In WPA keys can be managed in two different manners: using pre-shared keys (called ‘WPA-Personal’) or by using an external authentication server (e.g., RADIUS) and the Extensible Authentication Protocol (EAP). The latter method, called WPA-Enterprise, is used by IEEE 802.1X. The purpose of both methods are the same: to generate a master session key for the AP and supplicant.

WPA uses a 4-Way Handshake and a Group Key Handshake to generate and exchange encryption keys between the authenticator and supplicant, for unicast and multicast traffic respectively. Both handshakes verify that both the authenticator and the supplicant know the master session key. These handshakes are independent of the selected key management mechanism, thus they is only one method for generating master session key changes [19].

After the final version of IEEE 802.11i was adopted, the Wi-Fi Alliance introduced a new version of WPA called WPA2. 802.1X serves primarily in this new standard. The 802.1X-2001[26] is called "Port-based network access control protocol”, because it makes use of point-to-point connection characteristics of IEEE 802 LAN, in which context the supplicant has a single point of attachment to the LAN infrastructure. By enforcing authenticating and authorizing over that port, it can prevent access from illegitimate users.

2.1.3 EAP

The Extensible Authentication Protocol (EAP)[53] is briefly reviewed before introducing 802.1X details in the next subsection.

The point-to-point protocol (PPP) was widely used for dial-up Internet access. PPP performs authentication at Layer 2 before establishing any network layer. PPP’s authentication methods, such as PAP and CHAP, had many limitations in terms of flexibility and security. Because most corporate networks want better access security

than offered by a simple username and password scheme, EAP was designed. EAP provides a generalized framework for various authentication methods by establishing a tunnel between the user and the authentication server. In the case of PPP, EAP operates inside PPP's authentication protocol. EAP was introducted to avoid proprietary authentication systems and enables authentication schemes ranging from passwords to challenge-response tokens and public key infrastructure certificates. Some popular EAP authentication mechanisms are listed in Table 2-6.

Table 2-6. EAP Types (adapted from [27])

	Server Authentication	**Supplicant Authentication**	**Dynamic key delivery**	**Risks**
EAP-MD5	None	Password hash	No	Man-in-the-middle attacks (MITM), session hijacking
LEAP	Password hash	Password hash	No	Identity exposed, directory attack
EAP-TLS	Public key (Certificate)	Public Key (either a certificate or a smart card)	Yes	Identity exposed
EAP-TTLS	Public key (Certificate)	CHAP, PAP, MS-CHAP (v2), EAP	Yes	MITM
PEAP	Public key (Certificate)	Any EAP	Yes	MITM; the identity is hidden in pahse2, but potentially exposed in phase 1

EAP is a very simple protocol with two message frames (Request or Response), four message types (request, response, success, and failure), and an extensible choreography. Figure 2-5 shows the EAP frame format. The Code field is one byte long and encodes the message type: (1) Request, (2) Response, (3) Success, and (4) Failure. The one byte identifier field is used to match requires and responses. The 16 bit length field indicates the total length of the EAP packet, and the data portion is a function of the method.

Byte offset 0	1	2	4
Code	Identifier	Length	Data

Figure 2-5. EAP Frame Format

EAP defines an EAP configuration negotiation packet. In this packet the data field contains a one byte Type field with the value 3, a Length field with a value of 4, and a two byte Authentication Protocol field with the value 0xC227[28].

EAP request and response packets (with Code values of 1 and 2 respectively), have a data field that contains a one byte Type field and data. The length of the data depends upon the value in the Type field. The EAP packet type values are assigned by the Internet Assigned Numbers Authority (IANA). Success and failure packets are a fixed 4 bytes long (hence their length field contains the value 4).

Figure 2-6 shows the EAP message exchange, which is similar to the IEEE 802.1X message exchange due to the fact that EAP is the basis for the IEEE 802.1X protocol. EAP starts after the supplicant has data and link layer connectivity (Step 0 in the figure), The link layer association process is covered in Section 2.2 Roaming. It is not specified who should start EAP first. Either the authenticator immediately detects the newly associated supplicant and sends out EAP-Request, or the client sends an EAPOL-Start message to the AP at first. In a word, EAPOL-Start is optional (as shown in the figure). It is important to note that there is no “EAP-start” packet in EAP. 802.1X does have an EAPOL-Start pakcet which is sent from the supplicant while RADIUS also has an EAP-Start message which is sent from the RADIUS client [4]. One thing we need to do is to take EAP messages out of the 802.11 frames and repackage them into Radius frames, vice versa.

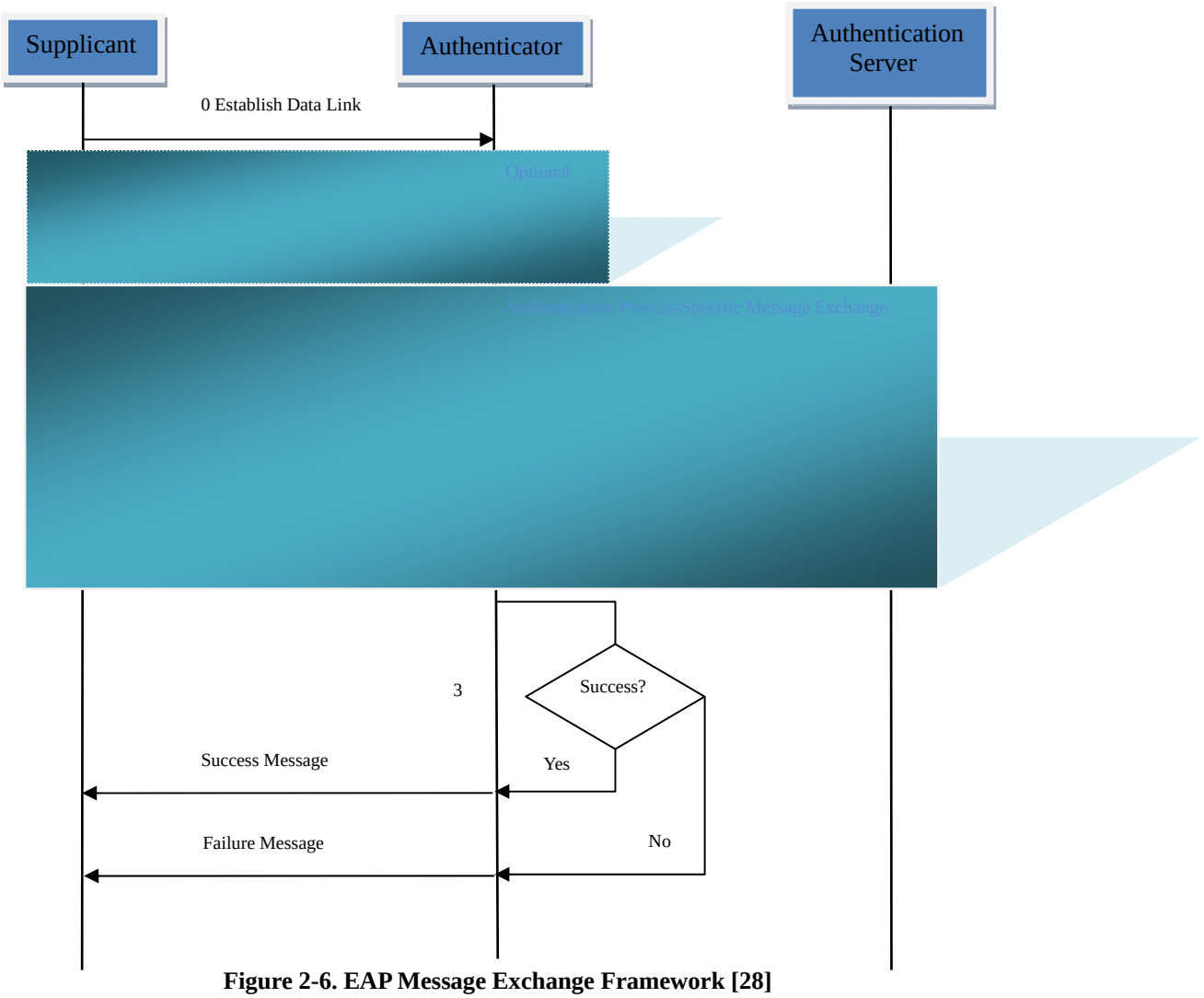

Figure 2-6. EAP Message Exchange Framework [28]

2.1.4 IEEE 802.1X Authentication on Wireless LANs

As mentioned in the last section, EAP contains an extensible choreography, which facilitates further RFCs defining EAP over various authentication processes, such as EAP-over-LDAP, EAP-SIM, EAP-over-GPRS, and EAP-over-802 (also know as EAP over LAN). Note that EAP-over-802 is the IEEE 802.1X specification. IEEE 802.1X describes port-based access models, while EAP adds the authentication mechanisms [28].

IEEE 802.1X defines a context (in terms of a port and supplicant), state machines, and the EAP over LAN (EAPOL) protocol. Details of them are given below. Actually, IEEE 802.1X simply passes EAP over a wired or wireless LAN and defines *an association between a station and an access point*. The association acts as a logical port for the purpose of interpreting the IEEE 802.1X standard. WLAN cards (or their software drivers) are supposed to support the IEEE 802.1X state engine, which requires that the IEEE 802.11 association must complete before the IEEE 802.1X negotiation begins. Since the IEEE 802.1X state machine requires an active link, successful exchange of Association Request and Association Response frames is reported to the IEEE 802.1X state machine as the link layer becomes active. The reason we call IEEE 802.1X a *binary* authentication is because the AP must drop all

non-802.1X traffic to and from the port (STA) prior to its successful IEEE 802.1X authentication. Once the authentication succeeds, the AP allows traffic to flow normally. Figure 2-7 shows the elements in IEEE 802.1X, specifically the STA, AP, AAA server (in this case a RADIUS server), port; and the protocols EAP, EAPOL, and RADIUS. **Error! Reference source not found.** shows 802.1X message exchanging process.

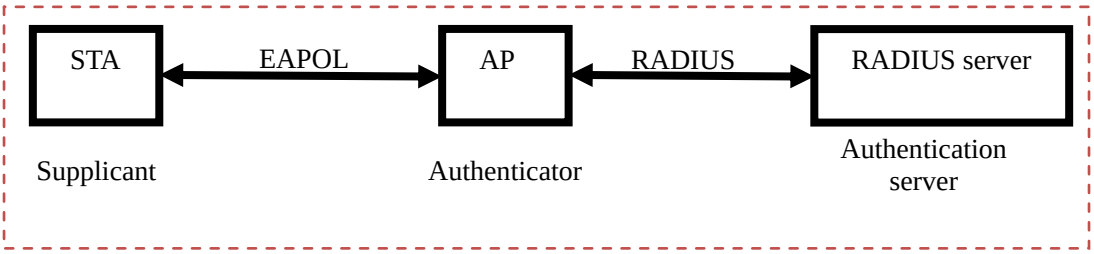

Figure 2-7. Elements in 802.1X (Adapted from [19])

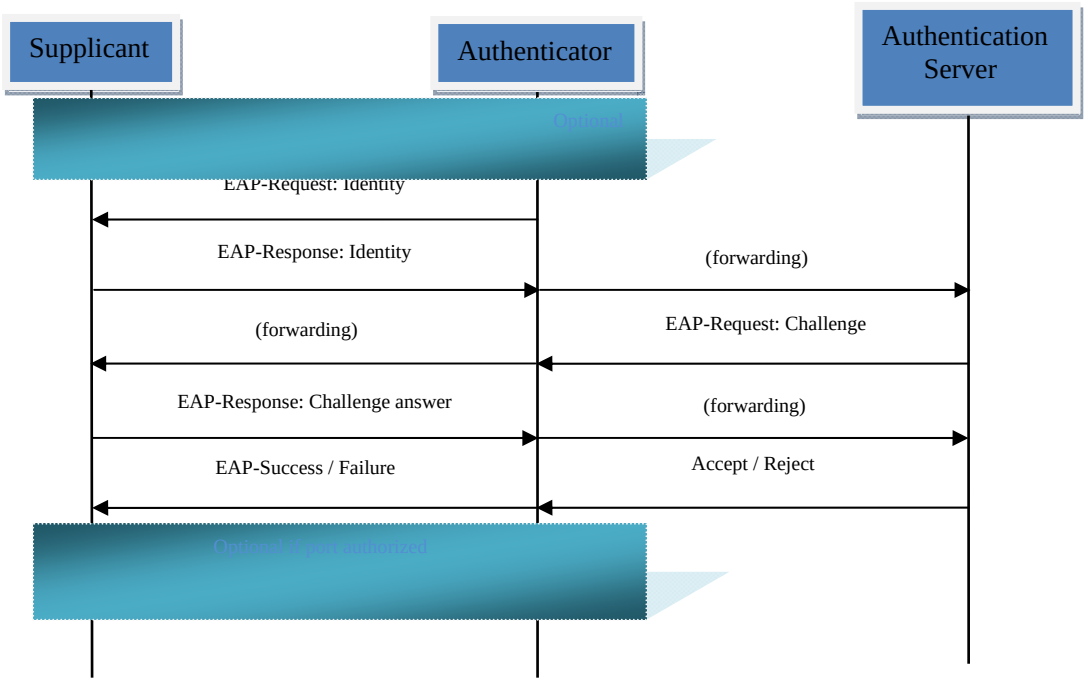

Figure 2-8. IEEE 802.1X Message Exchange Framework [37]

A supplicant (running on a STA), who wishes to access the WLAN's service, is responsible for replying to any authentication requests from the authenticator in order to establish its identity.

A port is a physical attachment point where the supplicant connects to the LAN. In the case of a wired network this is frequently a multiport switch or in the case of a wireless LAN it is an AP. In an IEEE 802.11 WLAN, the authenticator manages two distinct logical ports connected to the wireless interface: one is a "controlled port" and the other is an "uncontrolled port". When a frame is received by the wireless interface the bridging mechanism inside the AP will either forward the frame to the controlled port or the uncontrolled port. This forwarding is controlled by the authenticator. When a new STA first appears its frames are automatically forward to a controlled port – which delivers the frames to the authenticator. This traffic will cause the authenticator to challenge the supplicant and forward authentication messages from the supplicant

to the authentication server. As mentioned above, the authenticator's behavior is *independent* of the authentication method. The authenticator needs little memory and processing power, as most of the processing is done at the supplicant and authentication server.

IEEE 802.1X uses EAP to carry authentication messages between the supplicant and the authentication server. However, EAP was primarily developed for dial-up connections, thus there is no link layer protocol to carry EAP in an IEEE 802 LAN. That is why EAPOL was defined. In fact, EAPOL was originally defined for IEEE 802.3 / Ethernet and Token Ring / FDDI links. Since IEEE 802.11 WLAN has the same basic frame format as IEEE 802.3, EAPOL encapsulation can be handled directly by a LAN MAC service. Figure 2-9 shows the EAPOL MAC Protocol Data Unit (MPDU) for Ethernet.

Byte 0	1	2	3	4	5	6	7	8	9	
EAPOL Packet										
Ethernet Type		Protocol version	EAPOL Code	Packet Body Length		EAP Code	ID	Length (Total length of packet)		Data
2 bytes		1 byte	1 byte	2 bytes		1 byte	1 byte	2 bytes		
0x88 ..0x8E		2				EAP packet				

Figure 2-9. EAP in EAPOL MPDU for Ethernet (Adapted from [28])

As stated in section 2.1.3 on page 14, the EAP Code is one byte long and encodes the message type: (1) Request, (2) Response, (3) Success, and (4) Failure. The EAPOL Code values are: (1) EAP-Packet, (2) EAPOL-Start, (3) EAPOL-Logoff, (4) EAPOL-Key, (5) EAPOL-Encapsulated-ASF-Alert, (6..255) reserved. The EAPOL-Key message can be used to distribute or obtain global key information to / from attached stations, following successful authentication [38].

2.1.5 RADIUS

A Remote Authentication Dial In User Service (RADIUS) server stores information about subscribers (the authorized users of a service) in a database, authenticates them, and provides optional services, such as dynamic virtual LAN (VLAN) assignment and accounting. In our case, the AP acts as RADIUS client and it contacts the authentication server to learn if it should provide services to a supplicant.

The RADIUS protocol can be used to provide authentication, authorization, and accounting (AAA). These services give network administrators an easy way to (1) identify (authenticate) remote users and control which users can access the network; (2) define what each user can do by controlling access to network resources (authorization); and (3) keep track of what resources each user consumes in order to bill them for services (accounting) [39].

RADIUS operates at the application layer in the TCP/IP protocol suite. The RADIUS protocol defines how to exchange information between a RADIUS client and a RADIUS server [16]. RADIUS uses UDP to transport its messages, using UDP port 1812 for RADIUS authentication messages and port 1813 for RADIUS accounting messages. Figure 2-10 shows the RADIUS message format.

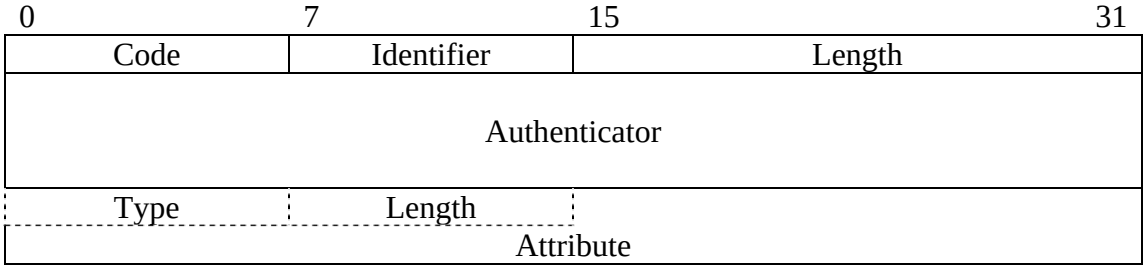

Figure 2-10. Radius Frame Format (Adapted from [9])

The Code field is 8 bits long. This field contains one of the following codes: (1) Access-Request; (2) Access-Access; (3) Access-Reject; (4) Accounting-Request; (5) Accounting-Response; (11) Access-Challenge; (12) Status-Server; (13) Status-Client; and (255) Reserved.

The Identifier field is 1 byte long. The Identifier value is used to match a request with its corresponding response. The value in a response is equal to the value in request. The identifier value is unchanged in the case of a retransmission [41].

The Length field is 2 bytes long and indicates the entire packet length.

The Authenticator field is sixteen bytes in length and contains the information that the RADIUS client and server use to authenticate each other. There are two kinds of authenticators: Request and Response [41]. For a Request the Authenticator value is randomly generated. A Reply is a MD5 digest of the reply message appended with the secret. For details of the RADIUS protocol see [54].

One or more RADIUS attributes are contained in the Attributes section, which carry specific authentication, authorization, and configuration details. Attribute Type denotes the type of the attribute. The attribute's name is not passed in the packet – just a number. Attribute Length indicates the length of the attribute field, which must be three or greater. Attribute Value contains the value of the attribute itself. This field is required for each attribute presented, even if the value itself is null [43, section 2.5]. Figure 2-11 shows the standard attribute-value pair (AVP) pattern. Table 2-7 shows the RADIUS Attribute Types.

Type	Length	Value
1..255	> 3	

Figure 2-11. RADIUS Attribute value pair format (Adapted from section 2.5 of [43])

Table 2-7. RADIUS attributes and their corresponding type number [3]

Type	Attribute	Type	Attribute
1	User-Name	2	User-Password
3	CHAP-Password	4	NAS-IP-Address
5	NAS-Port	6	Service-Type
7	Framed-Protocol	8	Framed-IP-Address
9	Framed-IP-Netmask	10	Framed-Routing
11	Filter-ID	12	Framed-MTU
13	Framed-Compression	14	Login-IP-Host
15	Login-Service	16	Login-TCP-Port
17	(unassigned)	18	Reply-Message
19	Callback-Number	20	Callback-ID
21	(unassigned)	22	Framed-Route
23	Framed-IPX-Network	24	State
25	Class	26	Vendor-Specific
27	Session-Timeout	28	Idle-Timeout
29	Termination-Action	30	Called-Station-ID
31	Calling-Station-ID	32	NAS-Identifier
33	Proxy-State	34	Login-LAT-Service
35	Login-LAT-Node	36	Login-LAT-Group
37	Framed-Apple Talk-Link	38	Framed-Apple Talk-Network
39	Framed-Apple Talk-Zone	40..59	(reserved for accounting)
60	CHAP-Challenge	61	NAS-Port-Type
62	Port-Limit	63	Login-LAT-Port

2.2 Roaming

Roaming extends connectivity for a subscriber to a network that is different from the home network [17]. The term originated in GSM, but here we will only be concerned roaming in the contents of a WLAN handover between two different authentication domains. There are two requirements for roaming, one is successful authentication and authorization of a subscriber (and in the background accounting for this visiting subscriber's resource usage so that their home network operator can be billed) and the other to minimize the period of time when the user has no connectivity due to the handover and AAA delays.

Since the original design of IEEE 802.11 did not consider mobility and security between networks, protocols such as IEEE 802.1X, 802.11i, 802.11f, and 802.11e were proposed to meet the increasing demand for WLAN mobility and security. Some manufacturers and engineers also introduce their own roaming solutions, however they suffered from various drawbacks such as being incompatible with existing devices, service degradation due to long authentication times, complicated certificate management, or highly complex platforms with little flexibility.

For example, one proposal is to implement Wi-Fi roaming using VPNs with client certificates. This assumes that all the APs are in the same VLAN and are connected to the outside world through an IPSec gateway. This gateway prevents any communication between supplicants and the internet until the supplicant has established an authenticated VPN. However, this proposal suffers from unreliable certificate management and long authentication times. What is more the requirement that "all APs are in the same VLAN" limits the scalabity of roaming.

A solution proposed by Deutsche Telecom provides an integrated Wi-Fi roaming platform (**Error! Reference source not found.**). Each of the wireless inteternet service providers (WISP) allow inbound roaming by opening their hotspots up to subscribers from the other operators and allowing easy access (for example with a central login page). Each of the internet service providers (ISPs) permit outbound roaming to other service providers. This system uses a central hotspot database with a single AAA Hub connected to a WiFi Roaming Platform. RADIUS mediation eliminates the need to configure a RADIUS interface per roaming partner [44]. This roaming solution acts as a pure wholesale positioning partner [44]. It solves authentication and accounting problems when roaming between different hotspots or service providers. Compared with the VLAN proposal above, this is more scalable (as long as you sign a contract to join their platform). However, their proposal does not even mention seamless roaming. It simply establishes a monopoly layer in the middle, which costs a lot but is neither flexible nor necessary for small to medium size operators (who may only have a limited set of converage areas).

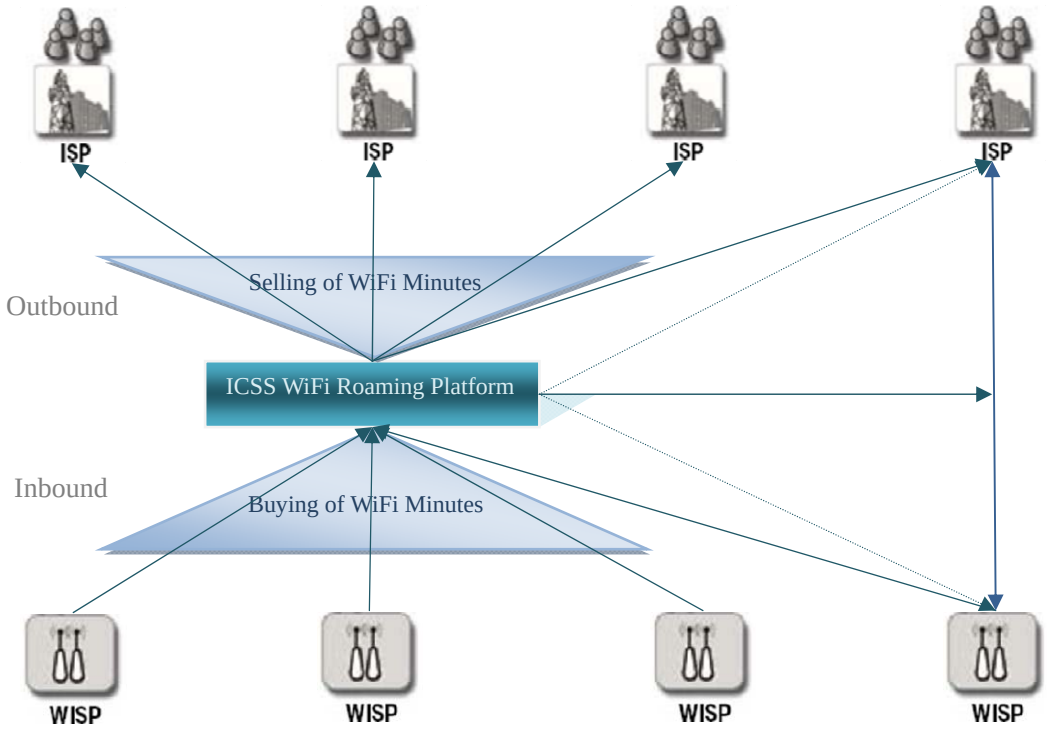

Figure 2-12. WiFi Roaming Platform [44]

In order to solve these problems, it is critical to make a clear track of the roaming process and analyze how much latency each step contributes. A normal handover process based on IEEE 802.11i includes five steps:

1. Discover the targeted AP.
2. Associate with this AP.
3. AAA conversion: Use IEEE 802.1X to negotiate an authentication scheme between supplicant and authentication server through the EAP, then carry out the authentication.
4. Link layer security: Provide link layer encryption to protect the traffic between the client device and the AP. Since all important applications provide end-to-end encryption, we omit consideration of link level encryption in this thesis.
5. ISP selection and QoS: If there are multiple ISPs, the AP has to map traffic to a specific ISP and potentially provide QoS guarantees. Details of this lie outside the scope of this thesis.

Handover latency primarily comes from steps 2 and 3. While this thesis focuses on minimizing or eliminating delay in step 3, it is import to pay attention to what is going on in steps 1 and 2. These two steps have been analyzed in detail in Jon-Olov Vatn's doctoral disseration [20]. He has shown that the delays due to step 1 and step 2 can be reduced to a total delay in the order of *milliseconds*, which is much shorter than the several *seconds* delay due to step 3. Because the delay due to the AAA conversion is so long it has a significant impact on many applications, therefore we need to markedly reduce this delay – hence reducing this delay is the focus of this thesis project.

2.3 hostapd

As mentioned in the beginning, we will leverage existing access points by upgrading their software. HostAP [49] is an open source Linux driver for WLAN, which enables a computer running Linux to act as an access point. It supports normal station operations in BSS and IBSS. It supports the IEEE 802.11 functions: authentication and deauthentication, association and disassociation, reassociation, data transmission, and power saving (PS) mode signaling and frame buffering for PS stations. The driver also implements the basic functionality needed to initialize and configure Prism2-based cards, to send and receive frames, and to gather statistics. The time critical tasks such as beacon sending and frame acknowledgments are taken care of by the firmware of Prism2 chipset [49].

On top of this driver is a user space daemon – hostapd. This daemon implements IEEE 802.11 access point management (authentication / association), an IEEE 802.1X/WPA/WPA2 Authenticator, integrated EAP server, RADIUS client, and RADIUS authentication server [50]. **Error! Reference source not found.** shows the hostapd modules. Using a combination of Host AP driver and hostapd daemon we can support the following features: IEEE 802.1X and dynamic WEP rekeying, RADIUS Accounting, RADIUS-based ACL for IEEE 802.11 authentication, minimal IAPP (IEEE 802.11f), WPA, and IEEE 802.11i/RSN/WPA2 [49].

With regard to the previous section the firmware and Host AP implement step 1 and step 2; while hostapd implements the functions that an AP needs for step 3: standard IEEE 802.1X framework, EAPOL support, multiple user authentication, and data privacy with strong encryption. As we focus on step 3, we will not examine the firmware or how Host AP operates as a WLAN driver. Additionally, we do not care about the exact authentication method used in EAP. As indicated earlier this is primarily a matter between the supplicant and authentication server. Considering the basic AAA conversion, there are three primary elements: (1) a sender and receiver of EAPOL on AP’s wireless side; (2) a RADIUS client on AP’s LAN side; and (3) a set of state machines cooperating with each other to fulfill the logic of IEEE 802.1X. The solution proposed in Chapter 2will implement these three elements based upon the hostapd module.

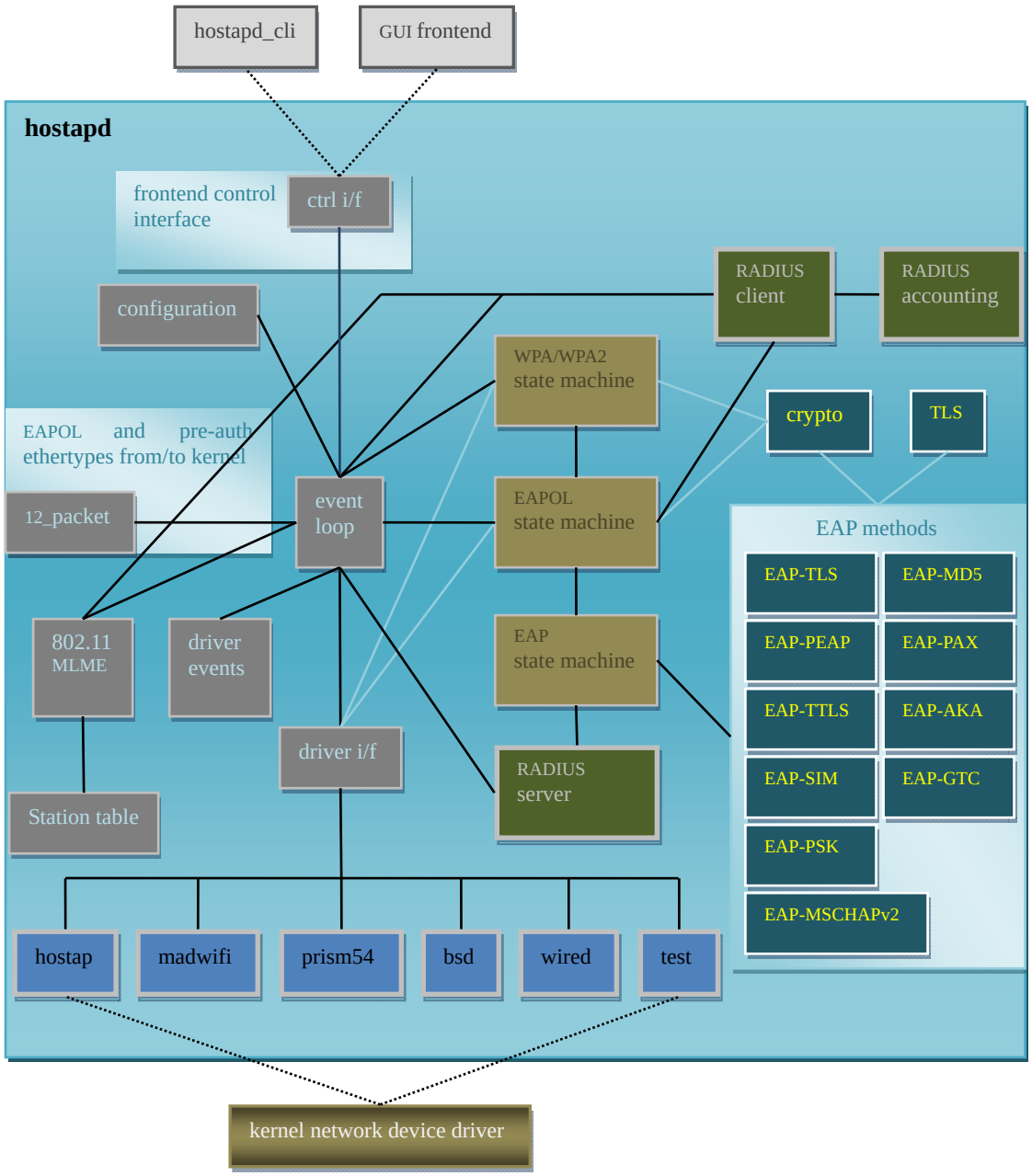

Figure 2-13. hostapd Modules [50]

2.4 Netfilter

The first problem in our design is how to acquire network packets without duplicating the logic in the Linux kernel. Fortunately the netfilter [15] subsytem in the kernel offers a means of getting packets, while minimizing the code that we need to write.

2.4.1 Netfilter Framework

Netfilter is a structured subsystem in Linux kernel with the ability to add code that will be invoked when packets reach various stages of processing the the Linux kernel's networking code. This functionality has been utilized in the past to implment a wide variety of network services, such as packet filtering, network address

translation (NAT), and connection tracking. The netfilter framework is designed to be highly flexible and scalable, allowing new features to be either statically built-in or dynamically loaded (as loadable modules). Figure 2-14 shows where the netfilter resides in the Linux network layer.

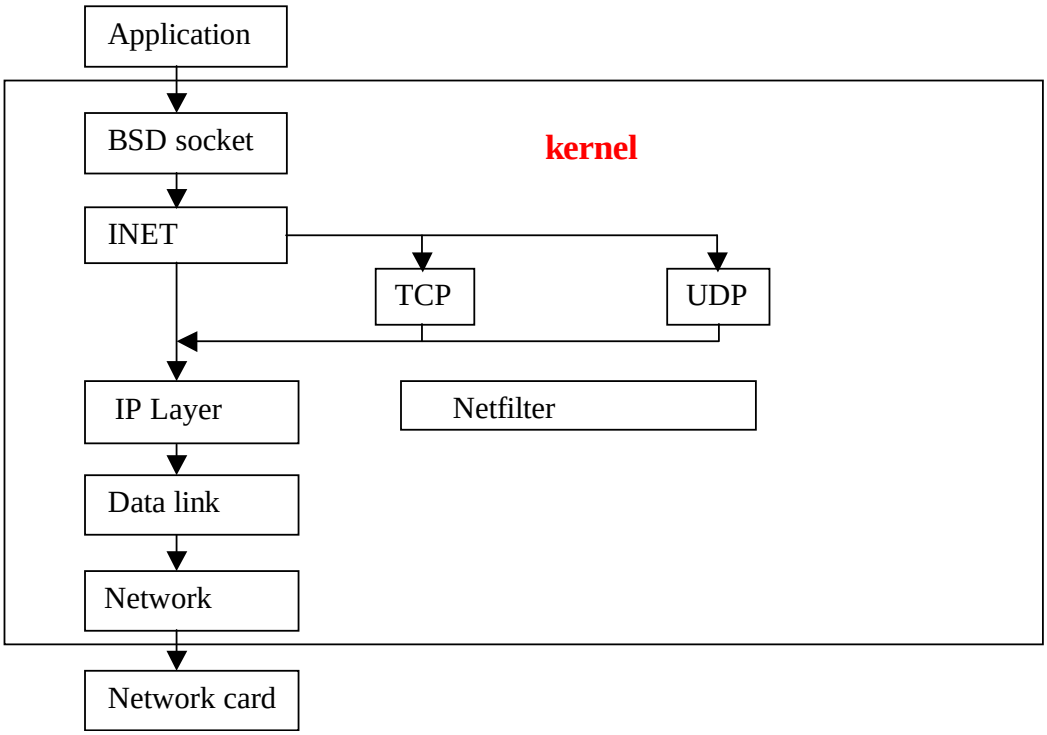

Figure 2-14. netfilter inside Linux kernel

Although netfilter operations are in the same layer and frequently interact with the Linux kernel networking code, netfilter function modules are clearly separated from the Linux kernel IP layer. The netfilter framework consists of three parts:

1. A suit of hooks for each network protocol (5 hooks for IPv4), which can be called when packets pass through them. Note that a hook is a function that enables other code to request that it be called when the hook is invoked.

2. In order to represent the above hooks, netfilter defines a two-dimensional list_head array:

   ```
   struct list_head nf_hooks [NPROTO] [NF_MAX_HOOKS]
   ```

   As there are thrity-two protocols supported by Linux, the value of NPROTO is 32 (defined in include/linux/socket.h). Each protocol has a maximum of NF_MAX_HOOKS hooks (currently 8), but only 5 of them are used in the IPv4 code. Each member of the array contains a link to the header of a hook for a specific protocol. Therefore, whenever a packet passes through the network stack, the kernel checks if there is a hook registered for this protocol in this position, if there is, then this element of the array contains the address of a function that will be invoked by the hook (as a call back) to handle this packet. This packet might be analyzed, modified, discarded, or even queued for further processing in user space.

3. The user space program processes the queued packets asynchronously. Thus the processing of these packets will only occur when the user space program is scheduled for execution by the scheduler. The user space program can inspect and modify the packet, and can also inject the packet back into the

kernel through **the same** hook. Thus it will appear that the processing takes place in the kernel – but without the code actually needing to be executed in either kernel mode or in kernel address space. This removes many of the restrictions that would occur if the code would be part of the kernel (such as the ability to do file I/O, invoke other programs, etc.)

Figure 2-15 displays the work flow of netfilter in IPv4 as well as the location of the five hooks. The names of these five hooks and their purpose are listed in **Table 2-8.**

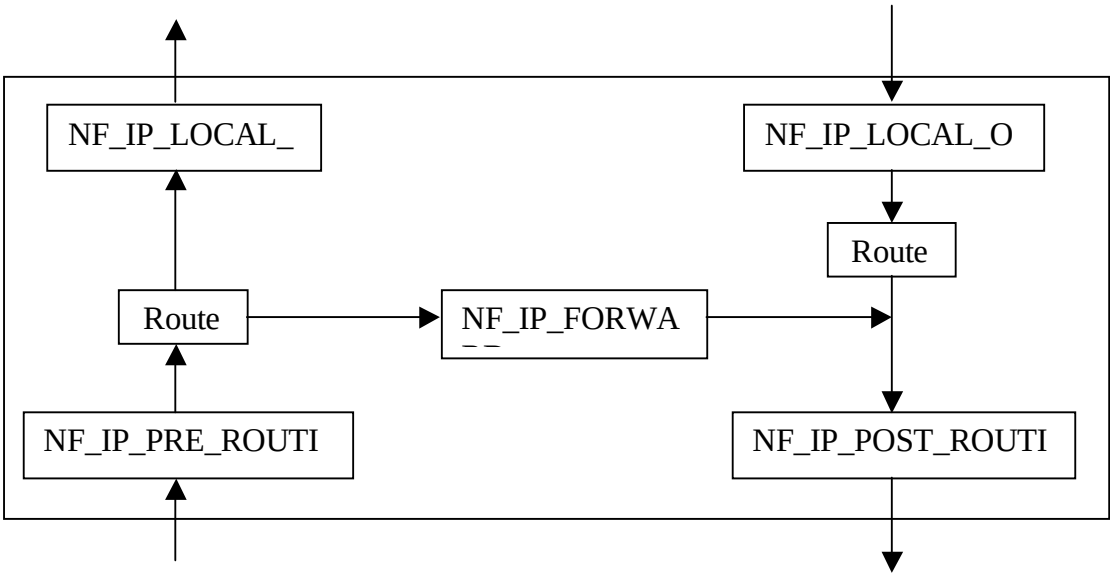

Figure 2-15. netfilter in IPv4

Table 2-8: IPv4 hooks

HOOK	Aimed Packet	Function
NF_IP_PRE_ROUTING	Before routing decisions, just into IP layer	Source address translation
NF_IP_LOCAL_IN	After routing decisions, destined for this host	Incoming packets filtering
NF_IP_FORWARD	After routing decisions, destined for another interface	transmitting packets filtering
NF_IP_LOCAL_OUT	Sent out by local processes	Destination address traslation
NF_IP_POST_ROUTING	All outbound packets	Outgoing packets filtering

The NF_IP_PRE_ROUTING hook is invoked when a sk_buff packet is passed to the IP stack successfully, that is, after sanity checks. It is called in ip_rcv() in net/ipv4/ip_input.c:

```
int ip_rcv (struct sk_buff *skb, struct net_device *dev, struct
packet_type *pt) {
   … …
return NF_HOOK (PF_INET, NF_IP_PRE_ROUTING,
                  skb,skb->dev, NULL, ip_rcv_finish);
}
```

Then the routing table would decide whether this packet is destined for this host. If so, before passing it to the upper layer protocols, it has to go through the NF_IP_LOCAL_IN hook, which is called in ip_local_deliver() in net/ipv4/ip_input.c:

```
int ip_local_deliver (struct sk_buff *skb) {
  … …
  return NF_HOOK (PF_INET, NF_IP_LOCAL_IN, skb,
                  skb->dev, NULL,
                  ip_local_deliver_finish);
}
```

Otherwise it would be handled by the NF_IP_FORWARD hook before forwarding. This is called in ip_forward() in net/ipv4/ip_input.c:

```
int ip_forward (struct sk_buff *skb) {
  … …
  return NF_HOOK (PF_INET, NF_IP_FORWARD, skb, skb->dev,
                 dev2, ip_forward_finish);
}
```

Packets sent by the local host need to pass the NF_IP_LOCAL_OUT HOOK before further routing decisions. This is called in ip_build_and_send_pkt() in net/ipv4/ip_output.c:

```
int ip_build_and_send_pkt (struct sk_buff *skb, struct sock
*sk, u32 saddr, u32 daddr,
struct ip_options *opt) {
  … …
  return NF_HOOK (PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
                  rt->u.dst.dev,
                  output_maybe_reroute);
}
```

The last hook all outbound packets will encounter is the NF_IP_POST_ROUTING hook. This is called in ip_finish_output() in net/ipv4/ip_output.c:

```
__inline__int ip_finish_output (struct sk_buff *skb) {
  … …
return NF_HOOK (PF_INET, NF_IP_POST_ROUTING, skb,
                  NULL, dev, ip_finish_output2);
}
```

All of these hooks already exist in the kernel and are ready for use, as long as at least one function is registered as a call back for each hook. Each registered hook returns one of the values shown in Table 2-9 as a result.

Table 2-9: netfilter return codes [12]

NF_ACCEPT	Transmit the packet as usual
NF_DROP	Discard the packet
NF_STOLEN	Taken over, no more transmission
NF_QUEUE	Put the packet into queue, generally for user space
NF_REPEAT	Call this hook function again

For questions about sk_buff, please refer to sk_buff analysis [8]. For questions about socket programming, please refer to manual of PF_PACKET [6] and manual of AF_PACKET [7].

2.4.2 Hook Operation

One of our key requirements is to do filtering, including redirecting registration packets, block unauthorized user packets, transmit authorized user packets, and so forth. Thus we need a more powerful filter, which is client specific, rule specific, and protocol specific. In order to be client specific, we need access to a AAA server and a list of local hosts; to be rule specific, we need a more functional rule table than iptable; and to be protocol specific, we need to register our own operations for these hooks. Before we can register an operation, we need to define our own nf_hook_ops and then call nf_register_hook (). This is done as follows:

```
/* include/linux/netfilter.h */
struct nf_hook_ops {
  struct list_head list;         // link list header
/* user fills in from here down. */
nf_hookfn *hook;          // user defined handling function

int pf;                  // protocol
int hooknum;             // hook number
/* hooks are ordered in ascending priority. */
int priority;
};
```

As mentioned above that netfilter defines a two-dimensional list_head array to represent the hooks:

struct list_head nf_hooks [NPROTO] [NF_MAX_HOOKS]

For an operation to attach itself into specific hook, it must know which hook it wishes to belong to. This is done by referring to the hook's properties: list_head list, int pf, and int hooknum. There can be many operations in the form of nf_hook_ops inserted in the link. The individual hooks are ordered in ascending priority, where the smaller number, the higher priority:

```
NF_IP_PRI_FIRST = INT_MIN,
NF_IP_PRI_CONNTRACK = -200,
NF_IP_PRI_MANGLE = -150,
NF_IP_PRI_NAT_DST = -100,
NF_IP_PRI_FILTER = 0,
NF_IP_PRI_NAT_SRC = 100,
NF_IP_PRI_LAST = INT_MAX,
```

These priorities show that netfilter first deals with connection track (CONNTRACK), secondly mangle, thirdly destination address translation (NAT_DST), fourthly filter, next source address translation (NAT_SRC), and any remaining hooks. At each point, if the operation discards the packet, then the packet is immediately discarded and does not flow to the following operations. Otherwise the packet continues to the next operation.

The address of nf_hook_ops serves as a parameter of nf_register_hook(), which returns 0. The following code is from "Hacking the Linux Kernel Network Stack"[13], which does a simple hook registration that will throw all packets away.

```
/* Sample code to install a Netfilter hook function that will
drop all incoming packets. */

#define __KERNEL__
#define MODULE

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

/* This is the structure we shall use to register our function
*/
static struct nf_hook_ops nfho;

/* This is the hook function itself */
unsigned int hook_func(unsigned int hooknum,
                   struct sk_buff **skb,
                   const struct net_device *in,
                   const struct net_device *out,
                   int (*okfn)(struct sk_buff *))
{
   return NF_DROP;          /* Drop ALL packets */
}

/* Initialisation routine */
int init_module()
{
   /* Fill in our hook structure */
   nfho.hook = hook_func;        /* Handler function */
   nfho.hooknum  = NF_IP_PRE_ROUTING; /* First hook for IPv4
*/
   nfho.pf      = PF_INET;
   nfho.priority = NF_IP_PRI_FIRST;   /* Make our function
first */

   nf_register_hook(&nfho);

   return 0;
}

/* Cleanup routine */
void cleanup_module()
{
   nf_unregister_hook(&nfho);
}
```

As the above example shows, to define our own filter module, we only need to define its initialization routine, which initiates our own nf_hook_ops by specifying its properties (hook number, protocol number, and priority), as well as defining its handler function. To unregister a filter module is also simple, simply call nf_unregister_hook() with the address of the same nf_hook_ops that was used for registering the hook.

Now that the connection between an operation and its hook is clear, let us look into how to pass a packet from the hook to its handler function. We will use the hook NF_IP_PRE_ROUTING as example. The NF_IP_PRE_ROUTING hook is called in ip_rcv() in net/ipv4/ip_input.c:

```
int ip_rcv (struct sk_buff *skb, struct net_device *dev, struct
packet_type *pt) {
  … …
  return NF_HOOK (PF_INET, NF_IP_PRE_ROUTING, skb, skb->dev,
NULL, ip_rcv_finish);
}
```

Here a socket buffer (called skb) is passed to the hook at the head of linked list of NF_IP_PRE_ROUTING hooks and it will be passed all the hooks on this list it is discarded or finally accepted (this means that it passes to the next stage of processing as shown in Figure 2-15). An operation, a nf_hook_ops, takes in the packet (by nf_register_hook) and passes it to its corresponding handler fuction nf_hookfn * (recall the structure of nf_hook_ops). The prototype for nf_hookfn is given in linux/netfilter.h as follows:

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
                      struct sk_buff **skb,
                      const struct net_device *in,
                      const struct net_device *out,
                      int (*okfn)(struct sk_buff *));
```

The first argument, hooknum identifies the specific hook. The pointer sk_buff ** points to the packet. This is a very complex structure, which will be used frequently in the following chapters. A packet typically includes a link layer header (Ethernet or RAW), network layer header (IPv4/6, IPX, RAW), and transport layer header (TCP, UDP, ICMP, SPX). These headers are organized into the corresponding unions: mac, nh, and h (for **MAC** layer header, **n**etwork **h**eader, and transport **h**eader). You must be very cautious when referring to header information. Consider a TCP packet as example, both the transport header **h**, and network header **nh** point to IP header structures. h->th is equal to nh->iph. Thus if you want to refer to a TCP packet's header, I suggest doing the following:

```
  …
  /* tcphdr is defined in linux/tcp.h */
  struct tcphdr *tcpheader;
  if (skb->nh.iph->protocol == IPPROTO_TCP)
     tcpheader = (struct tcphdr *) (skb->data +
(skb->nh.iph->ihl*4));
  …
```

The IP header length skb->nh.iph->ihl is 32 bits long, which is 4 chars in length. Thus, skb->data + (skb->nh.iph->ihl*4) would skip the complete IP header and would point to the TCP header. The following code explains the unions in sk_buff.

```
struct sk_buff {
...
/* Transport Layer header */
union {
struct tcphdr *th;
...
unsigned char *raw;
} h;

/* Network Layer header */
union {
struct iphdr *iph;
...
unsigned char *raw;
} nh;
..
}
```

This header structure contains packet header information, while the real data is inside skb->h.raw and skb->nh.raw, which are "unsigned char *" pointers. That is why it is necessary to explictly coerce the type into our desired type (struct tcphdr *).

The third argument is of type net_device *in. This is used to describe the packet's incoming interface and type net_device *out describes the packet's outgoing interface. Generally, in is only relevant for NF_IP_PRE_ROUTING and NF_IP_LOCAL_IN hooks; while out is only relevant for NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING. In the case of ip_rcv only one interface /the incoming interface) is passed to the hook function. For example, if we want to block packets coming from eth0, we do this filtering in NF_IP_PRE_ROUTING. If we want to block packets being send eth1, we do this filtering in NF_IP_POST_ROUTING, when the destined interface is known after routing decision is made. Within the hook we will look for packets destined to to the net_device *out corresponding to eth1. Note that there is no incoming device for local generated packets and there is no outgoing device for host destined packets. Please refer to chapters 4 and 5 of "Hacking the Linux Kernel Network Stack"[13] to see how the code exactly works.

2.4.3 Rules Table

In order to abstract those behaviors carried out by each hook, netfilter defines rule tables and each table is described by a structure ipt_table. The most familiar examples of table are `filter`, `nat`, and `mangle`, which are the default rule tables in Linux. Each table can be divided into several chains and each chain utilizes a specific hook. For example, table `filter` has three chains and they are for NF_IP_LOCAL_IN, NF_IP_LOCAL_OUT, and NF_IP_FORWARD. In order to go through the complete rule collection on a hook we call the function ipt_do_table(). The relationship between netfilter hooks and ipt_table may be confusing, thus the following section describes the ipt_table or IPTABLE in more detail.

2.5 IPTable

The rules that a Linux user can enter using the `iptable` command are connected to tables in netfilter. We will start our examination of the IPTable by asking three questions:

1. What is the relation between rules, tables, and hooks?
2. If we are already able to define filtering rules in nf_hook_ops, then why is it necessary to build a rule table system on top of these hooks?
3. Netfilter has its own hook functions that packets would go through. However, IPTABLE introduces its own chains, so how do packets actually get processed?

You can refer to Iptables Instruction [2] for common questions and answers.

2.5.1 The three Default IP Tables

For the ease of further reference, we put the definition and operations of each of these three tables in Table 2-10 and Table 2-11, respectively.

Table 2-10. IPTABLE Definitions

Table	Definition	Operation	Location
filter	ipt_table packet_filter	struct nf_hook_ops ipt_ops[]	netfilter/iptable_filter.c
nat	ipt_table nat_table	ip_nat_standalone.c	netfilter/ip_nat_rule.c
mangle	ipt_table packet_mangler	struct nf_hook_ops ipt_ops[]	netfilter/iptable_mangle.c

Table 2-11. IPTABLE Operations

table	Hook	Operation
filter	NF_IP_LOCAL_IN	ipt_hook() calls ipt_do_table() to connect to its INPUT chain
	NF_IP_LOCAL_FORWARD	ipt_hook() calls ipt_do_table() to connect to its FORWARD chain
	NF_IP_LOCAL_OUT	ipt_local_out_hook() calls ipt_do_table() to connect to its OUTPUT chain
nat	NF_IP_PRE_ROUTING	struct nf_hook_ops ip_nat_in_ops -> ip_nat_in() calls ip_nat_rule_find(); calls ipt_do_table() to connect to its PREROUTING chain
	NF_IP_POST_ROUTING	struct nf_hook_ops ip_nat_out_ops -> ip_nat_out() calls ip_nat_rule_find(); calls ipt_do_table() to connect to its POSTROUTING chain
	NF_IP_LOCAL_OUT	
mangle	NF_IP_PRE_ROUTING	ip_route_hook() calls ipt_do_table to coonect to its PREROUTING chain
	NF_IP_LOCAL_IN	ip_route_hook() calls ipt_do_table to coonect to its INPUT chain
	NF_IP_FORWARD	ip_route_hook() calls ipt_do_table to coonect to its FORWARD chain
	NF_IP_LOCAL_OUT	ip_route_hook() calls ipt_do_table to coonect to its OUTPUT chain
	NF_IP_POST_ROUTING	ip_route_hook() calls ipt_do_table to coonect to its POSTROUTING chain

A table is just a collection of rules that do similar jobs, such as filtering. Rules are actually stored in as a structure ipt_entry. In order to locate the relevant rule accurately and quickly, all the rules inside a table are put in an array ipt_entry[], whose length is dynamic. From a user's perspective, there is a structure ipt_match to describe the rule's matching conditions and if the rule is matched there is a structure ipt_target to invoke the functions necessary to carry out the target.

2.5.2 Data Structures

If you want to create a new table, you need to create a structure ipt_table which looks like this:

```
/* netfilter_ipv4/ip_tables.h */
struct ipt_table
{
  /* tables are put in a linked list, here your table is hung
*/
  struct list_head list;

  /* the name must be unique */
  char name [IPT_TABLE_MAXNAMELEN];

  /* prepare table, providing basic infor, replace the old one
by register_table */
struct ipt_replace *table;

/* valid hooks the table will hang on */
unsigned int valid_hooks;

/* lock for table operation */
rwlock_t lock;

/* indexed pointer for rules, initiated as NULL */
struct ipt_table_info *private;

/* self referenced, used for stat */
struct module *me;
}
```

The structure ipt_table_info is explained below:

```
/* include/linux/netfilter_ipv4/ip_tables.h */
struct ipt_table_info
{
/* table size */
unsigned int size;

/* Number of entries, an entry per rule */
unsigned int number;

/* Initial number of entries. Needed for module usage count
*/
unsigned int initial_entries;

/* offset of the first rule for each hook */
```

```
unsigned int hook_entry[NF_IP_NUMHOOKS];

/* offset of the last rule for each hook */
unsigned int underflow[NF_IP_NUMHOOKS];

/* ipt_entry tables: one per CPU */
char entries[0] ____cacheline_aligned;
};
```

The structure ipt_replace is explained as following:

```
/* include/linux/netfilter_ipv4/ip_tables.h */
struct ipt_replace
{
  /* table name */
  char name[IPT_TABLE_MAXNAMELEN];

 /* valid hook entry points : bitmask */
unsigned int valid_hooks;

/* number of entries */
unsigned int num_entries;

/* total size of new entries */
unsigned int size;

/* offset of the first rule for each hook */
unsigned int hook_entry[NF_IP_NUMHOOKS];

/* offset of the last rule for each hook */
unsigned int underflow[NF_IP_NUMHOOKS];

/* information about old entries */
/* number of counters, a counter per entry */
unsigned int num_counters;

/* the old entries' counters. */
struct ipt_counters *counters;

/* the table's entrance */
struct ipt_entry entries[0];
};
```

In the above above see see that the rules are stored in ipt_entry[ ]. All the rules belonging to a table are stored in an array. The method “chain” for each available hook is actually part of the array. These “chains” are ordered according to their hook number sequence. For example, for the table `filter`, the array starts with INPUT, then FORWARD, then OUTPUT, and finally error handling. The head and end rules of each chain, or hook, are marked by hook_entry[ ] and underflow[ ]. Now it is time when a picture is more worthy than a thousand words. The following code is from net/ipv4/netfilter/iptable_filter.c and shows the built-in filter table. Figure 2-16 illustrates the relationship between the filters and the chains.

```
static struct ipt_table packet_filter = {
{NULL,NULL},      // link list
“filter”,            // name
&initial_table.repl,    // initial table model
/*
 * this is very interesting, valid hooks are defined as
 *
(1<<NF_IP6_LOCAL_IN)|(1<<NF_IP6_FORWARD)|(1<<NF_IP6_LOCAL_
OUT)
*/
FILTER_VALID_HOOKS,
RW_LOCK_UNLOCKED,  // lock
NULL,                  // empty
THIS_MODULE
};
```

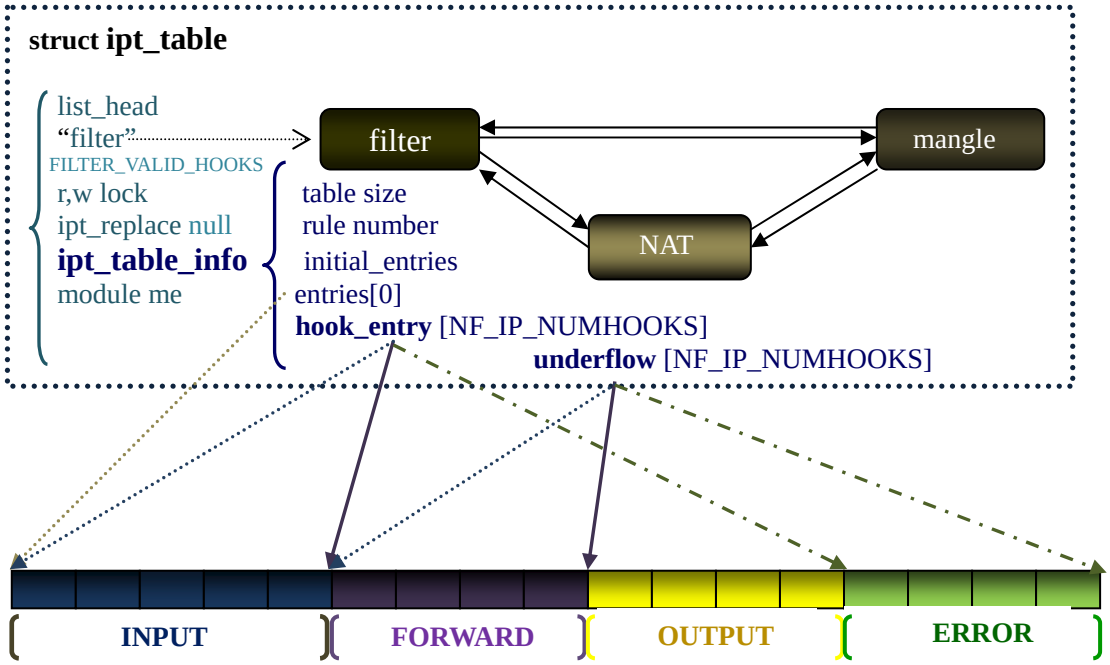

Figure 2-16. IPTable Structure

Entries in the ip_table give general information, while entries in the ipt_table_info concern the chains. For example, the struct ipt_replace contains parameters both from ip_table and ipt_table_info. These parameters will be transferred into members of ipt_table_info when you register a new table by calling the function int ipt_register_table (struct ipt_table *). You call void ipt_unregister_table (struct ipt_table *) to remove a table.

Each entry in ipt_entry represents a rule. This rule contains a matching IP header, one or more Match items, and one Target. Its size is not fixed since its Match items could vary. The ipt_entry struct is shown below:

/* include/linux/netfilter_ipv4 */

```
struct ipt_entry
{                    // IP header used for matching
  struct ipt_ip ip;
                     // mark of concern on packet
unsigned int nfcache;
                  // target comes after match, match is in the
end of ipt_entry
u_int16_t target_offset;
                   // offset of the next rule, or size of this
rule, sizeof(ipt_entry) +
                   // sizeof(ipt_match) * n + sizeof
(ipt_target), n>=0
u_int16_t next_offset;
                   // marking the hook it belongs to
unsigned int comefrom;
                  // accumulated number of packets and data
struct ipt_couners counters;
                    // position of target or the first match
unsigned char elems[0];
}
```

The array ipt_entry[ ] is located right after the ipt_table->private->entries[0]. Each member of the array contains its own match(es) and target.

2.5.3 Work Flow

Now that we are familiar with the netfilter structure and iptables, let us take a look at their work flow in order to understand their relationship. First we take a systematic view of packet processing in Linux network stack.

shows the flowchart for sending packets and **Error! Reference source not found.** & Figure 2-19 are for receiving packets.

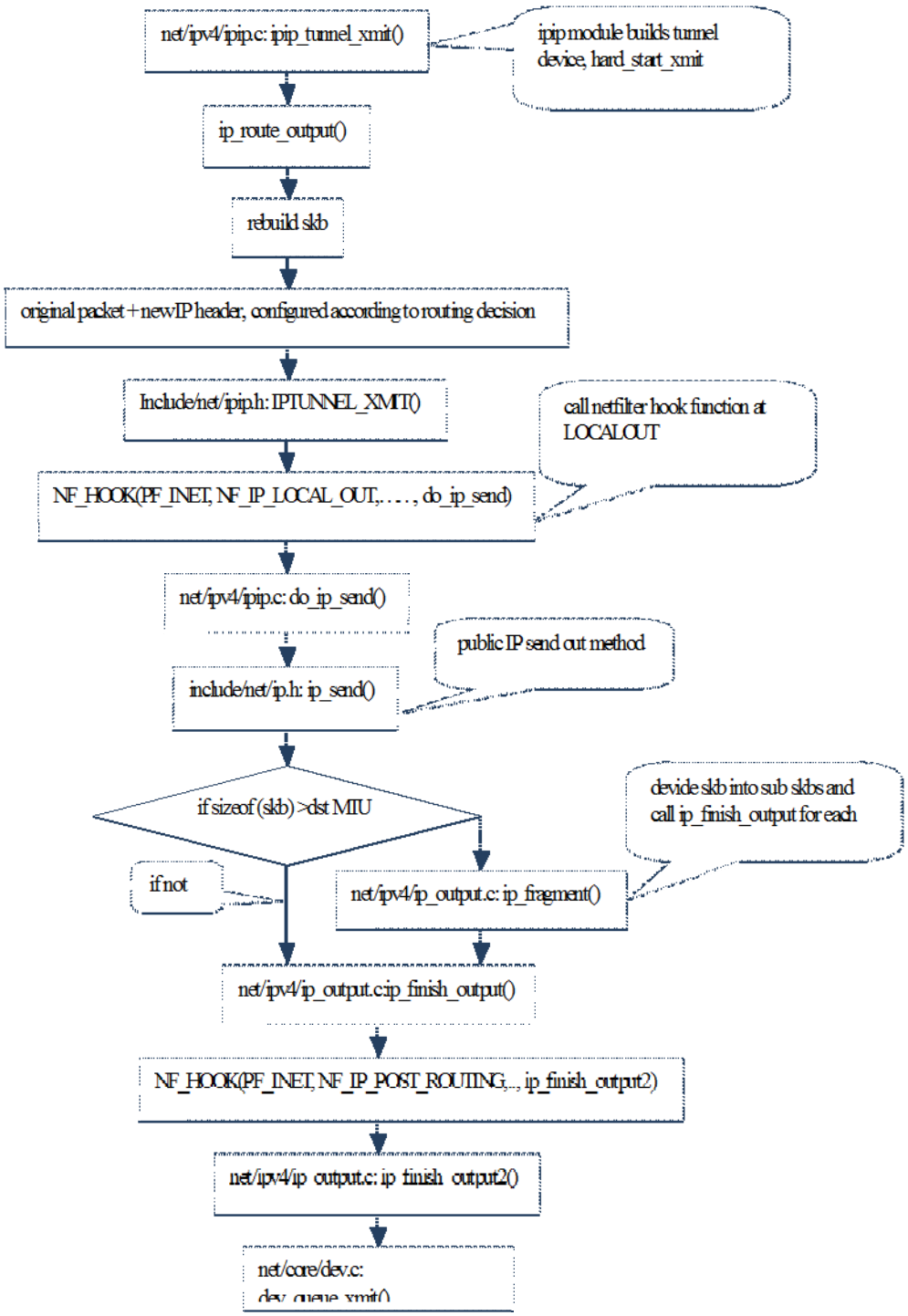

Figure 2-17. IP Packet Out

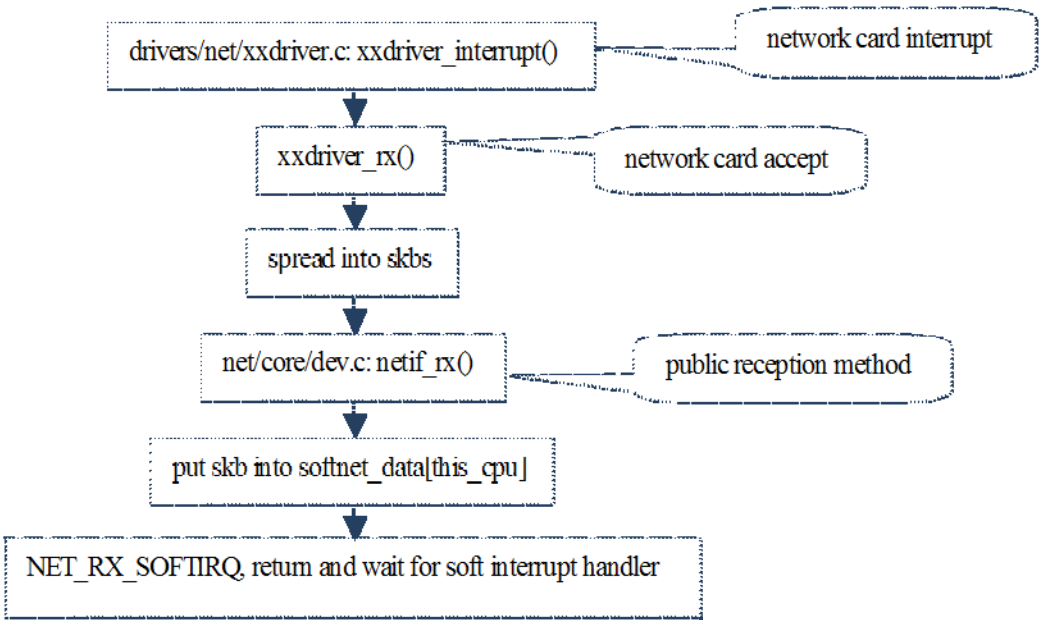

Figure 2-18. IP Packet Input Processing from The Driver

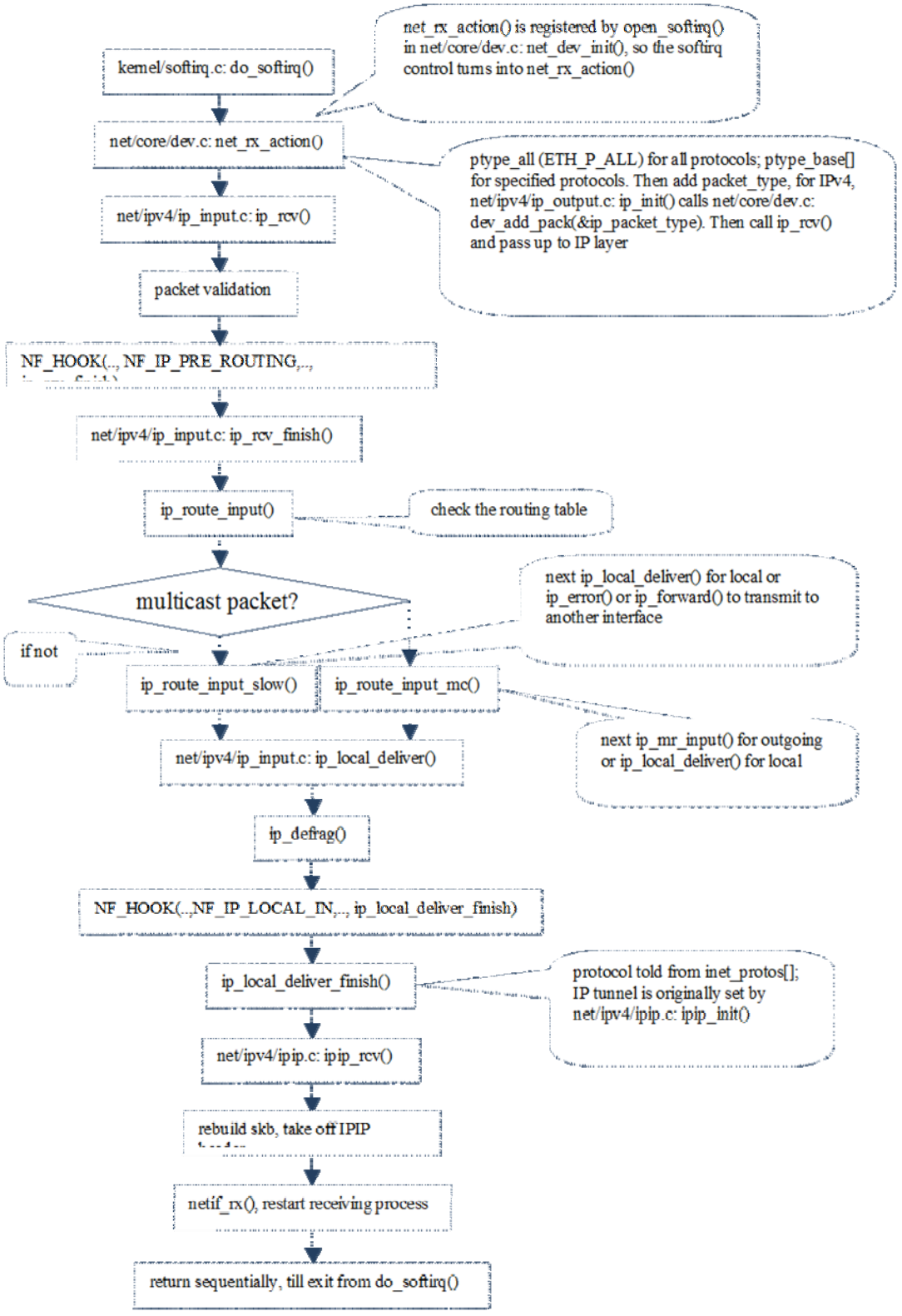

Figure 2-19. IP Packet In Processing

If a packet should be forwarded to another interface, then ip_route_input_slow() calls ip_forward(). This processing is shown in Figure 2-20.

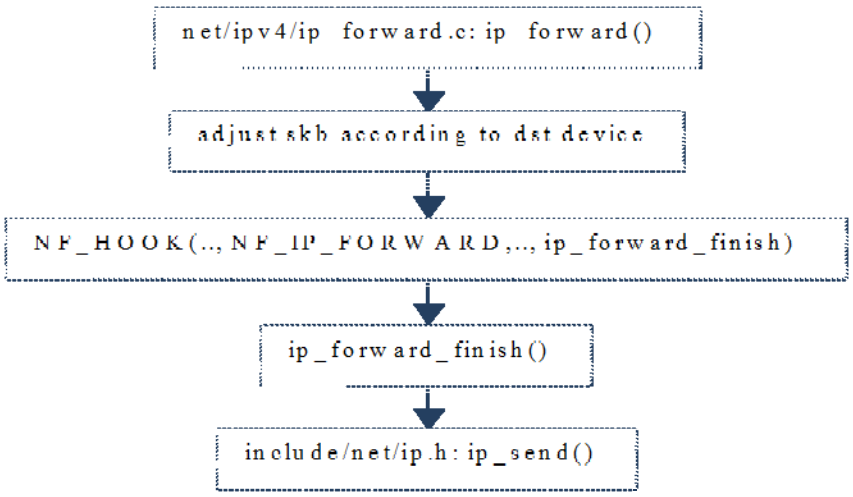

Figure 2-20. Packet forwarding

As an example, we will examine how ip_forward() to see how the rules are applied for filtering. Before ip_forward() returns, NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, dev2, ip_forward_finish) would be executed. The following code gives NF_HOOK's definition (as a macro):

```
# define NF_HOOK (pf, hook, skb, indev, outdev, okfn) \
(list_empty (&nf_hooks [(pf)] [(hook)]) ) ? (okfn) (skb) \
: nf_hook_slow( (pf), (hook), (skb), (indev), (outdev),
(okfn) )
```

If the list that nf_hooks[PF_INET][NF_IP_FORWARD] points to is empty, ip_forward() would directly call ip_forward_finish(skb), otherwise it would call nf_hook_slow to carry out netfilter handling().

Since table `filter` has three chains, it must register each of them by calling nf_register_hook() during initialization. This requires three nf_hook_ops structures, representing the chains: INPUT, FORWARD, and OUTPUT. For table `filter`, at the hook FORWARD, the handler is set as ipt_hook(), which directly calls ipt_do_table(). Subsequently ipt_do_table() works its way through the table. In practice nearly all hook handler call ipt_do_table() to process rules. The filtering process for the case of forwarding is shown in **Error! Reference source not found.**.

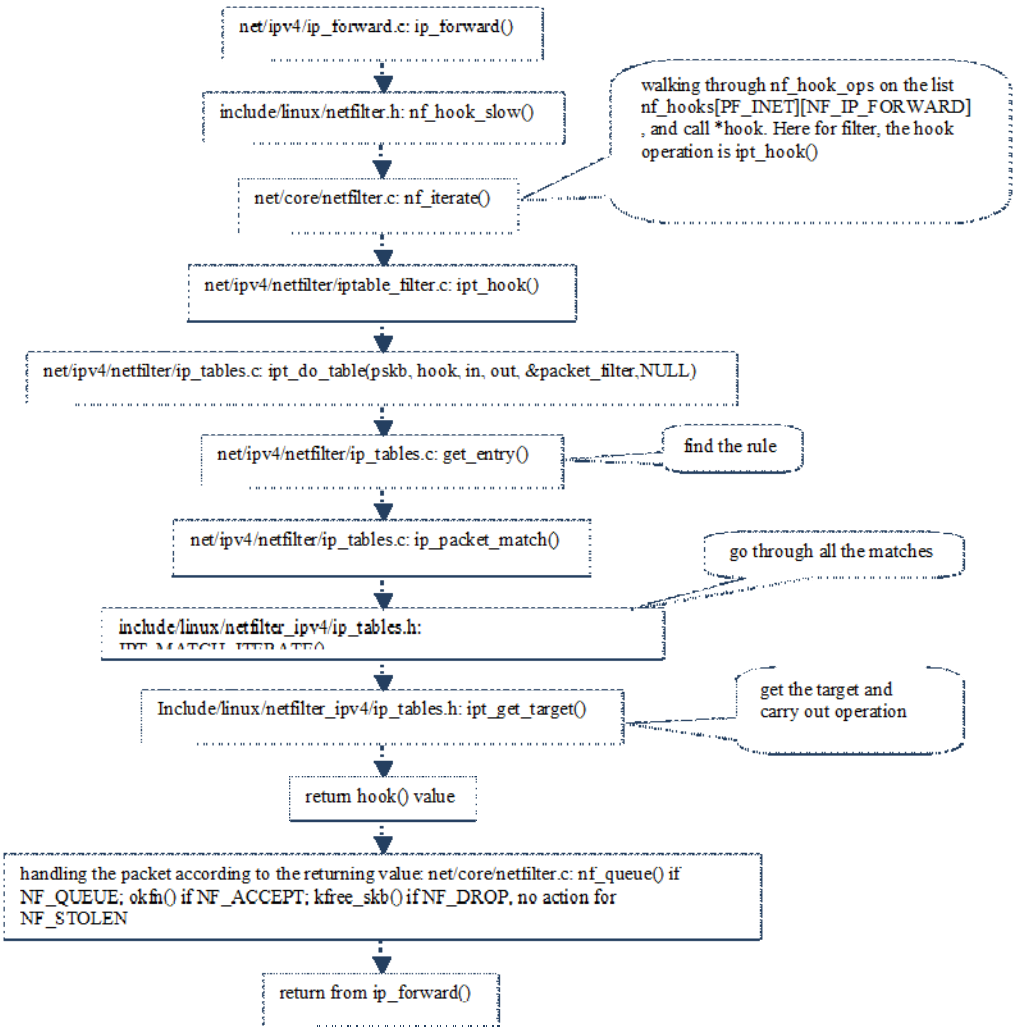

Figure 2-21. Filtering during Forwarding

Returning to the three questions proposed in the beginning of section 2.3. The first and the third questions have been answered. The second one is interesting. Although it is straight forward to do matching and targeting in nf_hookfn, it is far from simple and flexible. You need to create a new kernel module and compile it, and then you need to initialize the module when loading it to the runining kernel. The most critical thing is that such a modification is beyond maintenance and troublesome to maintain the module with changes to the kernel. In contrast, IPTable introduces an excellent abstraction for similar operations using hooks. The table is simply inserted and the rules are stored in an array attaching to the table. No longer are specific operations added into netfilter beyond an invitation for packets to be passed to the hook. Users can easily organize and dynamically edit their rules.

Unfortunately, despite IPTable's advantages, it does not fulfill our requirements. Firstly, our filtering rules change frequently (i.e., everytime a client device moves into our out of the cell). IPTable stores rules inside an array, which is easy to walk through when processing, but it is not an efficient structure when frequently updating rules. As can be seen in the existing routine ipt_replace(), replacing a table entry leads to large amounts of copying and redistribution. Secondly, each entry of our table should have a time limit associated with it. Thus after a supplicant has be authorized to access the network via this AP, the rule should automatically removed after some time limit,

unless the authorized usage period is extended. Therefore we turn our focus to another mechanism called IP Set, which is covered in the next section.

2.6 IP Set

As mentioned in the project description, we need a black list, a temp list, and a white list. The white list maintains normal traffic for authenticated users. The temp list keeps traffic of users who are only to receive a limited bandwidth and redirects authentication packets for a specified time, during which the customer is supposed to finish authentication. Otherwise the user will be moved to the black list. Blacklisted user can only send authentication traffic. Actually the black list and temp list can be unified, because the black list is simply an extreme case of bandwidth limiting with a bandwidth of zero. We can now restate our requirements:

- A list of MAC, IP addresses, and port numbers;
- Each entry of the temp list as an associated time limit;
- After authentication the corresponding entry is moved to the white list, otherwise to black list;
- Match against lists using iptables; and
- Efficiently dynamically update iptables rules for newly added clients, authenticated clients, and unauthenticated clients.

If you look at the official website of IP sets [52] the features of Ipsets seem be be just what we need. IPset is an extension of IPTables. While IPTables classifies rules into tables in a *behavior oriented* way, ipset sorts rules into sets which are *address oriented*. An IP set can be associated with an IP address, MAC address, port number, or a combination of them. An entry in one set can be bound to another set, similar to a database“join” operation, making sophisticated matching possible

There are 11 different types of sets, among which iptree is used to store IP addresses in a tree, optionally with timeout values. Since the other ten sets resemble iptree, we can easily understand them by looking only at iptree. The ipset programs work in user space, accept commands, and interacting with the kernel. These programs work together with kernel modules to implement the IPset functionality. We first create our three sets using a command of the form:

ipset -N set_name type_specification [options]

This command creates a set identified by set_name with the specified type. Type-specific options may also be supplied. The option for the set type “iptree” is a timeout value in seconds (default 0) for the entries. To specify our “whitelist” without a timeout value we use ipmap rather than iptree:

```
ipset –N blacklist iptree –timeout 1200
ipset –N templist iptree –timeout 120
ipset –N whitelist ipmap
```

Figure 2-22 explains how these commands operate.

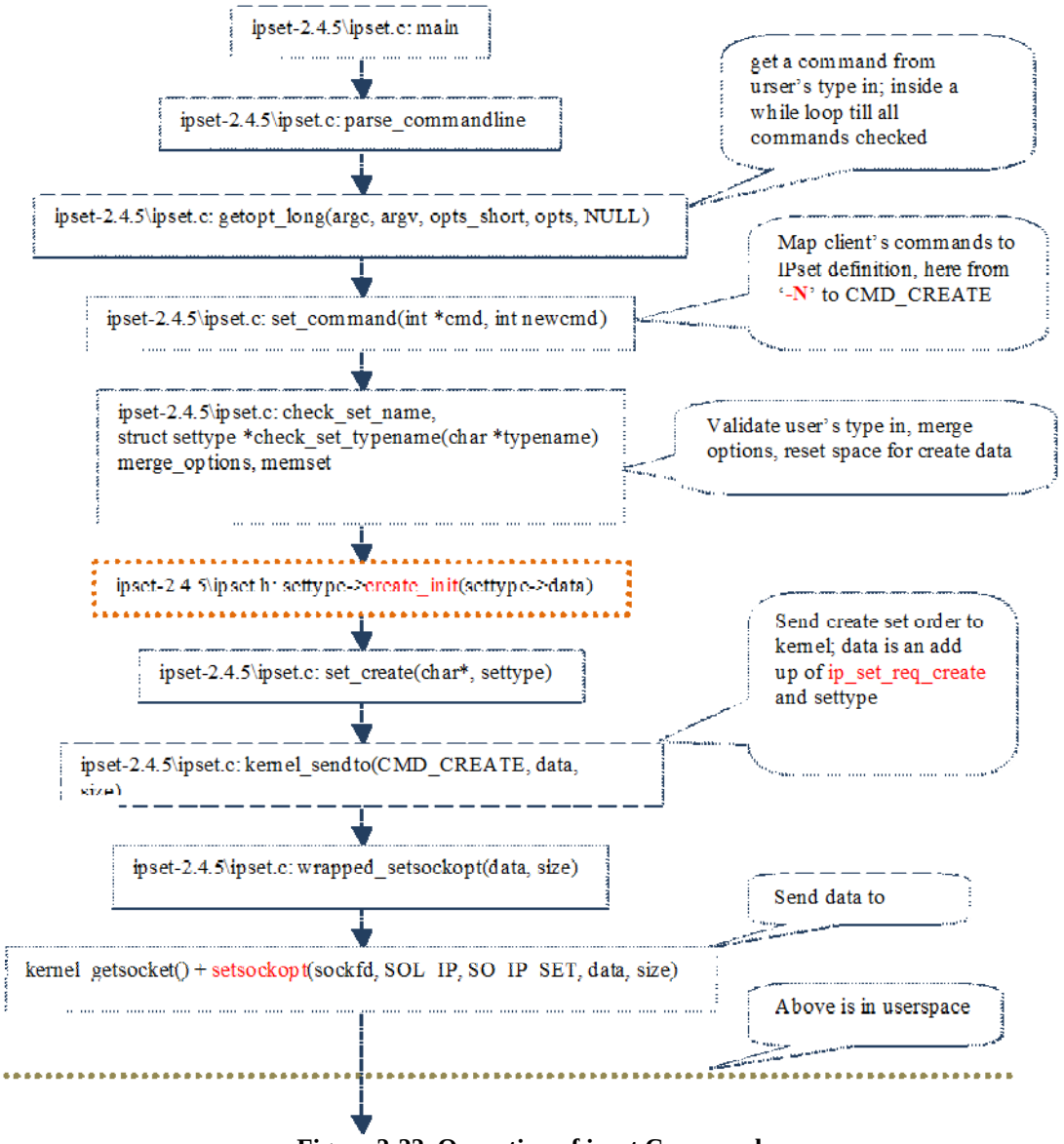

Figure 2-22. Operation of ipset Commands

Before creating a set we specify our set type. The functions settype->create_init(), settype->create_final() in method set_create(), and the default settype->create_parse() in method parse_commandline(), all call their corresponding methods in ipset_iptree.c, which implements the actual operations on an iptree.

Inside the kernel, all registered IP sets are put in a structure list_head set_type_list. Sets can be identified through an identifier (id) or by its index in ip_set_list. The id representing a unique set inside kernel never changes, but the index may change. The id can be also used to find a key in the hash of bindings.

Requests from userspace are serialized by **ip_set_mutex**. Remember that sets can be deleted only from userspace. Therefore you will see in the following code that ip_set_list locking obeys the following rules:

- kernel requests: read and write locking mandatory
- user requests: read locking optional, write locking mandatory

Wherever requests come from, their function can be divided into two parts: to set the ipset, using method ip_set_sockfn_set(), or to get information from an ipset using ip_set_sockfn_get(). These socket functions are put in the structure nf_sockopt_ops which serves as the interface to kernel sets. Entries are defined by netfilter to describe the getsockopt/setsockopt interfaces for a certain protocol. Different protocols are linked together through a structure list_head and the header of this list is defined in net/core/netfilter.c: nf_sockopts(struct list_head). **Error! Reference source not found.** shows the relationship between these chains.

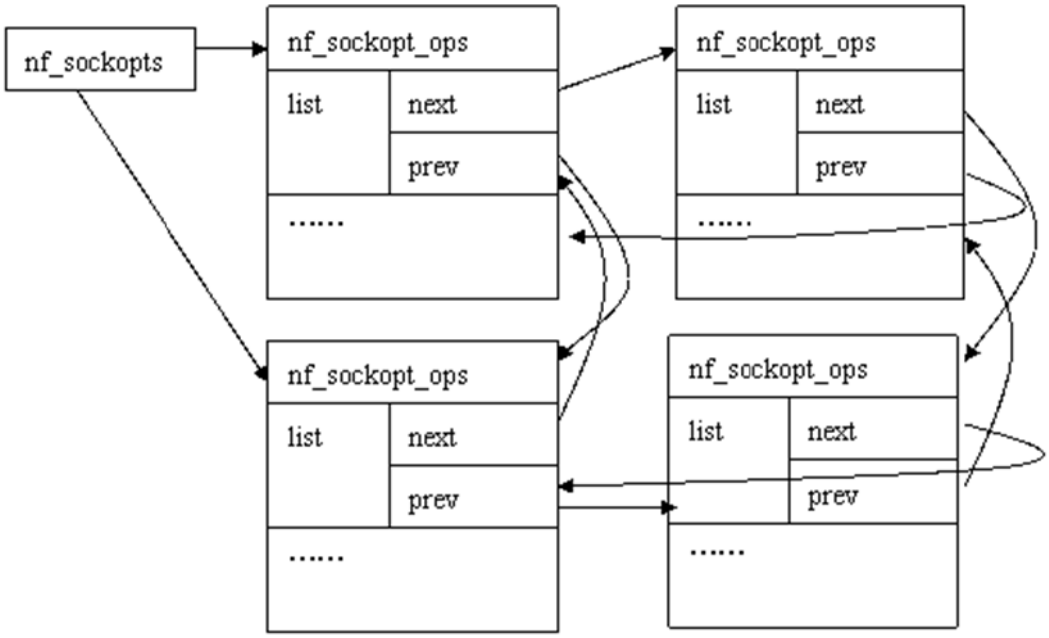

Figure 2-23. Chains of nf_sockopt_ops

For IPset, its own nf_sockopt_ops is given in ipset-2.4.5\kernel\ip_set.c:

```
static struct nf_sockopt_ops so_set = {
   .pf       = PF_INET,
   .set_optmin  = SO_IP_SET,
   .set_optmax  = SO_IP_SET + 1,
   .set      = &ip_set_sockfn_set,
   .get_optmin  = SO_IP_SET,
   .get_optmax  = SO_IP_SET + 1,
   .get      = &ip_set_sockfn_get,
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,23)
   .use      = 0,
#else
   .owner    = THIS_MODULE,
#endif
};
```

In order to create an iptree, the function iptree_create() in kernel/ip_set_iptree.c is invoked. Figure 2-24 shows how we create a new set, starting with a request from userspace. This request was due to a call to the method ip_set_sockfn_set().

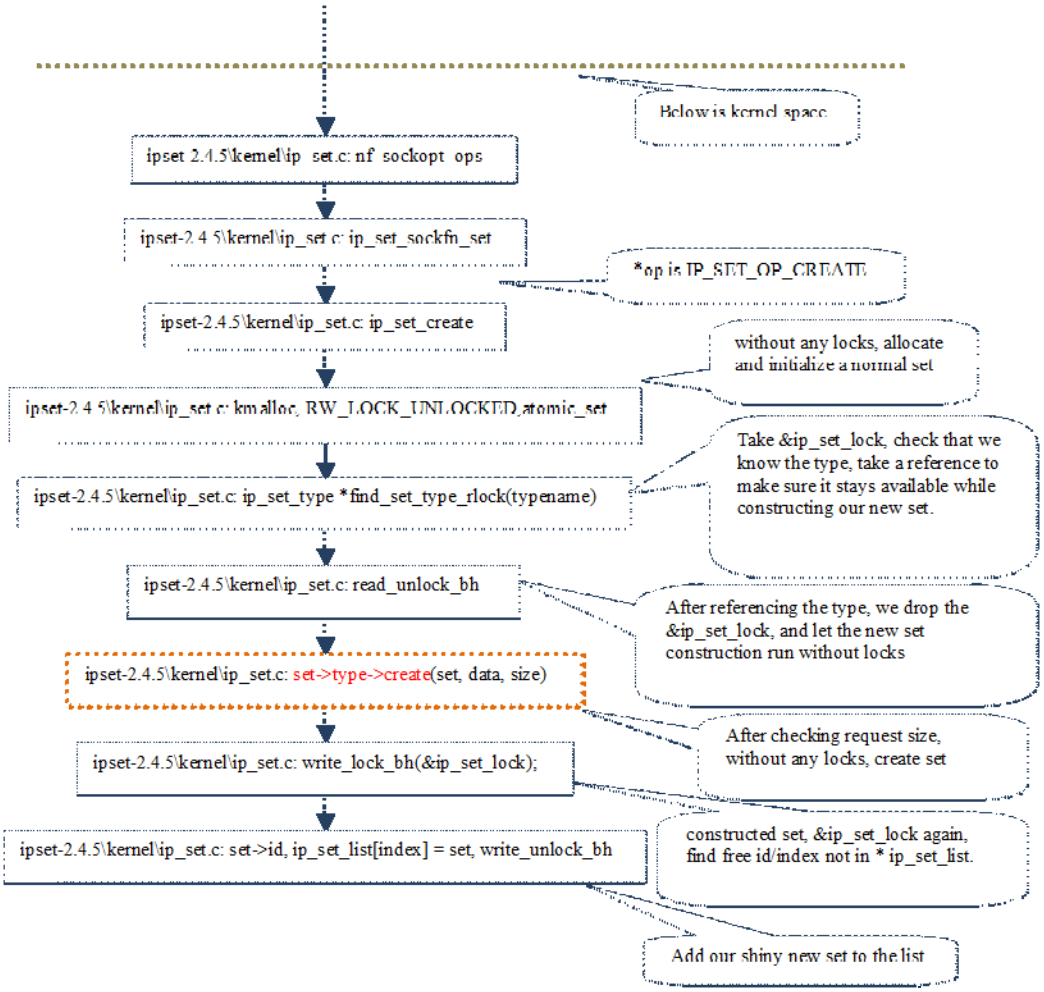

Figure 2-24. The Process of Creating A New Set

After determing the type and performing the correct locking, the type specific operations are carried out by their own methods. For example, in order to add an IP address to iptree, after parsing the command whee the type is iptree, the userspace program calls iptree's adt_parser(), which is used to add, delete, and test the parser. Here "adt" stands for add, delete, and test. After the command and parameters are transferred into kernel, eventually the funciton ip_set_sockfn_set() will be called. Each type of operation is placed in a structure fn_table, and the various types form an array adtfn_table[] as shown below:

```
/* ipset-2.4.5\kernel\ip_set.c */
struct fn_table {
      int (*fn)(ip_set_id_t index,
           const void *data, size_t size);
  } adtfn_table[] = {
    { ip_set_addip }, { ip_set_delip }, { ip_set_testip},
    { ip_set_bindip}, { ip_set_unbindip },
{ ip_set_testbind },
  };
```

So we get the type of operation by comparing the commands:

```
adtfn = adtfn_table[*op - IP_SET_OP_ADD_IP].fn
```

In the case of CMD_ADD, the function table would invoke method iptree_add() in ip_set_iptree.c, which would call ADDIP_WALK three times to finish the task of adding the IP address. Before we examine this function, we need first to know that an iptree is organized as follows:

```
/* ipset-2.4.5\kernel\include\linux\netfilter_ipv4\
ip_set_iptree.h */
struct ip_set_iptreed {
  unsigned long expires[256];     /* x.x.x.ADDR */
};

struct ip_set_iptreec {
  struct ip_set_iptreed *tree[256];  /* x.x.ADDR.* */
};

struct ip_set_iptreeb {
  struct ip_set_iptreec *tree[256];  /* x.ADDR.*.* */
};

struct ip_set_iptree {
  unsigned int timeout;
  unsigned int gc_interval;
#ifdef __KERNEL__
  uint32_t elements;     /* number of elements */
  struct timer_list gc;
  struct ip_set_iptreeb *tree[256];  /* ADDR.*.*.* */
#endif
};
```

The last 8 bits of the IP address are represented by an unsigned long, and since there are 256 different values, they are put in an array – unsigned long [256]. So structure ip_set_iptreed represents the last 8 bits of an IP address. The structure ip_set_iptreec represents the third octet of the IP address. As we know ip_set_iptreec has 256 different values and each should contain an ip_set_iptreed. The same approach applies to ip_set_iptreeb and finally ip_set_iptree, which represents all 65536 possibilities, or so called entries. Now let us turn to the macro ADDIP_WALK:

```
/* ipset-2.4.5\kernel\ip_set_iptree.c */
#define ADDIP_WALK(map, elem, branch, type, cachep) do {\
  if ((map)->tree[elem]) {           \
     DP("found %u", elem);           \
     branch = (map)->tree[elem];        \
  } else {                  \
     branch = (type *)            \
        kmem_cache_alloc(cachep, GFP_ATOMIC); \
     if (branch == NULL)          \
        return -ENOMEM;           \
     memset(branch, 0, sizeof(*branch));       \
     (map)->tree[elem] = branch;        \
     DP("alloc %u", elem);           \
  }                     \
} while (0)
```

When invoked this code will first detect if the entry (tree[elem]) at the map level exists. If so, it would branch to the next level through the entry. Before looking into how ADDIP_WALK is utilized in the method iptree_add, we introduce another macro ABCD:

```
/* ipset-2.4.5\kernel\ip_set_iptree.c */
#if defined(__LITTLE_ENDIAN)
#define ABCD(a,b,c,d,addrp) do {      \
  a = ((unsigned char *)addrp)[3];    \
  b = ((unsigned char *)addrp)[2];    \
  c = ((unsigned char *)addrp)[1];    \
  d = ((unsigned char *)addrp)[0];    \
} while (0)
#elif defined(__BIG_ENDIAN)
#define ABCD(a,b,c,d,addrp) do {      \
  a = ((unsigned char *)addrp)[0];    \
  b = ((unsigned char *)addrp)[1];    \
  c = ((unsigned char *)addrp)[2];    \
  d = ((unsigned char *)addrp)[3];    \
} while (0)
```

According to host form or network form; a,b,c, and d point to 4 8-bit field of an IP address. Thus iptree_add() is simply a series of calls to ADDIP_WALK, as follows:

```
/* ipset-2.4.5\kernel\ip_set_iptree.c */
Static inline int iptree_add(struct ip_set *set, ip_set_ip_t
*hash_ip,
ip_set_ip_t ip, unsigned int timeout)
{ …
  ABCD(a, b, c, d, hash_ip);
  ADDIP_WALK(map, a, btree, struct ip_set_iptreeb,
branch_cachep);
  ADDIP_WALK(btree, b, ctree, struct ip_set_iptreec,
branch_cachep);
  ADDIP_WALK(ctree, c, dtree, struct ip_set_iptreed,
leaf_cachep);
  …
}
```

Imagine we want to insert the Ipv4 address 192.168.0.2. We first check if there is entry 192 at the first level, if not, we create a new one. Otherwise we enter its second level and check if an entry 168 exists, continuing in this way to the last 8 bits.

There are many other behaviors, such as test for an entry or deleting an entry, to print or restore a set, to save, and execute a session. They should be easy to understand after the two examples above.

As you may have observed, each entry in the iptree can be viewed as a pointer, hence we judge if an entry exists by telling if the value is not NULL. We can not prune empty branches by calling delete from userspace because iptree_del() simply reduces the value to zero but zero is still a non-NULL value. The real removal function is iptree_flush():

```
/* ipset-2.4.5\kernel\ip_set_iptree.c */
static void iptree_flush(struct ip_set *set)
{
  struct ip_set_iptree *map = set->data;
  unsigned int timeout = map->timeout;

  /* gc might be running */
  while (!del_timer(&map->gc))
     msleep(IPTREE_DESTROY_SLEEP);
  __flush(map);
  memset(map, 0, sizeof(*map));
  map->timeout = timeout;

  init_gc_timer(set);
}
```

This function first checks if a timer is running. When this time is up, it calls __flush(map) and then resets the set and initialize the timer. Inside __flush(), the code walks through the whole tree and releases every entries' unused memory by calling kmem_cache_free(), as shown below:

```
static inline void
__flush(struct ip_set_iptree *map)
{
  struct ip_set_iptreeb *btree;
  struct ip_set_iptreec *ctree;
  struct ip_set_iptreed *dtree;
  unsigned int a,b,c;

  LOOP_WALK_BEGIN(map, a, btree);
  LOOP_WALK_BEGIN(btree, b, ctree);
  LOOP_WALK_BEGIN(ctree, c, dtree);
  kmem_cache_free(leaf_cachep, dtree);
  LOOP_WALK_END;
  kmem_cache_free(branch_cachep, ctree);
  LOOP_WALK_END;
  kmem_cache_free(branch_cachep, btree);
  LOOP_WALK_END;
  map->elements = 0;
}
```

You might wonder why the timer is part of the iptree. No matter how you create a new iptree (by iptree_create or flush and reset a tree by iptree_flush), there the function init_gc_timer is always called. This function is shown below:

```
static inline void init_gc_timer(struct ip_set *set)
{
  struct ip_set_iptree *map = set->data;
  map->gc_interval = IPTREE_GC_TIME;
  init_timer(&map->gc);
  map->gc.data = (unsigned long) set;
  map->gc.function = ip_tree_gc;
  map->gc.expires = jiffies + map->gc_interval * HZ;
  add_timer(&map->gc);
}
```

As you can see the core function is init_timer, with a parameter timer_list. Next the function assigns a value to timer.function and timer.expires. Add a timer and when the time is up the function in timer.function will be called. To delete a timer just call del_timer(&timer). The timer associated with an iptree is used to periodically invoke garbage collection, to prune unneeded entries from the tree and to return this space to the kernel.

Next we examine how IPset relates to netfilter. Inside ipt_set.h there are structures for “match info” and “target info”. These are used for registering IPset as an entry in IPTable. These structures and related functions are:

```
/*
ipset-2.4.5\kernel\include\linux\netfilter_ipv4\ipt_set.h
*/
struct ipt_set_info {
  ip_set_id_t index;
  u_int32_t flags[IP_SET_MAX_BINDINGS + 1];
};

/* match info */
struct ipt_set_info_match {
  struct ipt_set_info match_set;
};

struct ipt_set_info_target {
  struct ipt_set_info add_set;
  struct ipt_set_info del_set;
};
```

They are used by ipt_SET_init in ipt_SET.c to get registered in IPTABLE:

```
/* ipset-2.4.5\kernel\ipt_SET.c */
#define xt_register_target  ipt_register_target
...
static int __init ipt_SET_init(void)
{
  return xt_register_target(&SET_target);
}
```

From user’s point of view, IPsets are used together with IPtables:

```
/* ipset-2.4.5\kernel\include\linux\netfilter_ipv4\ip_set.h
*/
/* API for iptables set match, and SET target */
extern int ip_set_addip_kernel(ip_set_id_t id,
              const struct sk_buff *skb,
              const u_int32_t *flags);
extern int ip_set_delip_kernel(ip_set_id_t id,
              const struct sk_buff *skb,
              const u_int32_t *flags);
extern int ip_set_testip_kernel(ip_set_id_t id,
            const struct sk_buff *skb,
            const u_int32_t *flags);
```

Now if you review our requirements as listed at the beginning of this section, you can easily understand how well IPset fits these requirements. IPTables had rules which were costly to modify. In our case the rules are very stable, but the IP addresses associated with a rule are frequently changed. IPSet binds IP addresses associated

with the same rule together. The rules are registered as an entry in iptables and serve as target in that entry, so that they must be walked if the entry is invoked by the iptables command. We can dynamically update iptables with new addresses by simply manipulating IPSets, rather than replacing the old table. As you have see from the structure of ip_set_iptree, it is well optimized to match an IP address against a set, since to find an address it is divided into four parts which are used as indexes to arrays. This is likely to be faster than walking through all the chains in IPTables.

2
Method

3 Method

As mentioned in Section2.3, we group the AP's function modules for AAA conversion into three parts: (1) a sender and receiver of EAPOL on the AP's wireless side; (2) a RADIUS client on AP's LAN side; and (3) a set of state machines cooperating with each other to implement the logic of IEEE 802.1X (these state machines are labelled 802.1X SM in the figure). **Error! Reference source not found.** shows this simplified architecture.

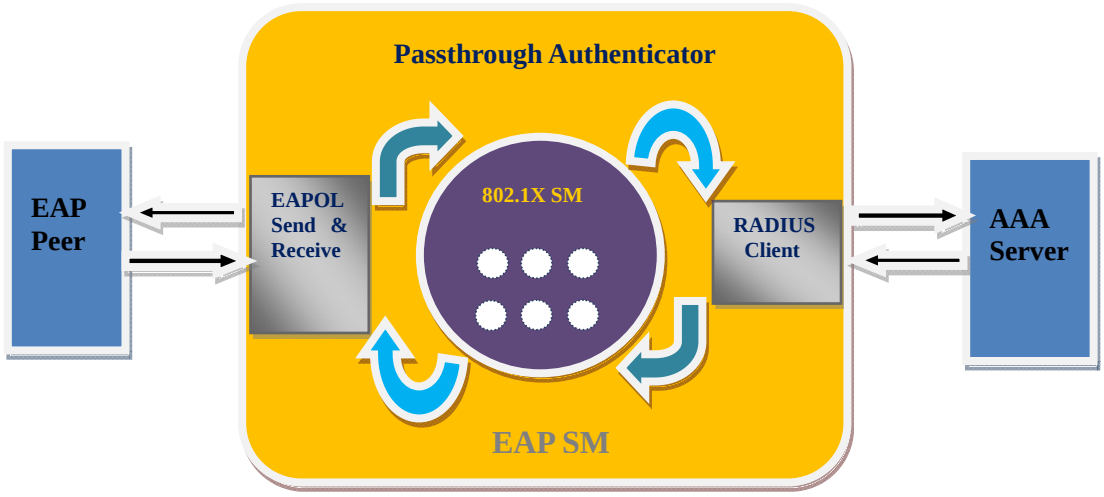

Figure 3-1. Simplified AP Architecture

The definition of components in Figure 3-1 and the interfaces between them are described in various IEEE standards and IETF RFCs. Sometimes these descriptions use different names for the same component. For example, the IEEE 802.1X standard refers to both the supplicant and the AP as a Port Access Entity (PAE), because both have the same two components, a set of Port Access Control Protocol (PACP) state machines and a *higher layer* with which these machines communicate. The difference between the two terms is that the higher layer of the supplicant PAE implements EAP functionality, while that of the authenticator PAE implements a combination of EAP and AAA functionality [33]. EAP is a big state machine (marked as *EAP SM* in Figure 3-1) which takes charge of the whole access control logic. The *EAP state machine* of the supplicant and the AP are different. In RFC 4137 [51], the *EAP state machine* functionality is refered to as the "*EAP layer*". *AAA functionality* is the same as the *RADIUS Client* in Figure 3-1. The *PACP state machines* are labelled as *802.1X SM* in Figure 3-1 while they are referred to as *EAPOL state machines* in hostapd was shown in **Error! Reference source not found.** on page **Error! Bookmark not defined.**. IEEE 802.1X defines an encapsulation format that allows EAP message to be carried directly by a LAN MAC service [33]. IEEE 802.1X also defines the PACP state machines and the interface between these PACP state machines and the higher-layer functionality. The EAP protocol exchanges are defined by the IETF's EAP standards, IETF RFC 3748 [37], and successor standards. For example the RADIUS AAA protocol is defined by the IETF RADIUS standards: RFC 2865 [54], IETF RFC 2866 [13], IETF RFC 3579 [40], and successor standards [33]. The descriptions of EAP SM can be found in RFC 4137. Among these various documents IEEE 802.1X-2004 and RFC 4137 are vital to this thesis. Both of them use a hierarchical structure as shown

in Figure 3-2 which is similar to **Error! Reference source not found.** but in a vertical way.

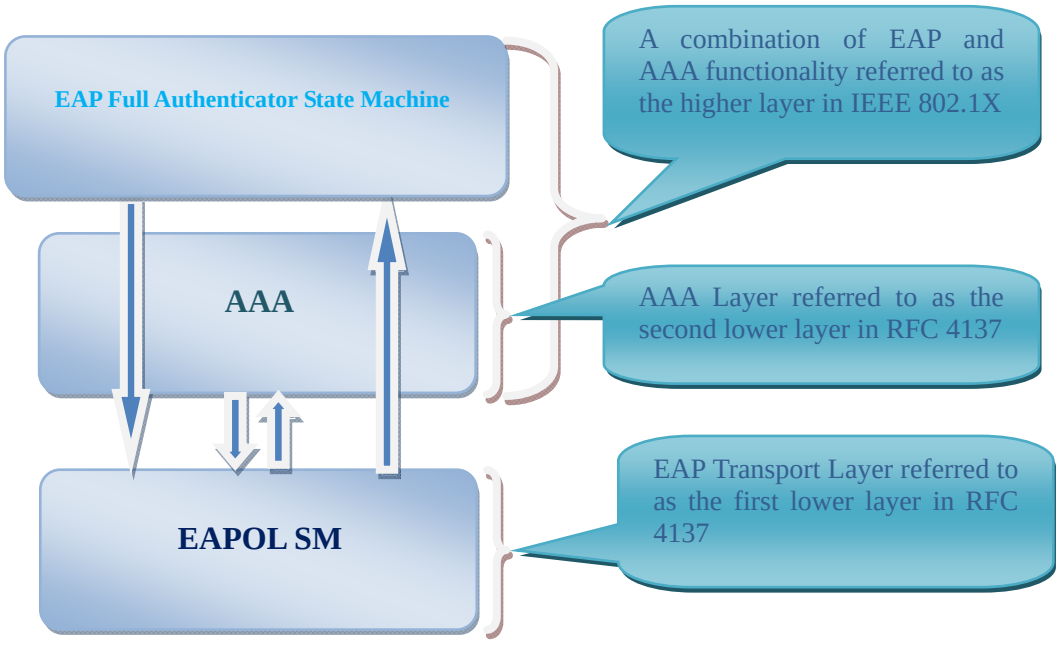

Figure 3-2. Hierarchical AP Architecture

IEEE 802.1X defines the EAPOL State Machines and treats both the AAA functionality and the EAP State Machine as the higher layer. However, RFC 4137 [51] focuses on the EAP State Machine, treating both the AAA functionality and the EAPOL SM as the lower layer. Both Figure 3-2 and **Error! Reference source not found.** on page **Error! Bookmark not defined.** show that, the EAP state machine directly communicates with the EAPOL SM; while EAPOL SM directly communicates with the AAA layer. This raises the question: Is there any direct interaction between the AAA layer and the EAP layer? If so, what is the reason behind this direct interaction? The question is answered in the end of Section 3.2.2 on page 84.

Following the hierarchical architecture in Figure 3-2, the next section will describe the EAP state machines; Section 3.2 will introduce the interfaces between the EAP layer and the AAA layer, and it will cover the AAA functions at length; Section 0 will introduce the interfaces between the EAP layer and the EAPOL layer, and it will present the EAPOL state machines; Section 3.4 will examine the interaction between the EAPOL Sender & Receiver and the EAPOL layer; Section 3.5 will discuss how to convert an AP into a non-binary authenticator by modifying hostapd. Finally Section 3.6 will examine another method of implementing the desired functionality.

3.1 State Machines for EAP

Most information of the information is this section about EAP state machines is from RFC 4137: State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator [51]. This RFC describes four different types of EAP state machines: peer, stand-alone authenticator (non-pass-through), EAP backend

authenticator (for use on AAA servers), and EAP full authenticator (for both local and pass-through).

The peer and stand-alone authenticator state machines are illustrative of how EAP as defined in RFC 3748 [37] may be implemented. An EAP authentication consists of one or more EAP methods in sequence followed by an EAP Success or EAP Failure sent from the authenticator to the peer [51]. Both the authenticator and the peer can implement one or more EAP methods. That is why each EAP peer has to select its choice of method and negotiate with its counterpart. The negotiation will determine which EAP method will be used, as well as the sequence of methods if more than one method will be used. This negotiation of EAP methods and sequences of methods is controlled by the "EAP Switch". The "EAP Switch Model" comprises events and actions for the interaction between the EAP Switch and EAP methods. The methods may also have state machines, which is beyond our scope. **Error! Reference source not found.** shows the EAP switch model for the stand-alone authenticator scenario.

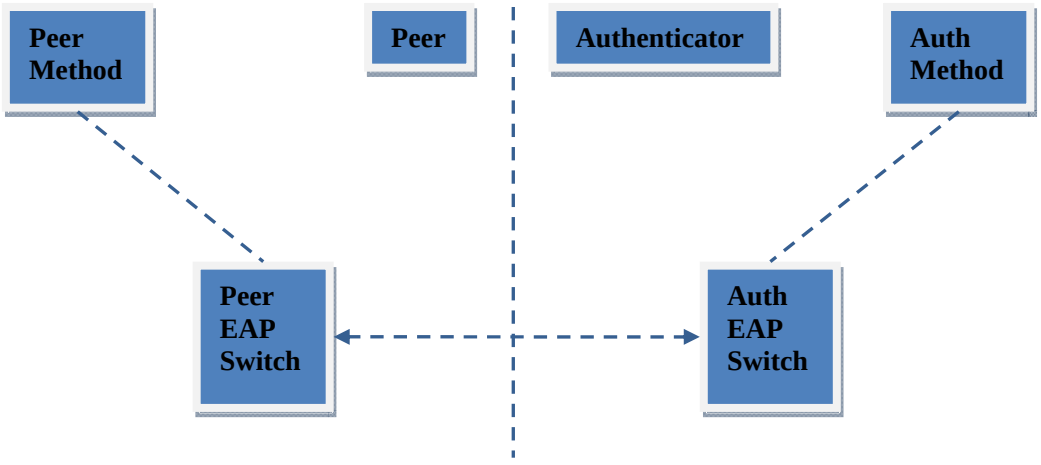

Figure 3-3. Stand-Alone EAP Switch Model [51]

The backend and full/pass-through authenticator machines illustrate how EAP/AAA protocol support defined in RFC 3579 [40] may be implemented. The full/pass-through state machine allows an NAS or edge device to pass EAP Response messages to a backend authentication server. A stand-alone authenticator carries out authentication locally, while a full authenticator can choose either to perform local authentication or remote authentication. **Error! Reference source not found.** shows the EAP switch model for the pass-through authenticator scenario.

1
Method

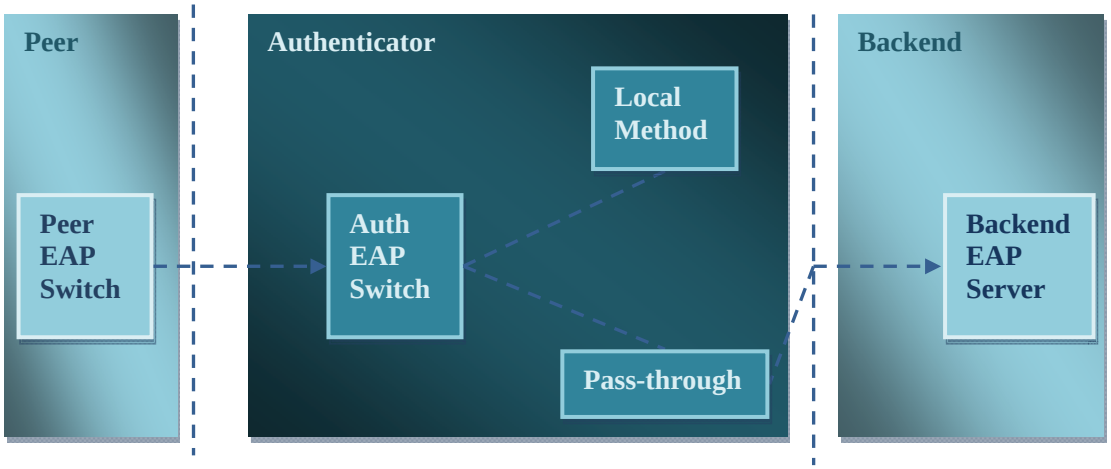

Figure 3-4. Pass-Through EAP Switch Model [51]

We use the full/pass-through authenticator for our project, thus we omit the EAP peer state machine which bears rare similarity to the EAP authenticator state machines. **Error! Reference source not found.** shows the EAP stand-alone authenticator state machine. Figure 3-6 shows the EAP backend authenticator state machine. They are identical to each other except that no retransmit is included in the IDLE state in the backend authenticator state machine. The reason is that with RADIUS, retransmission is handled by the NAS. Also, a PICK_UP_METHOD state and a variable in INITIALIZE state are added to the backend authenticator state machine to allow the method to "pick up" a method started in a NAS [51].

2
Method

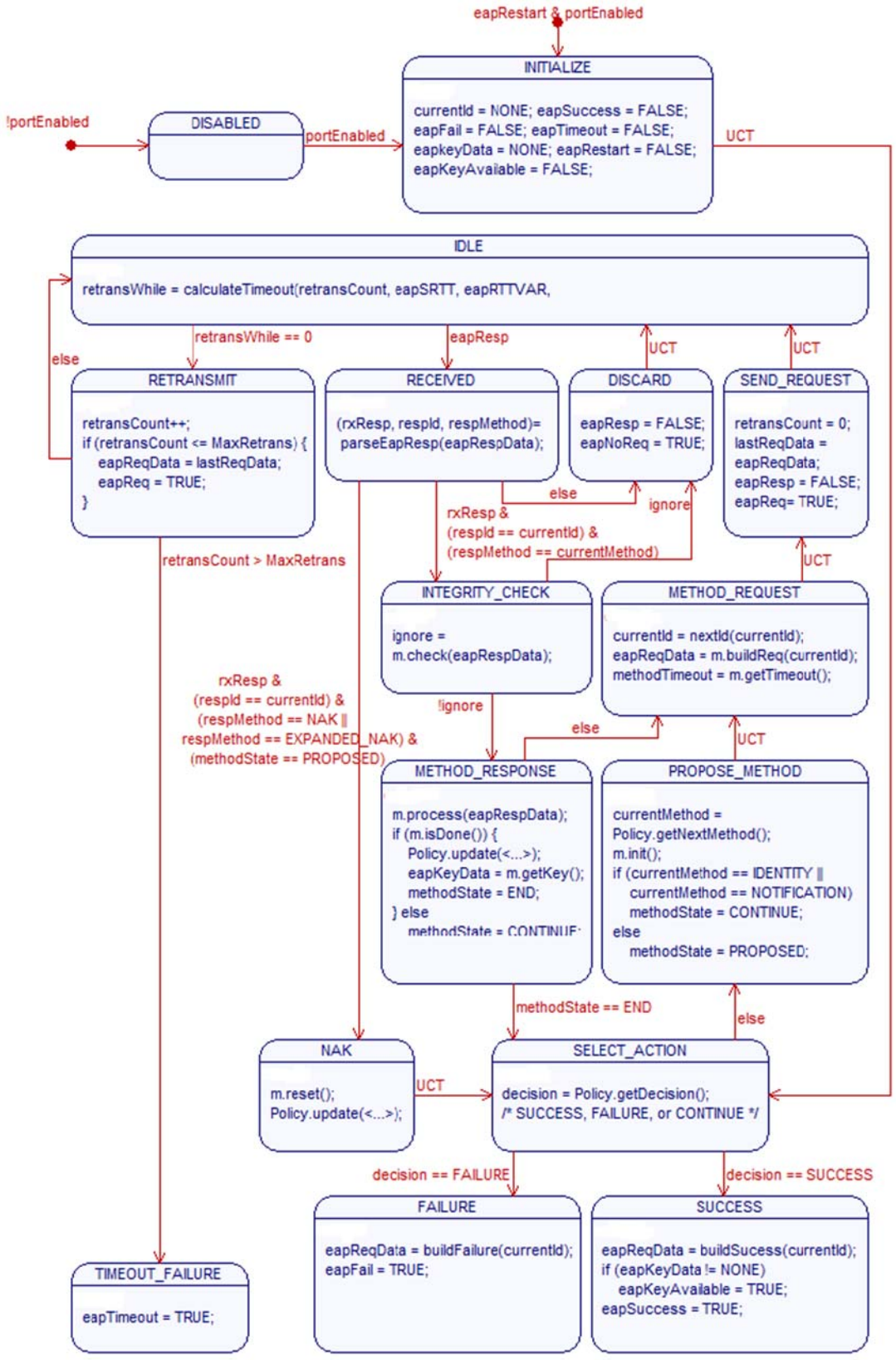

Figure 3-5. EAP Stand-Alone Authenticator State Machine [51]

1
Method

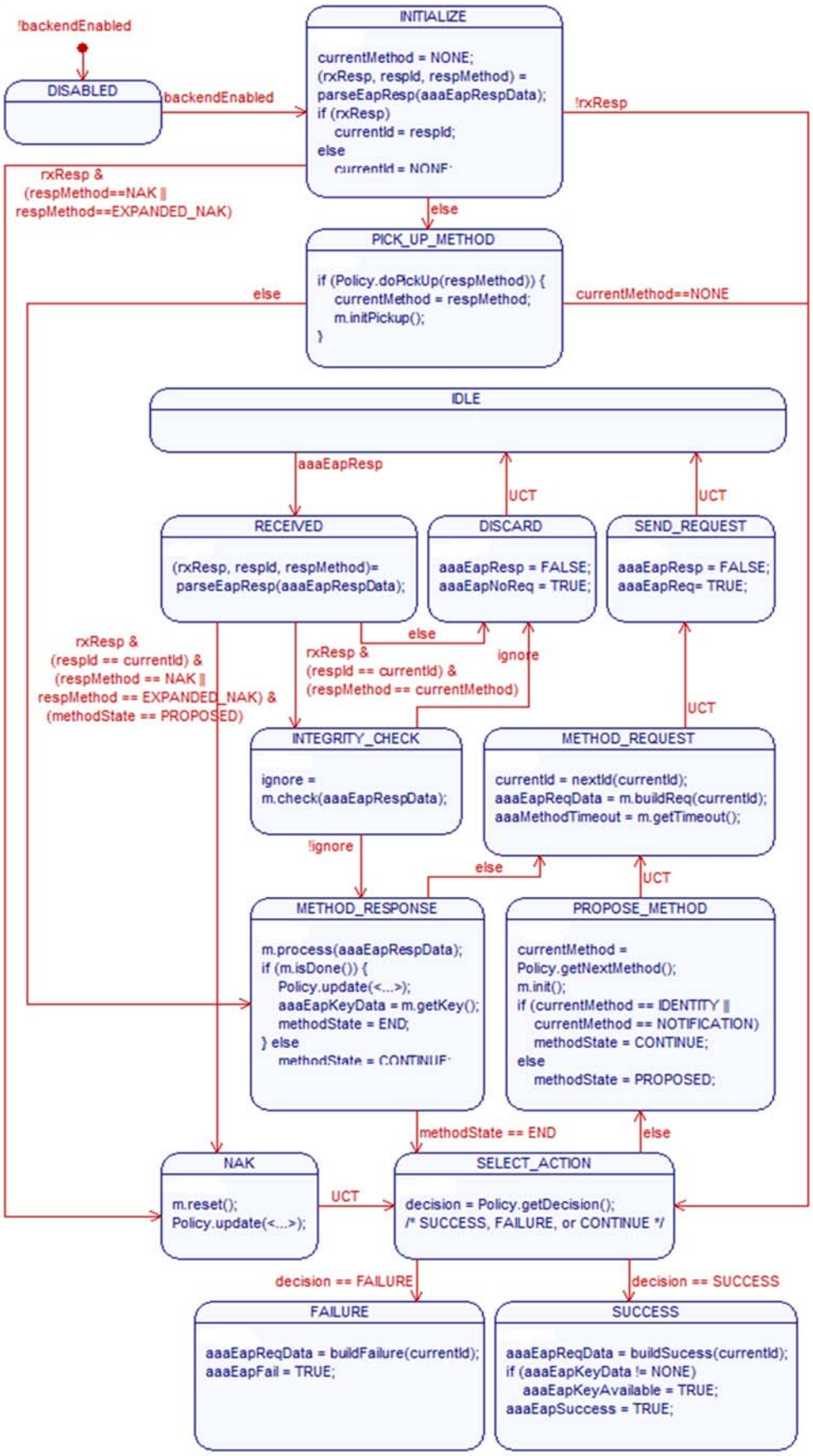

Figure 3-6. EAP Backend Authenticator State Machine [51]

2
Method

1

Method

Error! Reference source not found. shows the EAP full authenticator state machine for a local AAA. It is identical to the stand-alone state machine, with the exception that the SELECT_ACTION state has an additional transition to PASSTHROUGH.

2
Method

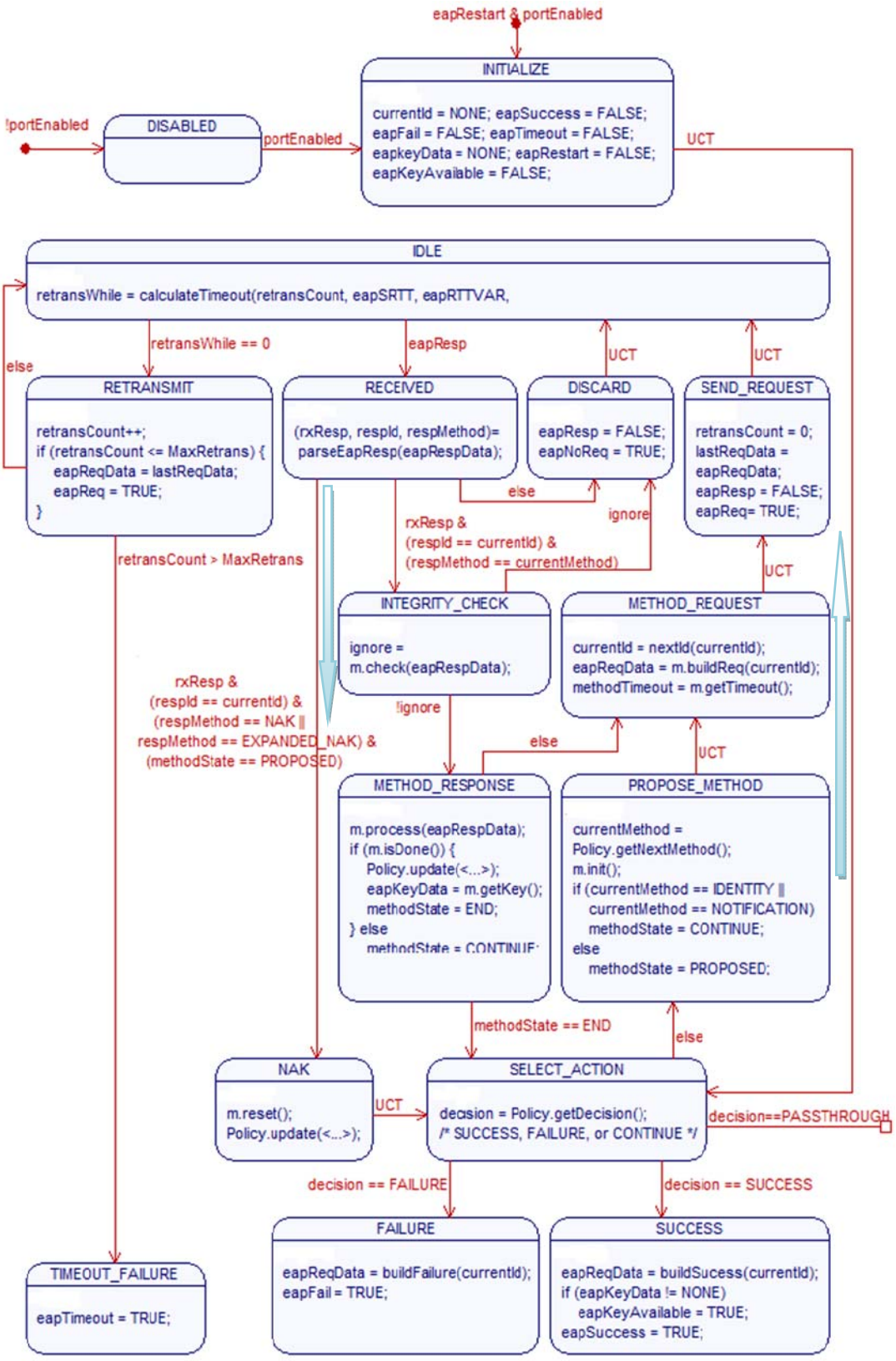

Figure 3-7. EAP Full Authenticator State Machine - 1 [51]

Error! Reference source not found. shows the remainder of the EAP full authenticator state machine in the case of pass-through for a remote AAA. Compared

with **Error! Reference source not found.**, retains most of the logic, except the four method states. Since the EAP SM layer interacts directly with the AAA layer when the RADIUS client is activated under pass-through mode, there are three replacement AAA states: AAA_REQUEST, AAA_RESPONSE, AAA_IDLE, which are responsible for handling EAP responses from the supplicant, RADIUS responses from the AAA server, and idleness respectively. All the other states have the suffix "2" to help distinguish them from their counterparts in **Error! Reference source not found.**.

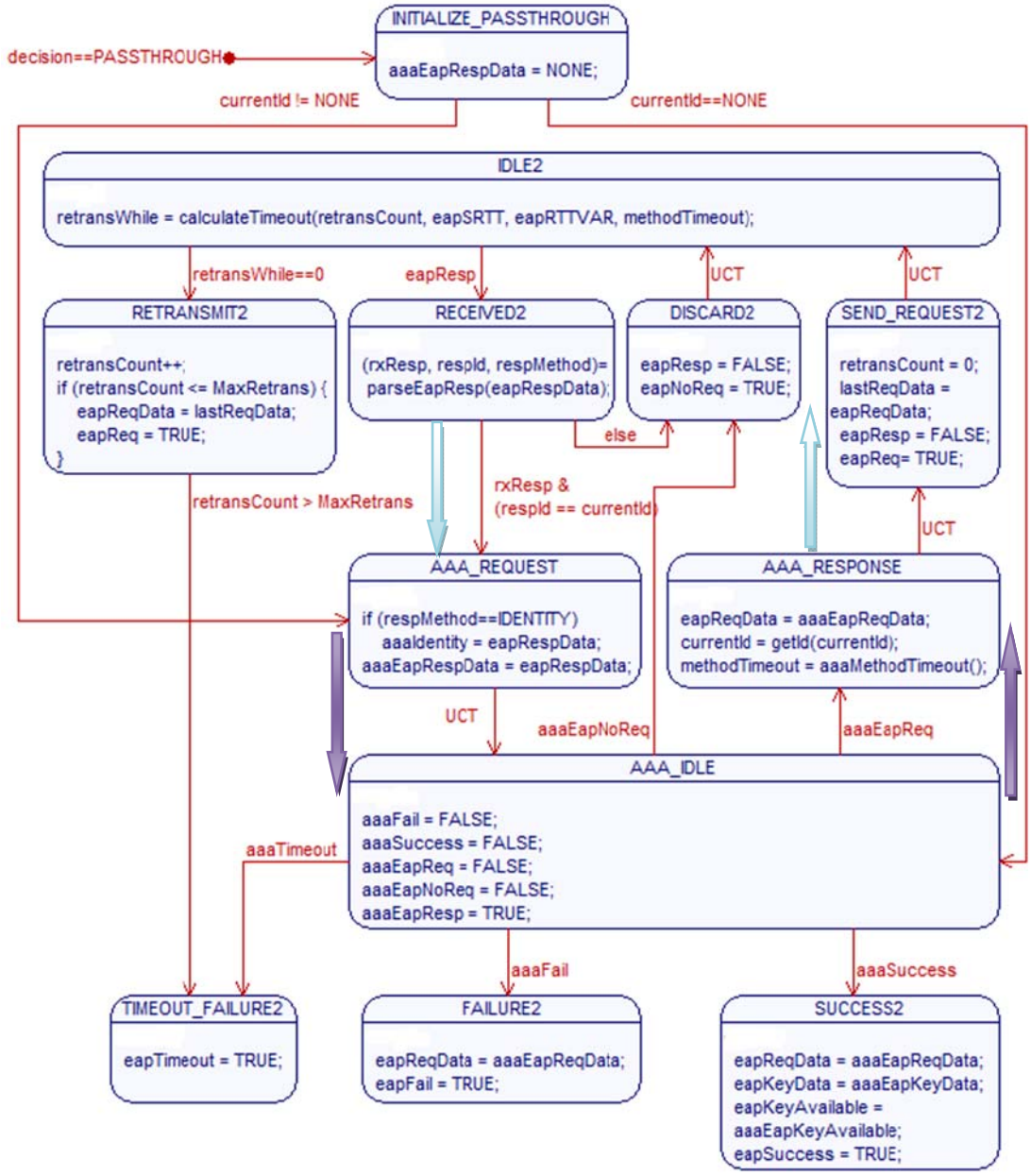

Figure 3-8. EAP Full Authenticator State Machine - 2 [51]

The lower layers that the EAP state machine maps to mainly do two jobs: present messages to the EAP SM and transmit the results created by the EAP SM. Peer state machine and stand-alone state machines interface to the EAPOL SM layer only. The backend authenticator state machine interfaces to the AAA layer only. However, the

2
Method
full authenticator state machine is unique in that it interfaces to multiple lower layers when operating under pass-through mode. The EAPOL SM is responsible for transmitting messages between the EAPOL Sender/Receiver and the EAP SM while the AAA layer is responsible for transmitting messages between the RADIUS client and the EAP SM. In **Error! Reference source not found.** the four method states organize the logic for the EAP switch. But in **Error! Reference source not found.** , the pass-through authenticator relies on the responses from the AAA server to judge whether the result is SUCCESS or FAILURE or if it should continue relaying the message. Therefore, the three AAA states replace the four method states.

In RFC4137 [51], the state machine description is carried out in the sequence shown in Table 3-1.

Table 3-1. SM Comparison between Stand-Alone and Full Authenticator

Sequence	Stand-Alone	Full
1 Interface between SM and Lower layer		
1.1 Variables (Lower Layer to SM)	EAPOL	EAPOL + AAA
1.2 Variables (SM to Lower Layer)	EAPOL	EAPOL + AAA
1.3 Constants	same	
2 Interface between SM and Methods	same	
3 SM Local Variables		
3.1 Long-Term	same	
3.2 Short-Term	only “*decision*” is different	
4 Procedures	same	
5 States	No Pass-Through	+ Pass-Through

Since our scenario uses the pass-through authenticator, we focus on the state machine presented in **Error! Reference source not found.**. Its further study will be divided into three parts. The remainder of this section describes the states in **Error! Reference source not found.** (pass-through only) – see section 3.1.1, local variables – see section 3.1.3, and procedures – see section 3.1.4; as well as the interface between SM and methods – see section 3.1.5. Section 3.1.2 lists and describes the constant(s). It is important to note that these constants belong to the interface between the state machines and the lower layer. Section 3.2 on page 72 will cover the interface related to AAA and Section 0 on page 91 will cover the interface related to EAPOL SM.

3.1.1 EAP Full Authenticator States under Pass-Through Mode

All of the states (see **Error! Reference source not found.**) in the EAP Full Authenticator state machine when operating under pass-through mode are described in Table 3-2.

Table 3-2. EAP Full Authenticator States under Pass-Through Mode [51]

State	**Purpose**
INITIALIZE_PASSTHROUGH	Initializes variables when the pass-through portion of the state machine is activated.
IDLE2	The state machine waits for a response from the primary lower layer (EAPOL SM), which transports EAP traffic from the peer.
RECEIVED2	This state is entered when an EAP packet is received and the authenticator is in PASSTHROUGH mode. The packet header is parsed here.
AAA_REQUEST	The incoming EAP packet is parsed for sending to the AAA server.
AAA_IDLE	Tell the AAA layer that it has a response and then waits for a new request, a no-request signal, or success/failure.
AAA_RESPONSE	The request from the AAA interface is processed into an EAP request.
SEND_REQUEST2	This state signals the lower layer (EAPOL SM) that a request packet is ready to be sent.
DISCARD2	This state signals the lower layer that the response was discarded, and that no new request packet will be sent at this time.
RETRANSMIT2	It retransmits the previous request packet.
SUCCESS2	A final state indicating success.
FAILURE2	A final state indicating failure.
TIMEOUT_FAILURE2	A final state indicating failure because no response has been received. Because **no response was received**, no new message (including failure) should be sent to the peer. This is different from the FAILURE2 state, in which a message indicating failure is sent to the peer.

3.1.2 Constants

There is only one constant: ***MaxRetrans*** (integer) a configurable maximum count indicating how many retransmissions should be attempted before aborting. Note that this constant belongs to the interface between the EAP State Machine and Lower Layer.

3.1.3 Local Variables

A list of long-term variables and their descriptions are given in Table 3-3. They are referred to as long term **variables, since the state is maintained between packets. Note that the names of the states in Table 3-2 have implicit postifx "2" to correspond to their states. In contrast to long term variables,**

Table 3-4 shows **short-term** variables whose state is **not** maintained between packets.

Table 3-3. Long Term Variables [51]

Variable	**Description**
currentMethod (EAP type)	EAP type, IDENTITY, or NOTIFICATION.
currentId (integer)	0..255 or NONE. Usually updated in PROPOSE_METHOD state. Indicates the identifier value of the currently outstanding EAP request.
methodState (enumeration)	As described in Table 3-2.
retransCount (integer)	Reset in SEND_REQUEST state and updated in RETRANSMIT state. Current number of retransmissions.
lastReqData (EAP packet)	Set in SEND_REQUEST state. EAP packet contains the last sent request.
methodTimeout (integer)	Method-provided hint for suitable retransmission timeout or NONE.

Table 3-4. Short Term Variables [51]

Variable	**Description**
rxResp (boolean)	Set in RECEIVED state. Indicates that the current received packet is an EAP response.
respId (integer)	Set in RECEIVED state. The identifier from the current EAP response.
respMethod (EAP type)	Set in RECEIVED state. The value is the method type of the current EAP response.
ignore (boolean)	Set in SELECT_ACTION state. Temporarily stores the policy decision to succeed, fail, continue with a local method, or continue in pass-through mode.
decision (enumeration)	Set in SELECT_ACTION state. Temporarily stores the policy decision to succeed, fail, or continue.

3.1.4 Procedures

For methods / procedures, the method uses its internal state in addition to the information provided by the EAP layer. The only arguments that are explicitly shown as inputs to the procedures are those provided to the method by EAP. Those inputs provided by the method's internal state remain implicit [51].

Table 3-5. Methods [51]

Method	**Description**
calculateTimeout()	Calculates the retransmission timeout, taking into account the retransmission count, round-trip time measurements, and method-specific timeout hint (see [37], Section 4.3). Returns an integer.
parseEapResp()	Determines the code, identifier value, and type of the current response. In the case of a parsing error (e.g., the length field is longer than the received packet), rxResp will be set to FALSE. The values of respId and respMethod may be undefined as a result. Returns a

1
Method

Method	**Description**
	boolean, an integer, and an EAP type.
buildSuccess()	Creates an EAP Success Packet. Returns an EAP packet.
buildFailure()	Creates an EAP Failure Packet. Returns an EAP packet.
nextId()	Determines the next identifier value to use, based on the previous one. Returns an integer.
Policy.update()	Updates all variables related to internal policy state. The return value is undefined.
Policy.getNextMethod()	Determines the method that should be used at this point in the conversation based on a predefined policy. Policy.getNextMethod() **must** comply with [37] (Section 2.1), which forbids the use of sequences of authentication methods within an EAP conversation. Thus, if an authentication method has already been executed within an EAP dialog, Policy.getNextMethod() **must not** propose another authentication method within the same EAP dialog. Returns an EAP type.
Policy.getDecision()	Determines if the policy will allow SUCCESS, FAIL, or is yet to be determined (CONTINUE). Returns a decision enumeration.
m.check()	Method-specific procedure to test for the validity of a message. Returns a boolean.
m.process()	Parses and processed a response for that method. The return value is undefined.
m.init()	Initializes state just before use. The return value is undefined.
m.reset()	Indicates that the method is ending in the middle of or before completion. The return value is undefined.
m.isDone()	To check for method completion. Returns a boolean.
m.getTimeout()	Determines an appropriate timeout hint for that method. Returns an integer.
m.getKey()	Obtains key material for use by EAP or lower layers. Returns an EAP key.
m.buildReq()	Produces the next request. Returns an EAP packet.

3.1.5 Interface between EAP SM and Methods

The following describes the interaction between the EAP state machine and EAP methods. The implict input parameters are ***IN:*** eapRespData, methodState; ***OUT:*** ignore, eapReqData; and ***IN/OUT:*** currentId, (method-specific state), (policy).

Table 3-6. Interface between EAP SM and Methods [51]

Method	**Description**
m.init (in: -, out: -)	When the method is first started, it must initialize its own method-specific state, possibly using some information from Policy (e.g., identity).
m.buildReq (in: integer, out: EAP packet)	Next, the method creates a new EAP Request packet, with the given identifier value, and updates its methodspecific state accordingly.

2

Method

Method	**Description**
m.getTimeout (in: -, out: integer or NONE)	The method can also provide a hint for retransmission timeout with m.getTimeout.
m.check (in: EAP packet, out: boolean)	When a new EAP Response is received, the method must first decide whether to process the packet or to discard it silently. If the packet looks like it was not sent by the legitimate peer (e.g., if it has an invalid Message Integrity Check, which should never occur), the method can indicate this by returning FALSE. In this case, the method should not modify its own method-specific state.
m.process (in: EAP packet, out: -)	
m.isDone (in: -, out: boolean)	
m.getKey (in: -, out: EAP key or NONE)	Next, the method processes the EAP Response and updates its own method-specific state. Now the options are to continue the conversation (send another request) or to end this method. If the method wants to end the conversation, it ● Tells Policy about the outcome of the method and possibly other information. ● If the method has derived keying material it wants to export, returns it from m.getKey(). ● Indicates that the method wants to end by returning TRUE from m.isDone(). Otherwise, the method continues by sending another request, as described earlier.

3.1.6 EAP SM Data Structure in hostapd

In hostapd, the structure *eap_sm* is defined to describe the EAP server state machine data. As is shown below, this also includes EAP states, constants, long-term local variables, short-term local variables, plus related data structures which are *not* defined in RFC 4137. The interfaces between EAP layer and AAA layer as well as EAPOL layer are placed together in a structure *eap_eapol_interface*, which is also included in the structure eap_sm.

```
/**
 * hostapd-0.7.3\src\eap_server\eap_i.h: struct eap_sm
 */
struct eap_sm {
	enum {
		EAP_DISABLED, EAP_INITIALIZE, EAP_IDLE, EAP_RECEIVED,
		EAP_INTEGRITY_CHECK, EAP_METHOD_RESPONSE,
		EAP_METHOD_REQUEST,
		EAP_PROPOSE_METHOD, EAP_SELECT_ACTION, EAP_SEND_REQUEST,
		EAP_DISCARD, EAP_NAK, EAP_RETRANSMIT, EAP_SUCCESS, EAP_FAILURE,
		EAP_TIMEOUT_FAILURE, EAP_PICK_UP_METHOD,
		EAP_INITIALIZE_PASSTHROUGH, EAP_IDLE2, EAP_RETRANSMIT2,
		EAP_RECEIVED2, EAP_DISCARD2, EAP_SEND_REQUEST2,
		EAP_AAA_REQUEST, EAP_AAA_RESPONSE, EAP_AAA_IDLE,
```

```
        EAP_TIMEOUT_FAILURE2, EAP_FAILURE2, EAP_SUCCESS2
    } EAP_state;


    /* Constants */
    int MaxRetrans;

    struct eap_eapol_interface eap_if;

    /* Full authenticator state machine local variables */

    /* Long-term (maintained betwen packets) */
    EapType currentMethod;
    int currentId;
    enum {
        METHOD_PROPOSED, METHOD_CONTINUE, METHOD_END
    } methodState;
    int retransCount;
    struct wpabuf *lastReqData;
    int methodTimeout;

    /* Short-term (not maintained between packets) */
    Boolean rxResp;
    int respId;
    EapType respMethod;
    int respVendor;
    u32 respVendorMethod;
    Boolean ignore;
    enum {
        DECISION_SUCCESS, DECISION_FAILURE, DECISION_CONTINUE,
        DECISION_PASSTHROUGH
    } decision;

    /* Miscellaneous variables */
    const struct eap_method *m; /* selected EAP method */
    /* not defined in RFC 4137 */
    Boolean changed;
    void *eapol_ctx, *msg_ctx;
    struct eapol_callbacks *eapol_cb;
    void *eap_method_priv;
    u8 *identity;
    size_t identity_len;
    /* Whether Phase 2 method should validate identity match */
    int require_identity_match;
    int lastId; /* Identifier used in the last EAP-Packet */
    struct eap_user *user;
    int user_eap_method_index;
    int init_phase2;
    void *ssl_ctx;
    void *eap_sim_db_priv;
    Boolean backend_auth;
    Boolean update_user;
    int eap_server;

    int num_rounds;
    enum {
        METHOD_PENDING_NONE, METHOD_PENDING_WAIT,
        METHOD_PENDING_CONT
    } method_pending;
```

```
        u8 *auth_challenge;
        u8 *peer_challenge;


        u8 *pac_opaque_encr_key;
        u8 *eap_fast_a_id;
        size_t eap_fast_a_id_len;
        char *eap_fast_a_id_info;
        enum {
                NO_PROV, ANON_PROV, AUTH_PROV, BOTH_PROV
        } eap_fast_prov;

        int pac_key_lifetime;
        int pac_key_refresh_time;
        int eap_sim_aka_result_ind;
        int tnc;
        struct wps_context *wps;
        struct wpabuf *assoc_wps_ie;

        Boolean start_reauth;
};
```

3.1.7 Data Structure of EAP SM & AAA Interface in hostapd

In hostapd structure *eap_eapol_interface* is defined to describe the interfaces between EAP SM and AAA as well as EAPOL SM. Sections 3.2 and 3.3 will start by introducing each of the corresponding variables related to its own interface.

```
/**
/*
  * hostapd-0.7.3\src\eap_server\eap.h: struct eap_eapol_interface
  */
struct eap_eapol_interface {
        /* Lower layer to full authenticator variables */
        Boolean eapResp; /* shared with EAPOL Backend Authentication */
        struct wpabuf *eapRespData;
        Boolean portEnabled;
        int retransWhile;
        Boolean eapRestart; /* shared with EAPOL Authenticator PAE */
        int eapSRTT;
        int eapRTTVAR;

        /* Full authenticator to lower layer variables */
        Boolean eapReq; /* shared with EAPOL Backend Authentication */
        Boolean eapNoReq; /* shared with EAPOL Backend Authentication */
        Boolean eapSuccess;
        Boolean eapFail;
        Boolean eapTimeout;
        struct wpabuf *eapReqData;
        u8 *eapKeyData;
        size_t eapKeyDataLen;
        Boolean eapKeyAvailable; /* called keyAvailable in IEEE 802.1X-2004 */

        /* AAA interface to full authenticator variables */
        Boolean aaaEapReq;
        Boolean aaaEapNoReq;
        Boolean aaaSuccess;
```

1

Method

```
        Boolean aaaFail;
        struct wpabuf *aaaEapReqData;
        u8 *aaaEapKeyData;
        size_t aaaEapKeyDataLen;
        Boolean aaaEapKeyAvailable;
        int aaaMethodTimeout;

        /* Full authenticator to AAA interface variables */
        Boolean aaaEapResp;
        struct wpabuf *aaaEapRespData;
        /* aaaIdentity -> eap_get_identity() */
        Boolean aaaTimeout;
};
```

3.2 AAA Layer

Table 3-77 shows the variables in the AAA Interface to Full Authenticator. While

2

Method

Table 3- shows the variables in the Full Authenticator Interface to AAA.

Table 3-7. Variables (AAA Interface to Full Authenticator) [51]

Variable	**Description**
aaaEapReq (boolean)	Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates that a new EAP request is available from the AAA server.
aaaEapNoReq (boolean)	Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates that the most recent response has been processed, but that there is no new request to send.
aaaSuccess (boolean)	Set to TRUE in lower layer. Indicates that the AAA backend authenticator has reached the SUCCESS state.
aaaFail (boolean)	Set to TRUE in lower layer. Indicates that the AAA backend authenticator has reached the FAILURE state.
aaaEapReqData (EAP packet)	Set in the lower layer when aaaEapReq, aaaSuccess, or aaaFail is set to TRUE. The actual EAP request to be sent (or success/ failure).
aaaEapKeyData (EAP key)	Set in lower layer when keying material becomes available from the AAA server.
aaaEapKeyAvailable (boolean)	Set to TRUE in the lower layer if keying material is available. The actual key is stored in aaaEapKeyData.
aaaMethodTimeout (integer)	Method-provided hint for suitable retransmission timeout, or NONE. (Note that this hint is for the EAP retransmissions done by the pass-through authenticator, not for retransmissions of AAA packets.)

1
Method

Table 3-8. Variables (Full Authenticator Interface to AAA) [51]

Variable	**Description**
aaaEapResp (boolean)	Set to TRUE in authenticator state machine, FALSE in the lower layer. Indicates that an EAP response is available for processing by the AAA server.
aaaEapRespData (EAP packet)	Set in authenticator state machine when eapResp is set to TRUE. The EAP packet to be processed.
aaaIdentity (EAP packet)	Set in authenticator state machine when an IDENTITY response is received. Makes that identity available to AAA lower layer.
aaaTimeout (boolean)	Set in AAA_IDLE if, after a configurable amount of time, there is no response from the AAA layer. The AAA layer in the NAS is alive and OK, but for some reason it has not received a valid Access-Accept/Reject indication from the backend.

The explanation for how AAA module functions in hostapd is divided into two parts: Receiving and Sending. For each sub section we start with a process flow chart, then we explain the functions in a top-down manner and end up with an analysis of how the interfaces described above come into effect.

3.2.1 RADIUS Client on Receiving

Figure 3-9 shows how the RADIUS client receives and processes frames from the AAA server. Moreover, it gives a complete top-down view of how the AAA client is initiated and how it interacts with the other parts of the system.

The code analysis is based on the last stable hostapd version 0.6.9, while the newest stable version is 0.7.3. Although in the new version the process may not exactly follow the map with some functionalities are moved from its original function to another, the main logic doesn’t change.

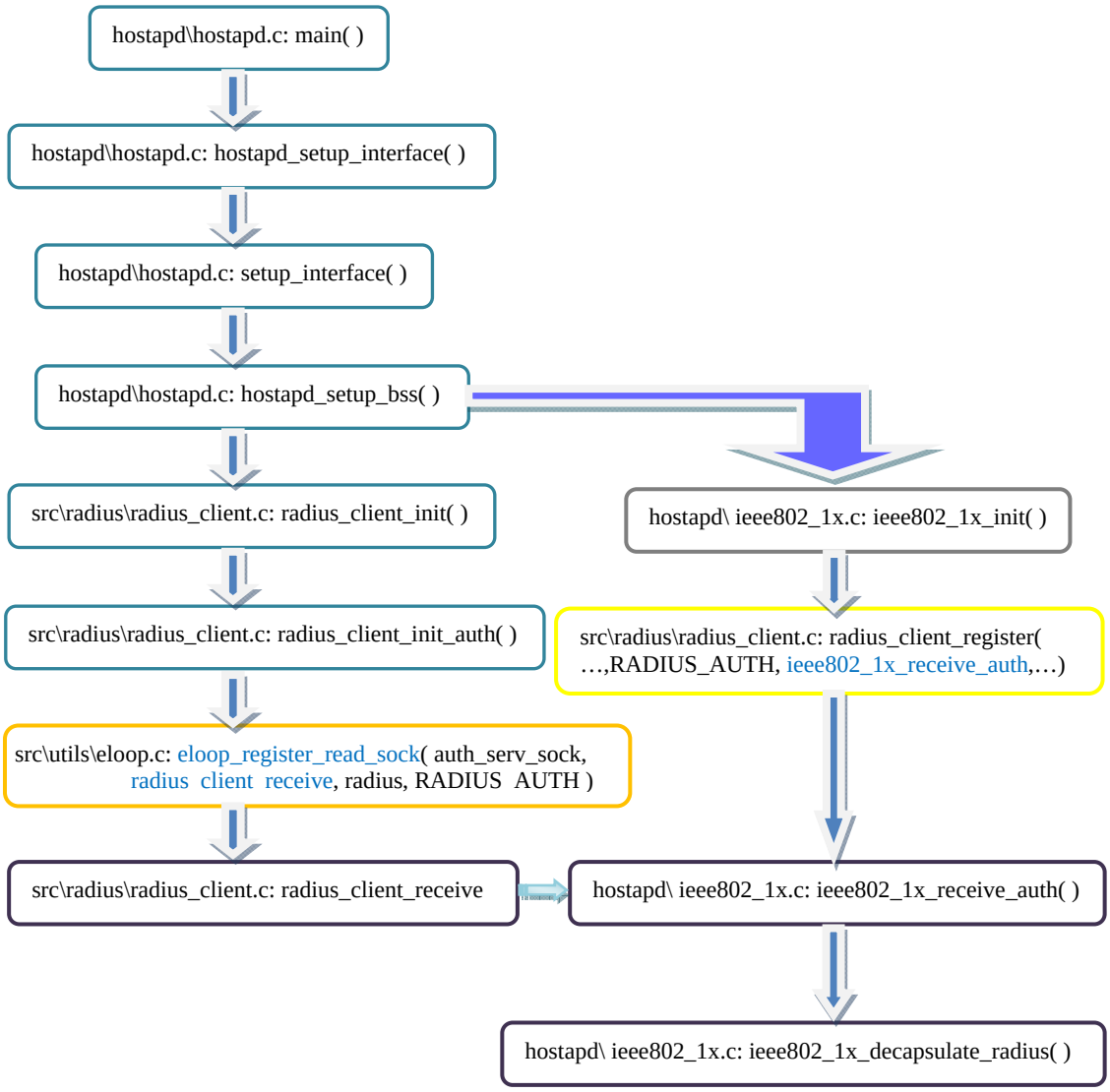

Figure 3-9. RADIUS Client on Receiving AAA Frames

The whole system starts at *main()*, which first reads in options. Then it registers the EAP methods, allocates space for each interface on demand and initializes an event loop. An AP may have several interfaces and **each interface may support up to 6 BSSes** [3] . Next each of the interfaces is initialized by calling *hostapd_setup_interface()*. The core function of *hostapd_setup_interface()* is *setup_interface()*. The latter function initializes the driver interface[4] and configures all BSSes with a pointer to this driver interface. Then it validates the BSS configurations, flushes old stations and sets up wireless link privacy for the driver. Next it sets the radio channel and frequency, clears default encryption keys (default management keys in the case of IEEE 802.11W), sets up link encryption, and sets the

[3] A single physical AP can act like multiple logical APs, each with a different BSS identifier. This enables mulitple operators to share an AP, while users logically connect via their network operator.
[4] Note that multiple BSSes can be configured with a pointer to a single driver interface.

beacon interval. After that it calls ieee802_11_set_beacon to prepare all the parameters that are needed in the beacon frame. Next the code sets a Request to Send (RTS) threshold and fragmentation threshold for the kernel driver. After configuring the driver, it continues setting up the BSS by calling *hostapd_setup_bss()*, which initializes all per-BSS data structures and resources. One of these resources, the RADIUS Client module, is initiated by calling *radius_client_init()*.

RADIUS Client has only two jobs: authentication and accounting. Thus *radius_client_init()* calls *radius_client_init_auth()* and *radius_client_init_acct()* respectively, which returns 0 if its initialization is successful. Both *radius_client_init_auth()* and *radius_client_init_acct()* calls *radius_change_server()* to prepare parameters needed for communication with the RADIUS server, like shared secret and retry counters. It also binds and connects RADIUS client address with RADIUS server address. The shared secret, retry counters as well as server sockets and addresses are written in a configuration file, which are read and saved in the structure *hostapd_radius_servers*. *radius_client_init_auth()* and *radius_client_init_acct()* are similar. Here we will focus on authentication. *radius_client_init_auth()* does three jobs: (1) Creates a socket; (2) Connects through the socket; and (3) Listens to this socket. For socket creation, it calls *socket(PF_INET, SOCK_DGRAM, 0)*. For connection, it calls *radius_change_server(radius, conf->auth_server, NULL, radius->auth_serv_sock, radius->auth_serv_sock6, 1*), which actually calls *connect(sel_sock, addr, addrlen)*. For socket listening, hostapd uses an event loop system. By registering a handler function for the auth socket, the system calls this handler function whenever a frame is received on that socket. Therefore, *eloop_register_read_sock* registers a handler *radius_client_receive* for auth_serv_sock.

The RADIUS Client has separate handlers and configurations for authentication and accounting messages. Thus *radius_client_receive* first examines the *msg_type* to prepare the corresponding configuration and handlers. Then it calls *recv(sock, buf, sizeof(buf), MSG_DONTWAIT)* and *radius_msg_parse(buf, len)*. The RADIUS message is stored in the structure *radius_msg*. While *radius_client_receive()* checks the message header code (ACCESS_ACCEPT, ACCESS_REJECT, ACCESS_CHALLENGE, or ACCOUNTING_RESPONSE) and increases the number of the corresponding message record by 1. As a RADIUS response should match a RADIUS request sent earlier by the RADIUS Client, a match means that their *identifier*s are equal. Thus the RADIUS Client puts each request in a list on sending a request and searches that list on receiving what looks like a response. If no matching RADIUS request is found, then that response message is dropped. If a match is found, then it calculates the round trip time and removes acknowledged RADIUS request from the list. Next, *radius_client_receive* goes through all the handlers, which deal with the RADIUS response.

The handler is stored in the data structure *radius_rx_handler*. This value is used *internally* inside the RADIUS client module. The structure is a combination of a handler function and context data (*void *data*). The context data is actually a pointer to the main data structure *hostapd*, which is mainly used for logging. The handler returns a return value *RadiusRxResult*, which is an enumeration, which indicates the result of processing by the handler function. The radius_rx_handler structure and the enumerated result values are shown below:

```
/**
 * src\radius\radius_client.c: struct radius_rx_handler - RADIUS client RX handler
 */
struct radius_rx_handler {
	/**
	 * handler - Received RADIUS message handler
	 */
	RadiusRxResult (*handler)(struct radius_msg *msg,
				  struct radius_msg *req,
				  const u8 *shared_secret,
				  size_t shared_secret_len,
				  void *data);
	/**
	 * data - Context data for the handler
	 */
	void *data;
};

/**
 * src\radius\radius_client.h: struct radius_rx_handler - RADIUS client RX handler
 */
typedef enum {
	RADIUS_RX_PROCESSED,
	RADIUS_RX_QUEUED,
	RADIUS_RX_UNKNOWN,
	RADIUS_RX_INVALID_AUTHENTICATOR
} RadiusRxResult
```

These call back handlers are registered by calls to *radius_client_register()* and unregistered when the RADIUS client is deinitialized with a call to *radius_client_deinit()*. There can be multiple registered RADIUS message handlers. Each of these handlers will be called in order until one of them indicates that it has processed or enqueued the message. As we can see from the code below, *radius_client_register()* first distinguishes if it is registering an auth or accounting handler. Then it finds the corresponding handler's array from the structure *radius_client_data*, which owns these handlers. By using realloc(), *radius_client_register()* gets enough additional space for the new handler (while keeping all the other handlers' information). Finally it adds the new handler at the end of the array.

1

Method

```
/**
 * src\radius\radius_client.c: radius_client_register - Register a RADIUS client RX handler
 * @radius: RADIUS client context from radius_client_init()
 * @msg_type: RADIUS client type (RADIUS_AUTH or RADIUS_ACCT)
 * @handler: Handler for received RADIUS messages
 * @data: Context pointer for handler callbacks
 * Returns: 0 on success, -1 on failure
 */
int radius_client_register(struct radius_client_data *radius,
                           RadiusType msg_type,
                           RadiusRxResult (*handler)(struct radius_msg *msg,
                                                     struct radius_msg *req,
                                                     const u8 *shared_secret,
                                                     size_t shared_secret_len,
                                                     void *data),
                           void *data)
{
        struct radius_rx_handler **handlers, *newh;
        size_t *num;

        if (msg_type == RADIUS_ACCT) {
                handlers = &radius->acct_handlers;
                num = &radius->num_acct_handlers;
        } else {
                handlers = &radius->auth_handlers;
                num = &radius->num_auth_handlers;
        }

        newh = os_realloc(*handlers,
                          (*num + 1) * sizeof(struct radius_rx_handler));
        if (newh == NULL)
                return -1;

        newh[*num].handler = handler;
        newh[*num].data = data;
        (*num)++;
        *handlers = newh;

        return 0;
}
```

There is only one authentication handler specifically registered for IEEE 802.1X, *ieee802_1x_receive_auth()*. The registration of *ieee802_1x_receive_auth()* comes during the initialization of 802.1X module. As can be seen in Figure 3-9 on page 75, both the RADIUS Client module and IEEE 802.1X module are initialized in *hostapd_setup_bss()*. However, the RADIUS Client module is initialized earlier than IEEE 802.1X module, as shown in the following code.

```
/**
 * hostapd\hostapd.c: hostapd_setup_bss - Per-BSS setup
 */
static int hostapd_setup_bss(struct hostapd_data *hapd, int first)
{
    ……
    ……
    if (wpa_debug_level == MSG_MSGDUMP)
        conf->radius->msg_dumps = 1;
    hapd->radius = radius_client_init(hapd, conf->radius);
    if (hapd->radius == NULL) {
        wpa_printf(MSG_ERROR, "RADIUS client initialization failed.");
        return -1;
    }

    if (hostapd_acl_init(hapd)) {
        wpa_printf(MSG_ERROR, "ACL initialization failed.");
        return -1;
    }
    if (hostapd_init_wps(hapd, conf))
        return -1;

    if (ieee802_1x_init(hapd)) {
        wpa_printf(MSG_ERROR, "IEEE 802.1X initialization failed.");
        return -1;
```

An investigation into *ieee802_1x_receive_auth()* as well as its sub function *ieee_802_1x_decapsulate_radius()* clearly shows how the interface variables from AAA to EAP work. These interface variables are colored blue in the following code.

```
/**
 *ieee802_1x.c: ieee802_1x_receive_auth - Process RADIUS frames from Authentication Server
 * @msg: RADIUS response message
 * @req: RADIUS request message
 * @shared_secret: RADIUS shared secret
 * @shared_secret_len: Length of shared_secret in octets
 * @data: Context data (struct hostapd_data *)
 * Returns: Processing status
 */
static RadiusRxResult
ieee802_1x_receive_auth(struct radius_msg *msg, struct radius_msg *req,
                        const u8 *shared_secret, size_t shared_secret_len,
                        void *data)
{
    struct hostapd_data *hapd = data;
    struct sta_info *sta;
    u32 session_timeout = 0, termination_action, acct_interim_interval;
    int session_timeout_set, old_vlanid = 0;
    struct eapol_state_machine *sm;
    int override_eapReq = 0;
```

1
Method

```
sm = ieee802_1x_search_radius_identifier(hapd, msg->hdr->identifier);
if (sm == NULL) {
        wpa_printf(MSG_DEBUG, "IEEE 802.1X: Could not find matching "
                   "station for this RADIUS message");
        return RADIUS_RX_UNKNOWN;
}
sta = sm->sta;

/* RFC 2869, Ch. 5.13: valid Message-Authenticator attribute MUST be
  * present when packet contains an EAP-Message attribute */
if (msg->hdr->code == RADIUS_CODE_ACCESS_REJECT &&
        radius_msg_get_attr(msg, RADIUS_ATTR_MESSAGE_AUTHENTICATOR, NULL,
                    0) < 0 &&
        radius_msg_get_attr(msg, RADIUS_ATTR_EAP_MESSAGE, NULL, 0) < 0) {
        wpa_printf(MSG_DEBUG, "Allowing RADIUS Access-Reject without "
                   "Message-Authenticator since it does not include "
                   "EAP-Message");
} else if (radius_msg_verify(msg, shared_secret, shared_secret_len,
                             req, 1)) {
        printf("Incoming RADIUS packet did not have correct "
                  "Message-Authenticator - dropped\n");
        return RADIUS_RX_INVALID_AUTHENTICATOR;
}

if (msg->hdr->code != RADIUS_CODE_ACCESS_ACCEPT &&
        msg->hdr->code != RADIUS_CODE_ACCESS_REJECT &&
        msg->hdr->code != RADIUS_CODE_ACCESS_CHALLENGE) {
        printf("Unknown RADIUS message code\n");
        return RADIUS_RX_UNKNOWN;
}

sm->radius_identifier = -1;
wpa_printf(MSG_DEBUG, "RADIUS packet matching with station " MACSTR,
           MAC2STR(sta->addr));

if (sm->last_recv_radius) {
        radius_msg_free(sm->last_recv_radius);
        os_free(sm->last_recv_radius);
}

sm->last_recv_radius = msg;

session_timeout_set =
        !radius_msg_get_attr_int32(msg, RADIUS_ATTR_SESSION_TIMEOUT,
                                   &session_timeout);
if (radius_msg_get_attr_int32(msg, RADIUS_ATTR_TERMINATION_ACTION,
                              &termination_action))
        termination_action = RADIUS_TERMINATION_ACTION_DEFAULT;

if (hapd->conf->radius->acct_interim_interval == 0 &&
        msg->hdr->code == RADIUS_CODE_ACCESS_ACCEPT &&
        radius_msg_get_attr_int32(msg, RADIUS_ATTR_ACCT_INTERIM_INTERVAL,
                                  &acct_interim_interval) == 0) {
        if (acct_interim_interval < 60) {
                hostapd_logger(hapd, sta->addr,
                               HOSTAPD_MODULE_IEEE8021X,
                               HOSTAPD_LEVEL_INFO,
                               "ignored too small "
```

2
Method

```
                                "Acct-Interim-Interval %d",
                                acct_interim_interval);
        } else
                sta->acct_interim_interval = acct_interim_interval;
    }


    switch (msg->hdr->code) {
    case RADIUS_CODE_ACCESS_ACCEPT:
        if (sta->ssid->dynamic_vlan == DYNAMIC_VLAN_DISABLED)
                sta->vlan_id = 0;
        else {
                old_vlanid = sta->vlan_id;
                sta->vlan_id = radius_msg_get_vlanid(msg);
        }
        if (sta->vlan_id > 0 &&
                hostapd_get_vlan_id_ifname(hapd->conf->vlan,
                                           sta->vlan_id)) {
                hostapd_logger(hapd, sta->addr,
                               HOSTAPD_MODULE_RADIUS,
                               HOSTAPD_LEVEL_INFO,
                               "VLAN ID %d", sta->vlan_id);
        } else if (sta->ssid->dynamic_vlan == DYNAMIC_VLAN_REQUIRED) {
                sta->eapol_sm->authFail = TRUE;
                hostapd_logger(hapd, sta->addr,
                               HOSTAPD_MODULE_IEEE8021X,
                               HOSTAPD_LEVEL_INFO, "authentication "
                               "server did not include required VLAN "
                               "ID in Access-Accept");
                break;
        }

        ap_sta_bind_vlan(hapd, sta, old_vlanid);

        /* RFC 3580, Ch. 3.17 */
        if (session_timeout_set && termination_action ==
                RADIUS_TERMINATION_ACTION_RADIUS_REQUEST) {
                sm->reAuthPeriod = session_timeout;
        } else if (session_timeout_set)
                ap_sta_session_timeout(hapd, sta, session_timeout);

        sm->eap_if->aaaSuccess = TRUE;
        override_eapReq = 1;
        ieee802_1x_get_keys(hapd, sta, msg, req, shared_secret,
                            shared_secret_len);
        ieee802_1x_store_radius_class(hapd, sta, msg);
        ieee802_1x_update_sta_identity(hapd, sta, msg);
        if (sm->eap_if->eapKeyAvailable &&
                wpa_auth_pmksa_add(sta->wpa_sm, sm->eapol_key_crypt,
                               session_timeout_set ?
                               (int) session_timeout : -1, sm) == 0) {
                hostapd_logger(hapd, sta->addr, HOSTAPD_MODULE_WPA,
                               HOSTAPD_LEVEL_DEBUG,
                               "Added PMKSA cache entry");
        }
        break;
    case RADIUS_CODE_ACCESS_REJECT:
        sm->eap_if->aaaFail = TRUE;
        override_eapReq = 1;
```

1

Method

```
                break;
        case RADIUS_CODE_ACCESS_CHALLENGE:
                sm->eap_if->aaaEapReq = TRUE;
                if (session_timeout_set) {
                        /* RFC 2869, Ch. 2.3.2; RFC 3580, Ch. 3.17 */
                        sm->eap_if->aaaMethodTimeout = session_timeout;
                        hostapd_logger(hapd, sm->addr,
                                       HOSTAPD_MODULE_IEEE8021X,
                                       HOSTAPD_LEVEL_DEBUG,
                                       "using EAP timeout of %d seconds (from "
                                       "RADIUS)",
                                       sm->eap_if->aaaMethodTimeout);
                } else {
                        /*
                         * Use dynamic retransmission behavior per EAP
                         * specification.
                         */
                        sm->eap_if->aaaMethodTimeout = 0;
                }
                break;
        }

        ieee802_1x_decapsulate_radius(hapd, sta);
        if (override_eapReq)
                sm->eap_if->aaaEapReq = FALSE;

        eapol_auth_step(sm);

        return RADIUS_RX_QUEUED;
}
/**
 *ieee802_1x.c: ieee802_1x_decapsulate_radius – Get EAP from RADIUS frames
*/
static void ieee802_1x_decapsulate_radius(struct hostapd_data *hapd,
                                          struct sta_info *sta)
{
        u8 *eap;
        size_t len;
        struct eap_hdr *hdr;
        int eap_type = -1;
        char buf[64];
        struct radius_msg *msg;
        struct eapol_state_machine *sm = sta->eapol_sm;

        if (sm == NULL || sm->last_recv_radius == NULL) {
                if (sm)
                        sm->eap_if->aaaEapNoReq = TRUE;
                return;
        }

        msg = sm->last_recv_radius;

        eap = radius_msg_get_eap(msg, &len);
        if (eap == NULL) {
                /* RFC 3579, Chap. 2.6.3:
                 * RADIUS server SHOULD NOT send Access-Reject/no EAP-Message
                 * attribute */
                hostapd_logger(hapd, sta->addr, HOSTAPD_MODULE_IEEE8021X,
                               HOSTAPD_LEVEL_WARNING, "could not extract "
```

```
                        "EAP-Message from RADIUS message");
        sm->eap_if->aaaEapNoReq = TRUE;
        return;
    }

    if (len < sizeof(*hdr)) {
        hostapd_logger(hapd, sta->addr, HOSTAPD_MODULE_IEEE8021X,
                       HOSTAPD_LEVEL_WARNING, "too short EAP packet "
                       "received from authentication server");
        os_free(eap);
        sm->eap_if->aaaEapNoReq = TRUE;
        return;
    }

    if (len > sizeof(*hdr))
        eap_type = eap[sizeof(*hdr)];

    hdr = (struct eap_hdr *) eap;
    switch (hdr->code) {
    case EAP_CODE_REQUEST:
        if (eap_type >= 0)
            sm->eap_type_authsrv = eap_type;
        os_snprintf(buf, sizeof(buf), "EAP-Request-%s (%d)",
                    eap_type >= 0 ? eap_type_text(eap_type) : "??",
                    eap_type);
        break;
    case EAP_CODE_RESPONSE:
        os_snprintf(buf, sizeof(buf), "EAP Response-%s (%d)",
                    eap_type >= 0 ? eap_type_text(eap_type) : "??",
                    eap_type);
        break;
    case EAP_CODE_SUCCESS:
        os_strlcpy(buf, "EAP Success", sizeof(buf));
        break;
    case EAP_CODE_FAILURE:
        os_strlcpy(buf, "EAP Failure", sizeof(buf));
        break;
    default:
        os_strlcpy(buf, "unknown EAP code", sizeof(buf));
        break;
    }
    buf[sizeof(buf) - 1] = '\0';
    hostapd_logger(hapd, sta->addr, HOSTAPD_MODULE_IEEE8021X,
                   HOSTAPD_LEVEL_DEBUG, "decapsulated EAP packet (code=%d "
                   "id=%d len=%d) from RADIUS server: %s",
                   hdr->code, hdr->identifier, be_to_host16(hdr->length),
                   buf);
    sm->eap_if->aaaEapReq = TRUE;

    wpabuf_free(sm->eap_if->aaaEapReqData);
    sm->eap_if->aaaEapReqData = wpabuf_alloc_ext_data(eap, len);
}
```

To better understand the code, it is necessary to clarify the relationship between some basic data structures. The data structures *eap_sm* (defined in \src\eap_server\eap_i.h) and *eap_eapol_interface* (defined in \src\eap_server\eap.h) have already been introduced. For each station there is an eap_sm and an eap_eapol_interface. In Section0 we will introduce *eapol_state_machine* (defined in

1
Method
\src\eapol_auth\eapol_auth_sm_i.h), whch holds ***per-supplicant*** authenticator state machines. This structure contains a pointer to eap_sm as well as a pointer to the eap_eapol_interface inside that eap_sm. For each supplicant, there is only one EAPOL state machine. Each station's information is put in a structure sta_info (see **Error! Reference source not found.**). This structure includes a pointer to its EAPOL state machine. All sta_info are put in a list, which is stored in the structure hostapd. Accordingly the structure hostapd contains all the information necessary for the operation of the daemon software.

The function *ieee802_1x_receive_auth()* first finds the EAPOL state machine matching the RADIUS response. Since sta_info contains the RADIUS message's identifier, *ieee802_1x_search_radius_identifier()* searches through the *sta_info* list in *hostapd* and returns the corresponding *eapol_state_machine*. Both sta_info and eapol_state_machine have a pointer referring to each other. Thus we can access the corresponding *sta_info* after we find the correct *eapol_state_machine*. Next the code will validate the RADIUS message by checking the message header as well as by matching this response with the corresponding request. After that, it updates the variable *last_recv_radius* stored in the *eapol_state_machine;* and sets session timeout and *acct_interim_interval* (The only difference between accounting and interim accounting messages is that the interim message will override any pending interim accounting updates; while a new accounting message does not remove any pending messages [49]). Next the code branches depending upon the header code: ACCESS_ACCEPT, ACCESS_CHALLENGE, or ACCESS_REJECT. You can see from the code that interface variables are set during this phase. For example, the setting of *aaaEapKeyData* can be seen in the function *ieee802_1x_get_keys*. A key requirement for implementing IEEE 802.1X is to extract the EAP message from the RADIUS frame. This is done by *ieee802_1x_decapsulate_radius()*. This function will also check if the EAP message is legal. If everything is OK, then the signal *aaaEapReq* is set to TRUE and the EAP message will be copied to *aaaEapReqData*. After all the relevant interface variables are set, *eapol_auth_step()* is called to revolve the EAPOL and EAP state machines (the EAP SM is also awakened by *eapol_auth_step()*).

It is important to note that the structure *eap_eapol_interface* which stores the interface variables resides in *eap_sm*. However the modification of the interface variables from the AAA layer to the EAP layer are made directly by *eapol_state_machine* rather than *eap_sm*. A pointer to *eap_eapol_interface* in *eapol_state_machine* helps renew the interface information while keeping the EAP module unattached.

3.2.2 RADIUS Client on Sending

Before giving the process flowchart, some critical data structure should be explained ahead. The structure *eapol_authenticator* is a global EAPOL authenticator data structure stored in the structure *hostapd*. It contains all the call back functions and configuration information needed by the authenticator. The references to the call back functions are put in the structure *eapol_auth_cb* (defined in eapol_auth_sm.h). The configuration information is stored in the structure *eapol_auth_config* (defined in eapol_auth_sm.h).

```
/**
  * src\eapol_auth\eapol_auth_sm_i.h: struct eapol_authenticator
  */
struct eapol_authenticator {
        struct eapol_auth_config conf;
        struct eapol_auth_cb cb;
```

```
    u8 *default_wep_key;
    u8 default_wep_key_idx;
};
```

Figure 3-10 shows the process flow map for the RADIUS client when sending RADIUS frames. The function *radius_client_send()* is used to transmit a RADIUS authentication (*RADIUS_AUTH*) or accounting (*RADIUS_ACCT* or *RADIUS_ACCT_INTERIM*) request. After some validation, it retrieves the *shared_secret* and calls *radius_msg_finish()* to add the attribute "Message-Authenticator" to the message, to set *msg->hdr->length,* and to encrypt the whole frame using the MD5 algorithm. After that it calls *send()* to transmit the frame to the authentication server. Note that the local socket is already connected with its counterpart in the authentication server due to the call to *radius_change_server()* during the *radius_client_init()* phase.

The function *radius_client_send()* is called by *ieee802_1x_encapsulate_radius(),* whose main job is to prepare an EAP message and itsert it into a RADIUS message. The first step is to get the user identity (from the EAP-Response/Identity) by calling *ieee802_1x_learn_identity()*. This user identity will later be used in the attribute *RADIUS_ATTR_USER_NAME* in the RADIUS frame. Next it gets the *radius_identifier* by calling *radius_client_get_id()* and creates a new RADIUS message by calling *radius_msg_new(RADIUS_CODE_ACCESS_REQUEST, sm->radius_identifier)*. Next it fills in the various attributes' values in the message by calling *radius_msg_add_attr()* for USER_NAME, NAS_IP_ADDRESS, NAS_IDENTIFIER, CALLED_STATION_ID, CALLING_STATION_ID, and CONNECT_INFO; as well as *radius_msg_add_attr_int32()* for NAS_PORT, FRAMED_MTU, and NAS_PORT_TYPE. Then it adds the resulting EAP message into the RADIUS frame by calling *radius_msg_add_eap()*. If this packet is an Access-Request reply to the previous Access-Challenge, then it must copy the STATE attribute from the previous RADIUS message. This is done by calling *radius_msg_copy_attr(msg, sm->last_recv_radius, RADIUS_ATTR_STATE)*. Now that the RADIUS frame is ready, it calls *radius_client_send()* – which was discussed in the previoous paragraph.

The function *ieee802_1x_encapsulate_radius()* is inside *ieee802_1x_aaa_send()*. During the initialization phase of IEEE 802.1X module, *ieee802_1x_init()* assigns a value to eapol_auth_config (*conf*) and eapol_auth_cb (*cb*). Then *ieee802_1x_aaa_send* serves as the *aaa_send* call back handler for eapol_auth_cb. Both eapol_auth_config and eapol_auth_cb are parameters to *eapol_auth_init()*, which initializes *hapd->eapol_auth*. Although the structure *eapol_authenticator* is part of the contents of the structure *hostapd*, it is also referenced through a pointer by the structure *eapol_state_machine*. That is how the functions in eapol_auth_cb serve as call back functions for EAPOL auth state machines.

Figure 3-10 shows the relationship between these methods, but it does not illustrate the whole process, like who actually calls the call back function *aaa_send()* and when it is called. Moreover it is also important to explore the interface variables from the EAP layer to the AAA layer. Thus it is necessary to examine the EAP state machine while at the same time looking into one of the EAPOL state machines (in this case the Backend Authentication state machine). The Backend Authentication

1
Method

state machine serves as a trigger for the EAP state machine by setting the appropriate interface variables. Figure 3-11 illustrates the relationship between EAP and EAPOL state machines as well as how those interface variables from the EAP layer to the AAA layer come into effect.

The following code shows how the Backend Authentication state machine transits from REQUEST to RESPONSE.

```
/**
 * eapol_sm.c
 */
SM_STEP(BE_AUTH)
{
        …
        switch (sm->be_auth_state) {
        …
        case BE_AUTH_REQUEST:
                if (sm->eapolEap)
                        SM_ENTER(BE_AUTH, RESPONSE);
                else if (sm->eap_if->eapReq)
                        SM_ENTER(BE_AUTH, REQUEST);
                else if (sm->eap_if->eapTimeout)
                        SM_ENTER(BE_AUTH, TIMEOUT);
                break;
        …
        }
}
```

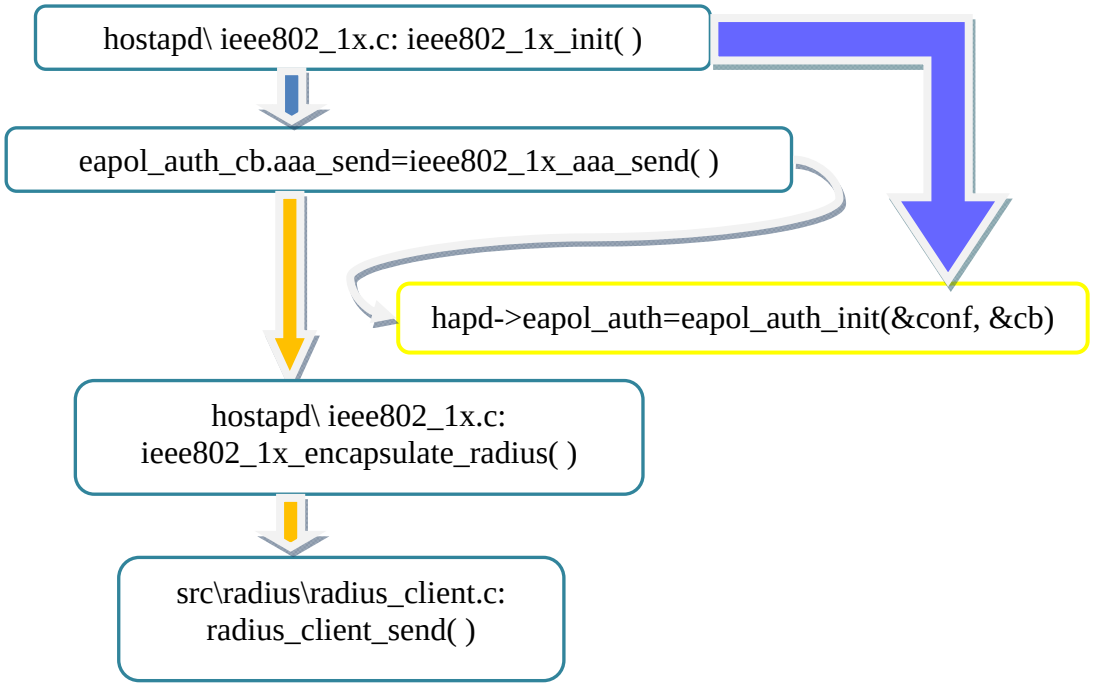

Figure 3-10. RADIUS Client on Sending AAA Frames

2

Method

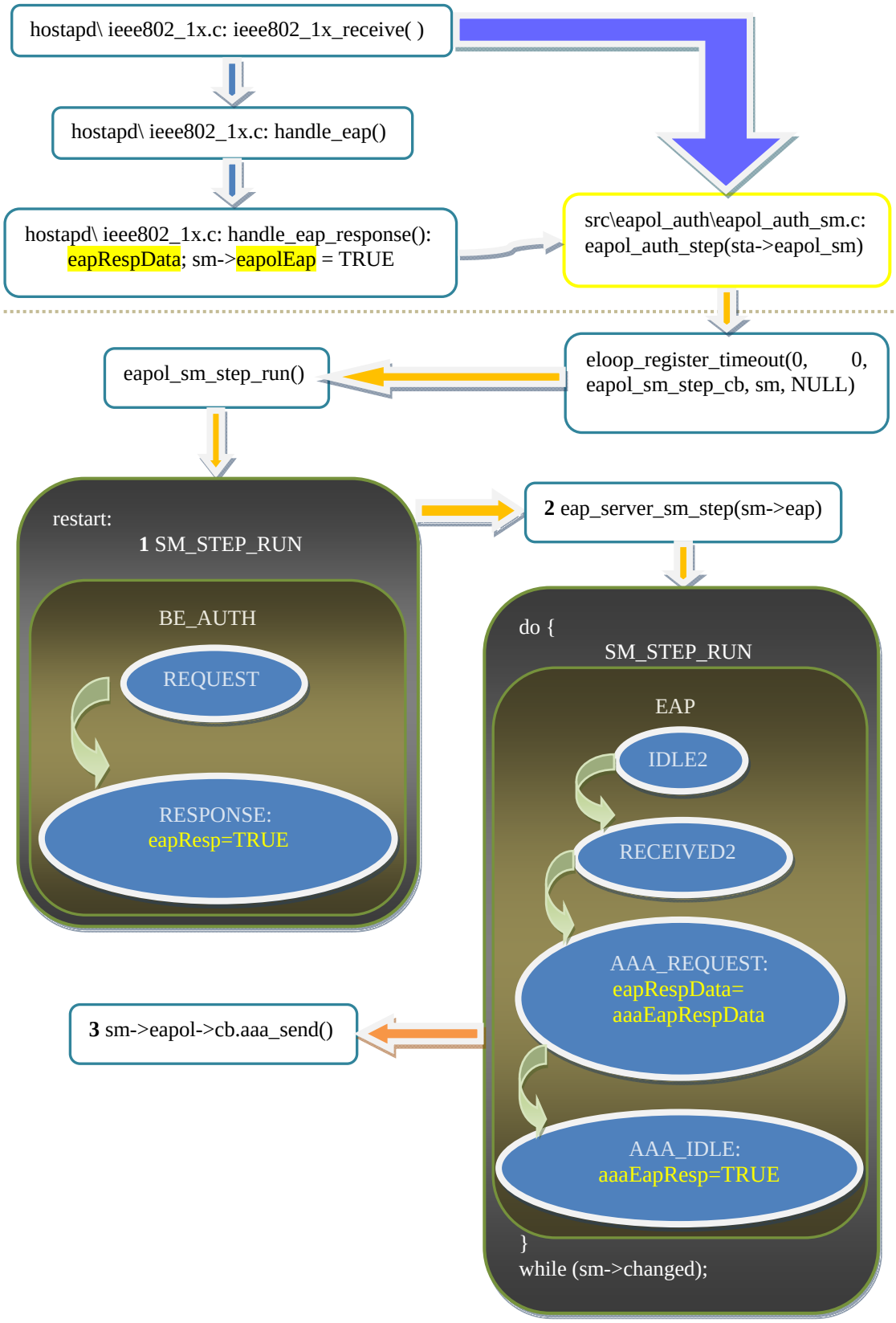

Figure 3-11. EAP and EAPOL SMs on Sending AAA Frames

1
Method

The function *ieee802_1x_receive()* processes the EAPOL frames from the supplicant. To process these frames *ieee802_1x_receive()* calls *handle_eap()* if the frame contains a 802.1X EAP packet. After this *handle_eap()* will call *handle_eap_response()* if the EAP code was RESPONSE (which we would expect in our scenario). The function *handle_eap_response()* finds the corresponding EAPOL state machine *sm* from *sta_info*, copies the EAP message to *sm->eap_if->eapRespData* and sets *sm->eapolEap* as *TRUE*. After that *ieee802_1x_receive()* calls *eapol_auth_step(sta->eapol_sm)* to step the EAPOL state machines forward.

The function *eapol_auth_step()* is called to advance all the EAPOL state machines after any change that could affect their state. Its core is actually *eapol_sm_step_run()*. Since a corresponding eapol_state_machine will exist as long as a supplciant is being serviced, and because the eapol_state_machine contains many state machines and methods whose work may be time consuming, plus there may be frequent state changes, *eapol_auth_step()* could take a lot of CPU cycles. Accordingly it is necessary to run *eapol_sm_step_run()* with a timeout in order to ensure that other potential timeouts and events are processed and to avoid long function call chains. The function *eapol_sm_step_cb()* simply serves as a package for *eapol_sm_step_run()* to get an eloop timeout registration.

The function *eapol_sm_step_run()* can be divided into three parts: (1) runs the EAPOL state machines; (2) runs the EAP state machine; (3) calls aaa_send(). The precondition for this function to march forward is that there is no more change to any of the state machines belonging to the current part. So the EAP state machine starts running when there is no further alteration of EAPOL state machines. However, if the state is changed after the EAP state machine runs, then it must go back to the EAPOL part again (Part 1 in Figure 3-11). When the EAP state machine is no longer changing, it advances to Part 3, using the call back function aaa_send(). At the beginning of Part 1, restart, it first keeps a record of all EAPOL state machines' status. Then it will run all the state machines with AUTH_PAE (the Authenticator PAE state machine) running first. Finally, it will compare all the state machines' status with the record at start. If there is any change, it will go back to restart to run those EAPOL state machines again. There is a countdown from 100 for "*goto restart;*" as a precaution against infinite loops inside the eloop callback. If 0 is reached, it will exit and return to restart through the event loop (*eapol_auth_step()*). The same process happens in Part 2.

In our scenario, one of the EAPOL state machines, Backend Authenticator state machine moves from REQUEST to RESPONSE, since *eapolEap* was set TRUE by *handle_eap_response()*. In the RESPONSE state, it sets *sm->eap_if->eapResp* as TRUE, which is a trigger for advancing the EAP state machine. Also it sets sm->eapolEap as FALSE, which is a trigger for itself. The function goes back to restart and finds there is no more change, then it moves to Part 2. Inside the function *eap_server_sm_step()* is a do while loop, so that the EAP state machine will keep running until there is no more change to any state machine. Then EAP steps from IDLE2 to RECEIVED2. In RECEIVED2 it calls *eap_sm_parse()* to get *respId*, *respMethod*, *respVendor* and *respVendorMethod*. *eap_sm_parse()* also sets *rxResp* TRUE if the header code is RESPONSE. rxResp is described in Table 3-4, it indicates that the current received packet is an EAP response. The EAP state machine runs again and this time it is in the state RECEIVED2. It finds out that *sm->rxResp* is

2

Method

TRUE (set just now) and *sm->respId* matches with *sm->currentId*, which means that the current EAP response belongs to the current EAP conversation. (***respId*** is the identifier from the current EAP response, see Table 3-4; ***currentId*** is the identifier of the currently outstanding EAP request; see Table 3-3). These two conditions make the EAP state machine transit to the state AAA_REQUEST, where it copies *eapRespData* to *aaaEapRespData*. Then it advances unconditionally to the AAA_IDLE state, where *sm->eap_if.aaaEapResp* is set TRUE. Till now, the EAP state machine has prepared *aaaEapResp* signal and *aaaEapRespData*. The function will go back to restart again and finds there isn't any change in either Part 1 or Part 2. So it marches to Part 3, where the call back *aaa_send* (*ieee802_1x_aaa_send()*, see Figure 3-10) is used to send the RADIUS message to the authentication server.

The following code shows the four states of EAP state machine (IDLE2, RECEIVED2, AAA_REQUEST, AAA_IDLE) as well as their transitions.

```
/**
 * src\eap.c
 */
SM_STATE(EAP, IDLE2)
{
	SM_ENTRY(EAP, IDLE2);

	sm->eap_if.retransWhile = eap_sm_calculateTimeout(
		sm, sm->retransCount, sm->eap_if.eapSRTT, sm->eap_if.eapRTTVAR,
		sm->methodTimeout);
}

SM_STATE(EAP, RECEIVED2)
{
	SM_ENTRY(EAP, RECEIVED2);

	/* parse rxResp, respId, respMethod */
	eap_sm_parseEapResp(sm, sm->eap_if.eapRespData);
}

SM_STATE(EAP, AAA_REQUEST)
{
	SM_ENTRY(EAP, AAA_REQUEST);

	if (sm->eap_if.eapRespData == NULL) {
		wpa_printf(MSG_INFO, "EAP: AAA_REQUEST - no eapRespData");
		return;
	}

	/*
	 * if (respMethod == IDENTITY)
	 *    aaaIdentity = eapRespData
	 * This is already taken care of by the EAP-Identity method which
	 * stores the identity into sm->identity.
	 */

	eap_copy_buf(&sm->eap_if.aaaEapRespData, sm->eap_if.eapRespData);
}
```

1
Method

```
SM_STATE(EAP, AAA_IDLE)
{
	SM_ENTRY(EAP, AAA_IDLE);

	sm->eap_if.aaaFail = FALSE;
	sm->eap_if.aaaSuccess = FALSE;
	sm->eap_if.aaaEapReq = FALSE;
	sm->eap_if.aaaEapNoReq = FALSE;
	sm->eap_if.aaaEapResp = TRUE;
}

SM_STEP(EAP)
{
	…
	case EAP_INITIALIZE_PASSTHROUGH:
		if (sm->currentId == -1)
			SM_ENTER(EAP, AAA_IDLE);
		else
			SM_ENTER(EAP, AAA_REQUEST);
		break;
	…
	case EAP_IDLE2:
		if (sm->eap_if.eapResp)
			SM_ENTER(EAP, RECEIVED2);
		else if (sm->eap_if.retransWhile == 0)
			SM_ENTER(EAP, RETRANSMIT2);
		break;
	…
	case EAP_RECEIVED2:
		if (sm->rxResp && (sm->respId == sm->currentId))
			SM_ENTER(EAP, AAA_REQUEST);
		else
			SM_ENTER(EAP, DISCARD2);
		break;
	…
	case EAP_AAA_REQUEST:
		SM_ENTER(EAP, AAA_IDLE);
		break;
	…
}
```

It is important to note that the modification to the interface variables from the EAP layer to the AAA layer are actually made directly by the EAP layer as a result of the direct interaction by the EAPOL layer on the EAP layer. The reference to *eap_eapol_interface* and *eapol_authenticator* in *eapol_state_machine* help the EAPOL layer invoke call back functions established by the RADIUS Client without touching the EAP module.

In the end of Section 3.2.1 it is mentioned that the modification to the interface variables from the AAA layer to the EAP layer were actually made by the direct interaction by the AAA layer on the EAPOL state machine. A pointer to the *eap_eapol_interface* from the *eapol_state_machine* to its true existence in *eap_sm* helps couple the interface information, while decoupling from the EAP module. As a result it does not matter the interaction is from the EAP layer to the AAA layer or the

reverse - “there is not necessarily a direct interaction between the EAP layer and the AAA layer, as in the case of 802.1X-2004” [51]. Such a design enables the EAP module and AAA module to intact while remaining loosely coupled. Figure 3-12 shows the relationship between EAP, EAPOL, and AAA during RADIUS receiving and sending phases.

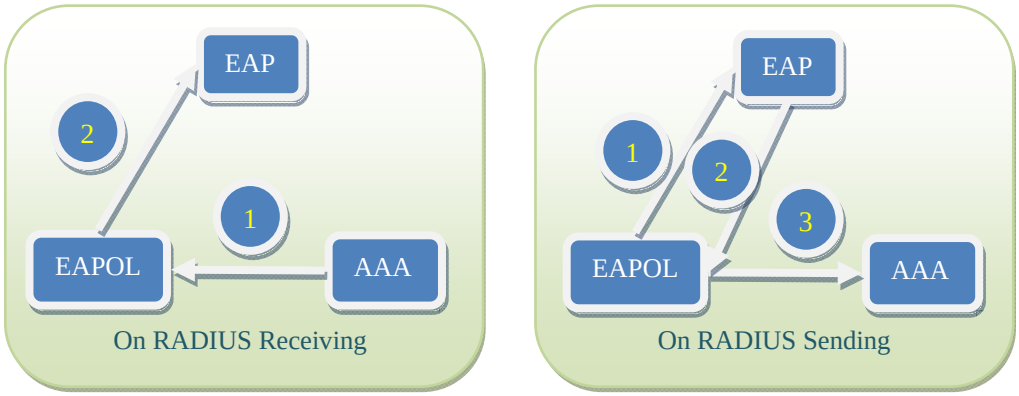

Figure 3-12. SM Relationship on RADIUS Sending & Receiving

3.3 EAPOL Layer

The EAPOL layer of the full authenticator, whether it is operating in pass-through mode or not, is the same as that of a stand-alone authenticator. Thus one can ignore the details of the AAA layer in this section. In the prior sections when studying the AAA interfaces, the focus was on sending and receiving RADIUS frames. With regard to EAPOL interfaces, the focus is sending and receiving EAPOL frames. Following the same pattern used in Section 3.2, we will introduce the interface variables between EAPOL and EAP first. Their descriptions are based on the explanation of EAP stand-alone authenticator in RFC 4137 [51].

Error! Reference source not found. shows the interface between the PACP state machines and the higher layer for the authenticator PAE. The system sends portEnabled signal to both the higher layer and the PACP, indicating that a port is active. The PACP transmits EAP messages between the physical port and the higher layer. The higher layer of the Authenticator uses eapReq/eapNoReq to disclose when it is prepared to receive a new message, and eapResp to hint that a new message is available to the higher layer. Inside the higher layer, EAP drives the authentication process together with the associated EAP methods. But on completion EAP will take its cue from AAA to signal eapSuccess or eapFail to the PACP. All EAP messages switched between Supplicant and Authenticator are produced by the EAP component.

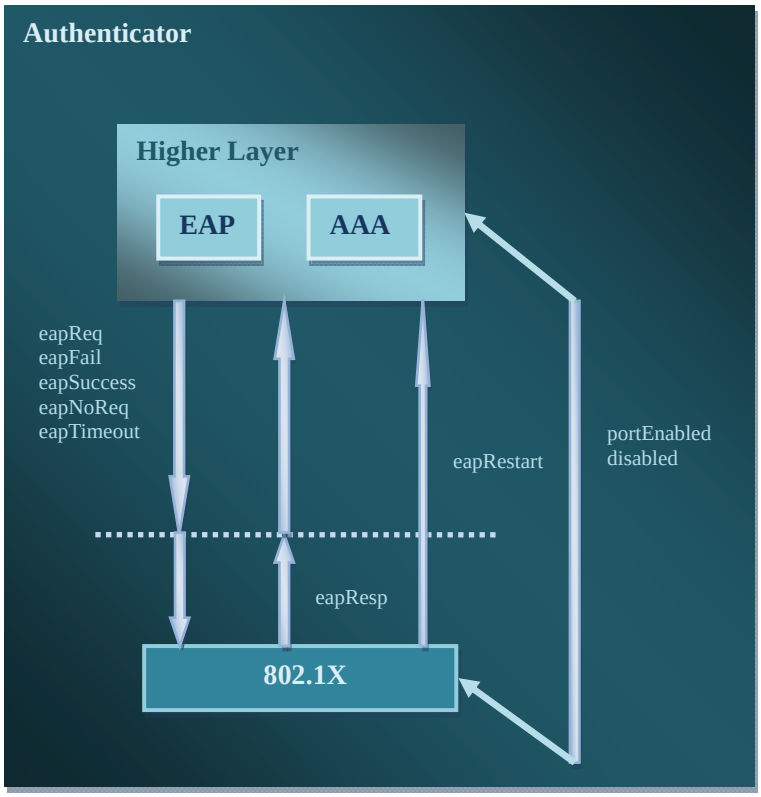

Figure 3-13. Interface between EAP & EAPOL SMs [33]

3.3.1 Variables

The variables used in the EAPOL Interface to the Authenticator are enumerated in Table 3-9. The variables used in the reverse direction are shown in Table 3-10.

2
Method

Table 3-9. Variables (EAPOL Interface to Full Authenticator)

Variable	**Description**
eapResp (boolean)	Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates that an EAP response is available for processing.
eapRespData (EAP packet)	Set in lower layer when eapResp is set to TRUE. The EAP packet to be processed.
portEnabled (boolean)	Indicates that the EAP authenticator state machine should be ready for communication. This is set to TRUE when the EAP conversation is started by the lower layer. If at any point the communication port or session is not available, portEnabled is set to FALSE, and the state machine transitions to DISABLED. To avoid unnecessary resets, the lower layer may dampen link down indications when it believes that the link is only temporarily down and that it will soon be back up (see [37] Section 7.12). In this case, portEnabled may not always be equal to the "link up" flag of the lower layer.
retransWhile (integer)	Outside timer used to indicate how long the authenticator has waited for a new (valid) response.
eapRestart (boolean)	Indicates that the lower layer would like to restart authentication.
eapSRTT (integer)	Smoothed round-trip time. (See [37] Section 4.3.)
eapRTTVAR (integer)	Round-trip time variation. (See [37] Section 4.3.)

Table 3-10. Variables (Full Authenticator Interface to EAPOL)

Variable	**Description**
eapNoReq (boolean)	Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the most recent response has been processed, but there is no new request to send.
eapSuccess (boolean)	Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates that the state machine has reached the SUCCESS state.
eapFail (boolean)	Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates that the state machine has reached the FAILURE state.
eapTimeout (boolean)	Set to TRUE in the TIMEOUT_FAILURE state if the authenticator has reached its maximum number of retransmissions without receiving a response.
eapReqData (EAP packet)	Set in authenticator state machine when eapReq, eapSuccess, or eapFail is set to TRUE. The actual EAP request to be sent (or success/failure).
eapKeyData (EAP key)	Set in authenticator state machine when keying material becomes available. Set during the METHOD state. Note that this document does not define the structure of the type "EAP key".
eapKeyAvailable (boolean)	Set to TRUE in the SUCCESS state if keying material is available. The actual key is stored in eapKeyData.

1
Method

3.3.2 How EAPOL functions in hostapd

The explanation for the EAPOL module in hostapd is divided into three parts: (1) Sending EAP Responses to the EAP layer; (2) Receiving EAP Requests from the EAP layer; and (3) EAPOL state machines. Part 1 has already been covered in Section 3.2.2. Part 2 is a succeeding process of RADIUS client receiving, which is covered in Section 3.2.1.

3.3.2.1 EAPOL on receiving an EAP Request

Error! Reference source not found. shows the remaining process after the AAA layer has transmitted the RADIUS response message to the EAP layer. When *aaaEapReq* is set to TRUE and *aaaEapReqData* assigned by *ieee802_1x_decapsulate_radius()*, then *eapol_auth_step()* at the end of *ieee802_1x_receive_auth()* will run the EAP state machine since there is no change in the EAPOL state machines. The status of the EAP state machine will transition from AAA_IDLE to AAA_RESPONSE, where the value of the interface variable aaaEapReqData will be copied to eapReqData. Then it will transition to SEND_REQUEST2, where it sets eapResp to FALSE and eapReq to TRUE. Both eapReq and eapReqData are interface variables from EAP to EAPOL (see Table 3-10). Next the EAP state machine will transition to IDLE2, where it sets *retransWhile* by calling *eap_sm_calculateTimeout()* and remains until a timeout. Note that *retransWhile* is an interface variable from EAPOL to EAP (see Table 3-9). The duration of *eap_sm_calculateTimeout()* determines how long the authenticator will wait for a valid response. When the EAP state machine is steady, then *eapol_auth_step()* returns to the restart state and tries to run EAPOL state machines. However, one machine has changed this time. This is the Backend Authenticator state machine, which is in charge of transmitting messages between EAPOL and EAP. The Backend Authenticator state machine will enter the REQUEST state from one of the following states: IGNORE, IDLE, or RESPONSE. In the REQUEST state, it will send an EAP Request frame by calling *txReq()* and set *eapReq* to FALSE. Note that retransWhile is controlled by *eapol_port_timers_tick()*. The following code expounds the above process of Backend Authenticator state machine.

```
/**
 * eapol_sm.c: struct eapol_authenticator
 */
SM_STEP(BE_AUTH)
{
	if (sm->portControl != Auto || sm->initialize || sm->authAbort) {
		SM_ENTER_GLOBAL(BE_AUTH, INITIALIZE);
		return;
	}

	switch (sm->be_auth_state) {
	case BE_AUTH_INITIALIZE:
		SM_ENTER(BE_AUTH, IDLE);
		break;
	case BE_AUTH_REQUEST:
		if (sm->eapolEap)
			SM_ENTER(BE_AUTH, RESPONSE);
		else if (sm->eap_if->eapReq)
			SM_ENTER(BE_AUTH, REQUEST);
		else if (sm->eap_if->eapTimeout)
```

```
                SM_ENTER(BE_AUTH, TIMEOUT);
            break;
        case BE_AUTH_RESPONSE:
            if (sm->eap_if->eapNoReq)
                SM_ENTER(BE_AUTH, IGNORE);
            if (sm->eap_if->eapReq) {
                sm->backendAccessChallenges++;
                SM_ENTER(BE_AUTH, REQUEST);
            } else if (sm->aWhile == 0)
                SM_ENTER(BE_AUTH, TIMEOUT);
            else if (sm->eap_if->eapFail) {
                sm->backendAuthFails++;
                SM_ENTER(BE_AUTH, FAIL);
            } else if (sm->eap_if->eapSuccess) {
                sm->backendAuthSuccesses++;
                SM_ENTER(BE_AUTH, SUCCESS);
            }
            break;
        case BE_AUTH_SUCCESS:
            SM_ENTER(BE_AUTH, IDLE);
            break;
        case BE_AUTH_FAIL:
            SM_ENTER(BE_AUTH, IDLE);
            break;
        case BE_AUTH_TIMEOUT:
            SM_ENTER(BE_AUTH, IDLE);
            break;
        case BE_AUTH_IDLE:
            if (sm->eap_if->eapFail && sm->authStart)
                SM_ENTER(BE_AUTH, FAIL);
            else if (sm->eap_if->eapReq && sm->authStart)
                SM_ENTER(BE_AUTH, REQUEST);
            else if (sm->eap_if->eapSuccess && sm->authStart)
                SM_ENTER(BE_AUTH, SUCCESS);
            break;
        case BE_AUTH_IGNORE:
            if (sm->eapolEap)
                SM_ENTER(BE_AUTH, RESPONSE);
            else if (sm->eap_if->eapReq)
                SM_ENTER(BE_AUTH, REQUEST);
            else if (sm->eap_if->eapTimeout)
                SM_ENTER(BE_AUTH, TIMEOUT);
            break;
        }
}

SM_STATE(BE_AUTH, REQUEST)
{
        SM_ENTRY_MA(BE_AUTH, REQUEST, be_auth);

        txReq();
        sm->eap_if->eapReq = FALSE;
        sm->backendOtherRequestsToSupplicant++;

        /*
         * Clearing eapolEap here is not specified in IEEE Std 802.1X-2004, but
         * it looks like this would be logical thing to do there since the old
         * EAP response would not be valid anymore after the new EAP request
         * was sent out.
```

1
Method

```
		 *
		 * A race condition has been reported, in which hostapd ended up
		 * sending out EAP-Response/Identity as a response to the first
		 * EAP-Request from the main EAP method. This can be avoided by
		 * clearing eapolEap here.
		 */
		sm->eapolEap = FALSE;
}

SM_STATE(BE_AUTH, RESPONSE)
{
	SM_ENTRY_MA(BE_AUTH, RESPONSE, be_auth);

	sm->authTimeout = FALSE;
	sm->eapolEap = FALSE;
	sm->eap_if->eapNoReq = FALSE;
	sm->aWhile = sm->serverTimeout;
	sm->eap_if->eapResp = TRUE;
	/* sendRespToServer(); */
	sm->backendResponses++;
}

SM_STATE(BE_AUTH, IDLE)
{
	SM_ENTRY_MA(BE_AUTH, IDLE, be_auth);

	sm->authStart = FALSE;
}


SM_STATE(BE_AUTH, IGNORE)
{
	SM_ENTRY_MA(BE_AUTH, IGNORE, be_auth);

	sm->eap_if->eapNoReq = FALSE;
}
```

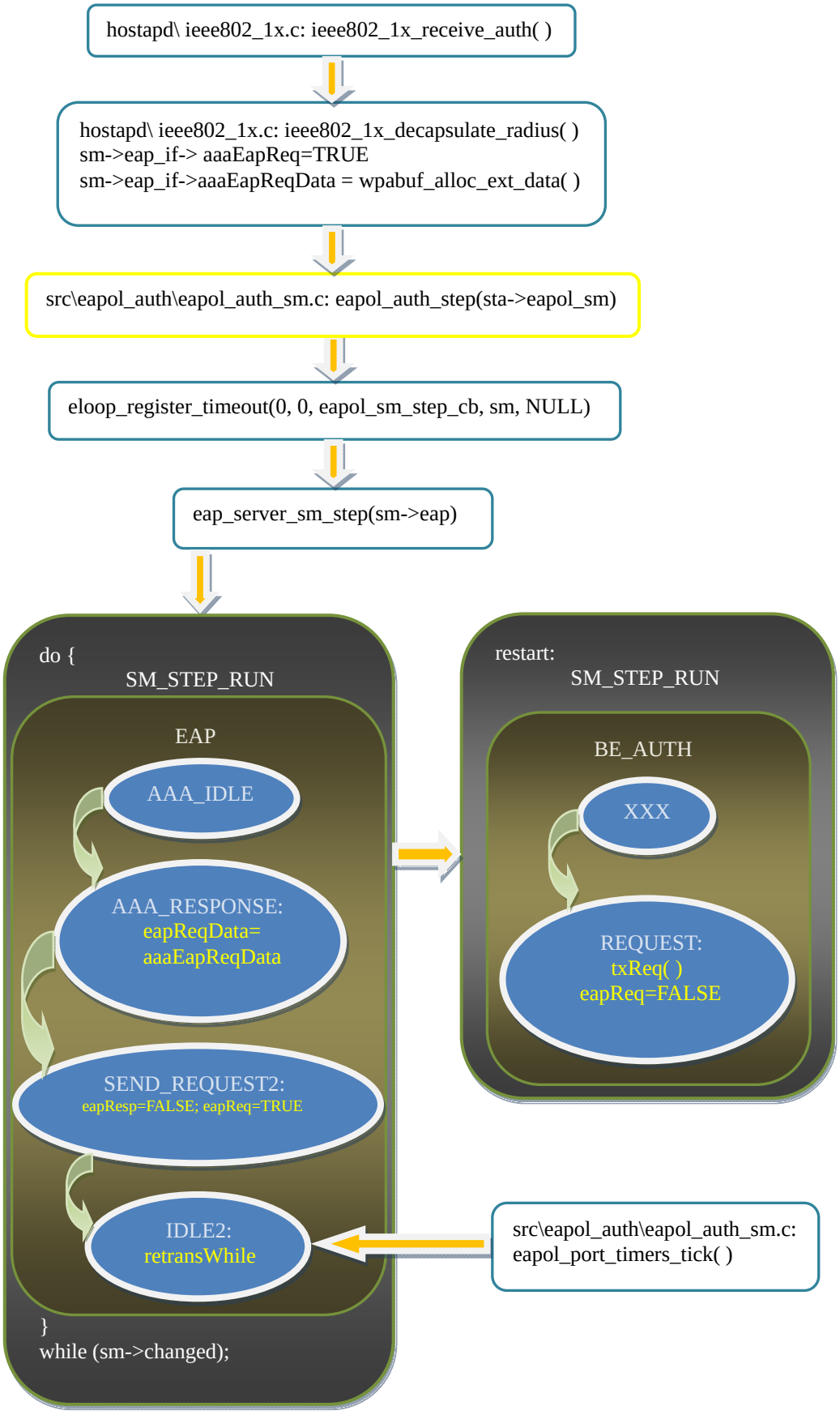

Figure 3-14. EAP & EAPOL SMs on Transmitting EAP Requests

1
Method

3.3.2.2 EAPOL State Machines

Most of the information in this section is based upon IEEE 802.1x Section 8.2. You can refer to the standard to dig deeper in specific state machines. Appendix C gives the definition of eapol_state_machine in hostapd. We will modify this definition later to implement the non-binary authenticator.

There are 10 different state machines. Table 3-11 lists them out and summarizes the requirements of them for implementations that support Authenticator, Supplicant, or both. An X mark means it is necessary while an O mark means it is optional.

Table 3-11. State Machine Support Requirements [33]

State Machine	Authenticator	Supplicant	Both
Port Timers state machine	X	X	X
Authenticator PAE state machine	X		X
Authenticator Key Transmit state machine	O		O
Supplicant Key Transmit state machine		O	O
Reauthentication Timer state machine	X		X
Backend Authentication state machine	X		X
Controlled Direction state machine	X		X
Supplicant PAE state machine		X	X
Supplicant Backend state machine		X	X
Key Receive state machine	X	X	X

The rest of this section will focus on the state machines which is necessary for Authenticator. For each state machine it gives a brief introduction plus the state diagram. For timers and global variables, please refer to IEEE 802.1X Section 8.2.2. For local variables, constants and procedures of each state machine, you can refer to the corresponding state machine's explanation in IEEE 802.1X Section 8.2.

The Port Timers state machine for a given Port is responsible for decrementing the timer variables for that Port each second, in response to an external system clock function. The timer variables, like retransWhile, are used and set to their initial values by the operation of the individual state machines for the Port.

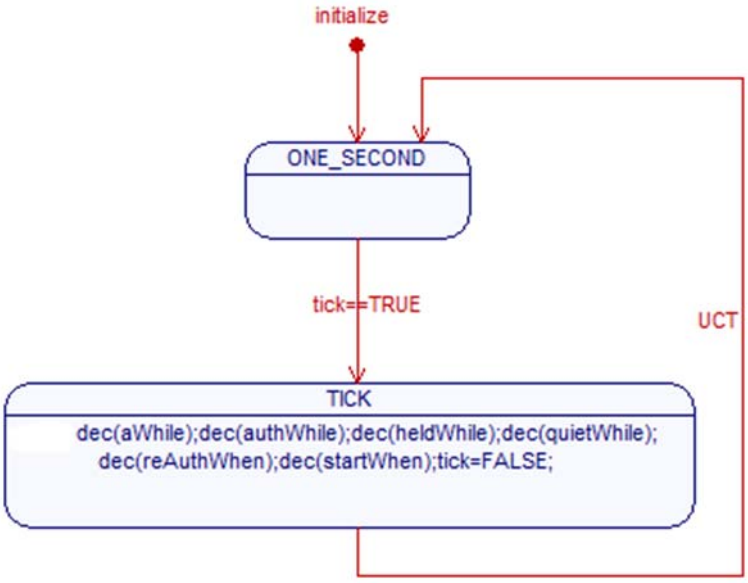

Figure 3-15. Port Timers State Machine [33]

The Authenticator PAE state machine represents the authenticating status of the corresponding supplicant, revealing the status of controlled port. It has two separate states, FORCE_AUTH and FORCE_UNAUTH, which are described in Figure 3-16 and Figure 3-17. The rest of the state machine are described in Figure 3-18.

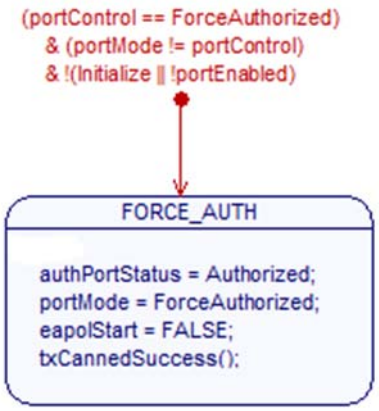

Figure 3-16. Authenticator PAE State Machine - 1 [33]

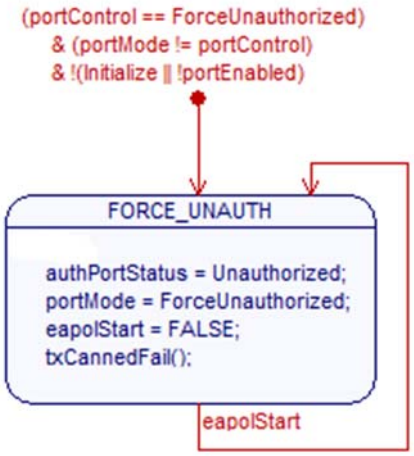

Figure 3-17. Authenticator PAE State Machine - 2 [33]

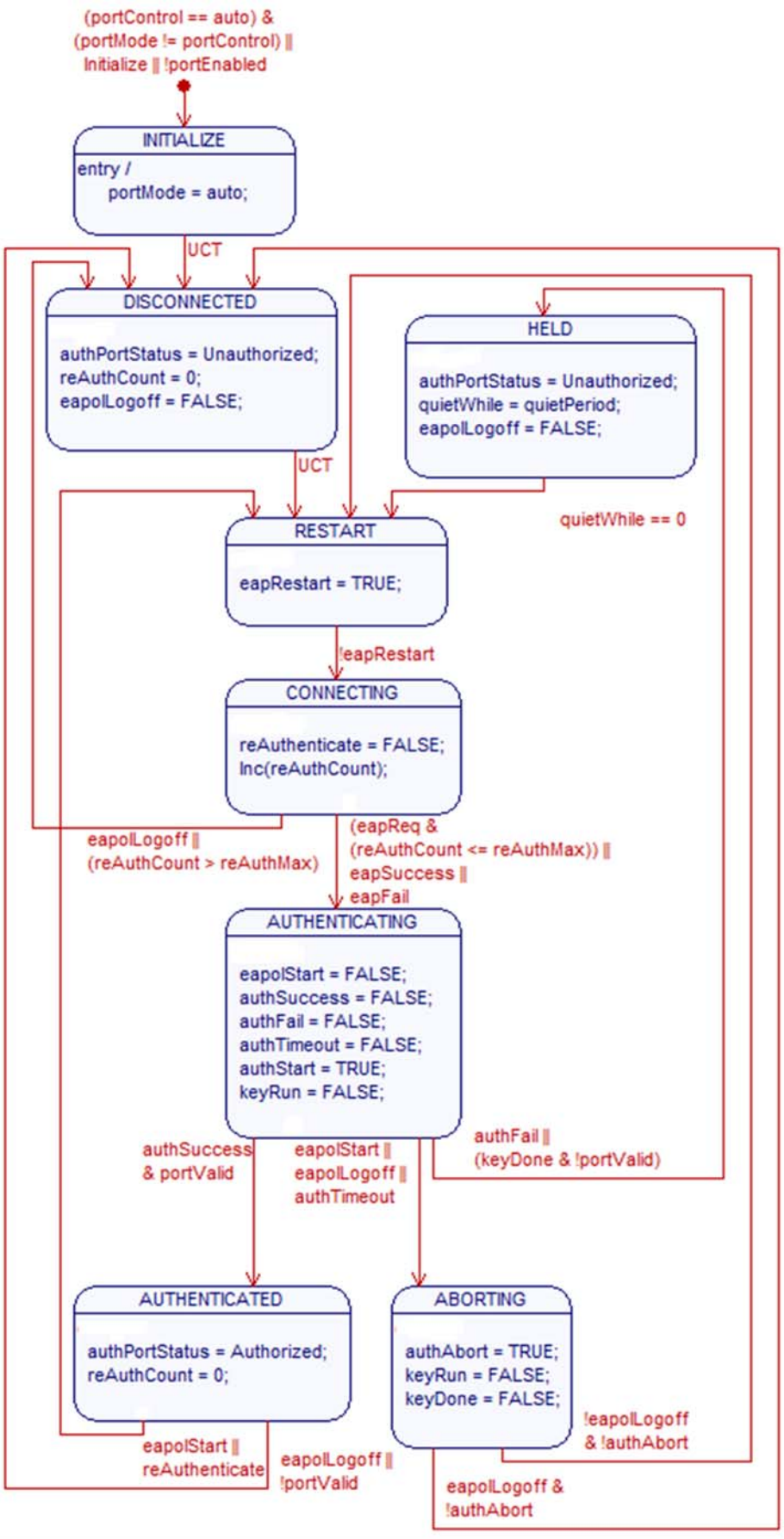

Figure 3-18. Authenticator PAE State Machine - 3 [33]

1

Method

The Key Receive state machine allows EAPOL-Keys PDUs to be received from the Supplicant or Authenticator and processed in accordance with any encryption mechanisms being employed by the Authenticator or Supplicant.

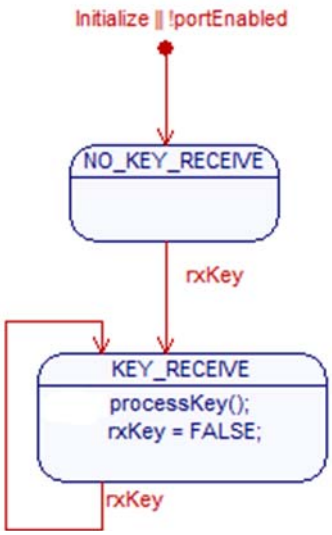

Figure 3-19. Key Receive State Machine [33]

The Reauthentication Timer state machine for a given Port is responsible for ensuring that periodic reauthentication of the Supplicant takes place, if periodic reauthentication is enabled (reAuthEnabled is TRUE). The state machine is held in the INITIALIZE state until such a time as the portControl for the Port is Auto, the portStatus for the Port becomes Authorized, the port is not being initialized, and thereAuthEnabled control is TRUE. The reAuthWhen timer is set to its initial value; when it expires, the state machine will then transition to the REAUTHENTICATE state, setting the reAuthenticate variable TRUE, and then transitioning back to INITIALIZE for a further timer cycle. It is important to note that the Authenticator PAE state machine stays in AUTEHTICATED status during the reauthentication period until the reauthentication fails.

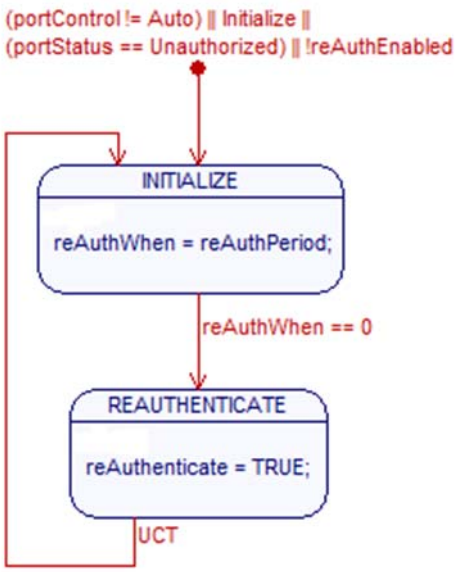

Figure 3-16. Reauthentication Timer State Machine [33]

The Backend Authentication state machine, which represents the authentication process, is responsible for transmitting EAP messages between EAPOL and EAP. Its use can be found in Section 3.2. Distinguish this Backend Authentication state machine from the EAP Backend Authenticator state machine in Figure 3-6.

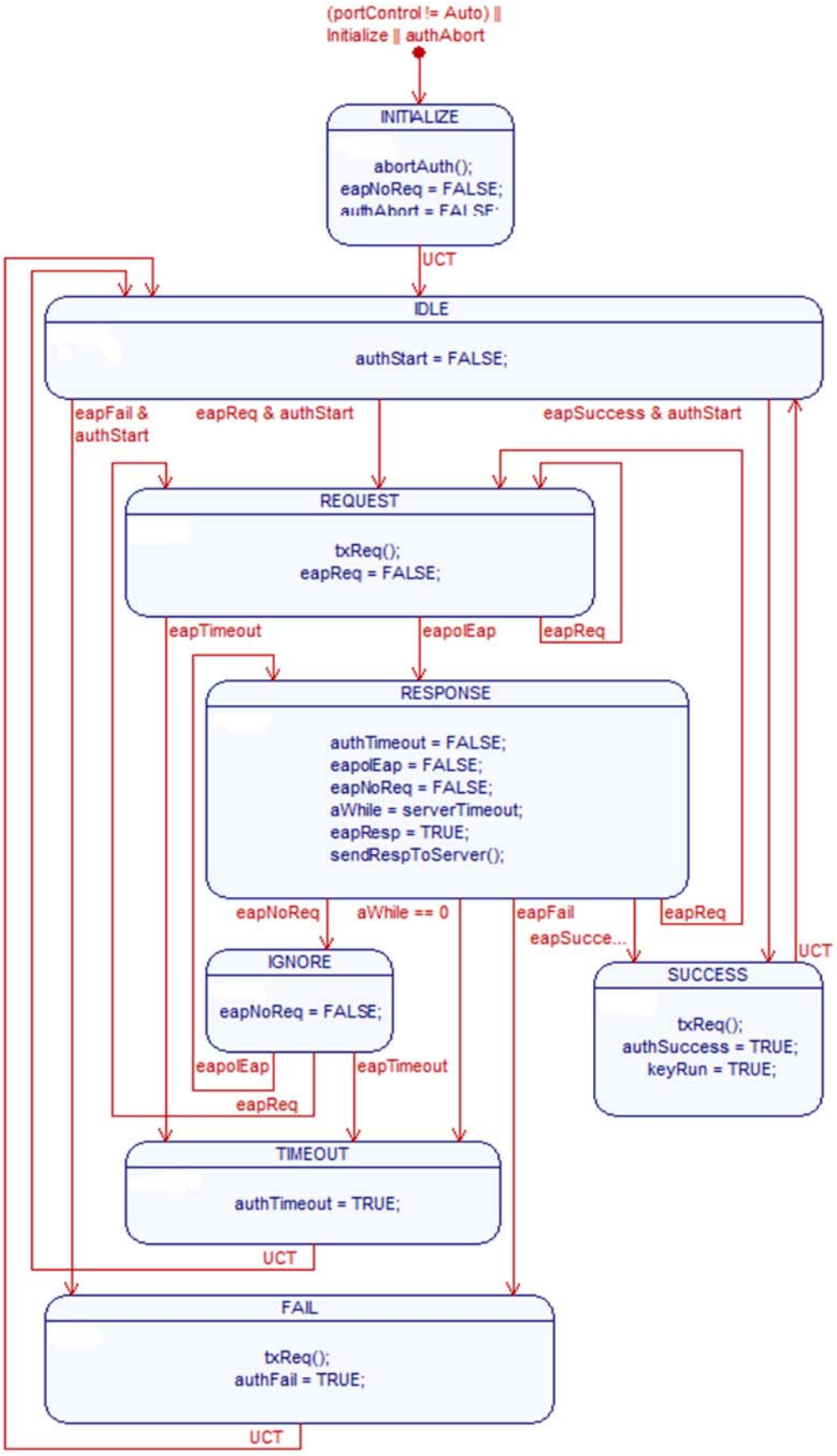

Figure 3-17. Backend Authentication State Machine [33]

2

Method

The Controlled Directions state machine for a given Port is responsible for ensuring that the value of the OperControlledDirections parameter for the Port correctly reflects the current state of the AdminControlledDirections parameter coupled with the operational state of the MAC and the presence or absence of a Bridge (see IEEE 802.1X Section 6.5)

If OperControlledDirections is set to IN on a Bridge Port, this allows the Bridge to forward frames received from its other Bridge Ports onto that Port, but prevents frames received on that Port (including BPDUs) from being processed or forwarded by the Bridge. In order to prevent the possibility of configuring inadvertent loops as a result of connecting a Bridge to a Bridge Port that is set to IN, OperControlledDirections is forced to BOTH if the operEdge variable (see Clause 17 of IEEE Std 802.1D) for the Port is FALSE.

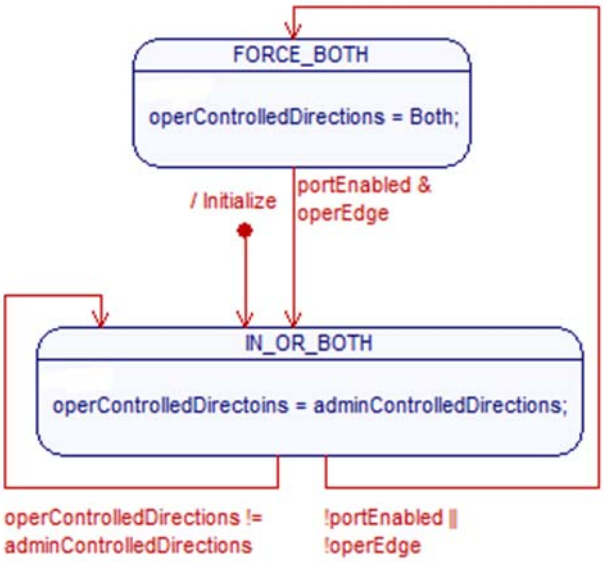

Figure 3-18. Controlled Directions State Machine [33]

3.4 EAPOL Sender & Receiver

This part explains the EAPOL sending and receiving process in hostapd. The EAPOL sending process is actually a succeeding process of EAPOL receiving EAP request (see Section 3.3.2.1). And the EAPOL receiving process is a **pre-action** of EAPOL sending EAP response (see Section 3.2.2).

Figure 3-21 shows how hostapd receives EAPOL frames in a top-down manner. Structure *wpa_driver_ops* is defined in src\drivers\driver.h. It defines the API that each driver interface needs to implement. Through this wrapper, callback functions are called by hostapd for requesting driver operations. All driver specific functionality is captured in this wrapper.

In the fold “drivers”, each file named “driver_xxx” maps to one type of driver. Each file also defines the callback functions according to its driver’s character. In the end of the file, it will define the wrapper for that driver in the structure wpa_driver_ops with name wpa_driver_xxx_ops. In our case, we are using hostap

driver, thus the interface is named wpa_driver_hostap_ops. Of course, only hostap and common parts in structure wpa_driver_ops are assigned value in our case.

Among various driver operations, one is *hostap_init()*. It first allocates memory for hostapd driver data. Next it tries to open ioctl_sock, which is used for system communication. Thirdly it enables hostapd mode for that interface. Finally it calls *hostap_init_sockets()* or *hostap_wireless_event_init()* if *hostap_init_sockets()* failed.

hostap_init_sockets() opens raw packet socket and registers call back handler *handle_read()* for that socket. *handle_read()* does only two things: receive packets from the socket by calling *recv()* and calls *handle_frame()* to deal with the frame stored in the buf. *handle_frame()* checks the frame type and sub type as well as frame length. After validation, it checks the protocol version. Protocol version 3 indicates extra data after the payload, version 2 indicates an ACKed frame (TX callbacks), and version 1 indicates a failed frame (no ACK, TX callbacks). If the protocol version is not 1, 2 or 3, then the function simply reports the error and returns. If the version number is 1 or 2, it calls *handle_tx_callback()* to deal with the frame. But if protocol version is 3, it will first try to get the complete frame and then check the frame type. If it is a management frame, *wpa_supplicant_event()* will be called. If it is a control frame, nothing is done. If it is a data frame, *handle_data()* will be called. *wpa_supplicant_event()* plays a core role in hostapd wireless receiving. It is not only called by *handle_frame()*, but also by *handle_tx_callback()* and *handle_data()*(see Figure 3-23).

wpa_supplicant_event() (defined in drv_callbacks.c) reports a driver event for wpa_supplicant. The driver wrapper, *wpa_driver_hostap_ops* (illustrated in Figure 3-24) calls this function whenever an event is received from the driver. All the events are abstracted into *wpa_event_data*, which is a union defined in driver.h. These events are also categorized and their types are enumerated in *enum wpa_event_type* in driver.h. *wpa_supplicant_event()* handles the event according to the event type.

In *handle_frame()*, after the whole frame is collected, if it is a management frame, it will be put into *event.rx_mgmt.frame* and a call will be made to *wpa_supplicant_event(drv->hapd, EVENT_RX_MGMT, &event)*. The function *wpa_supplicant_event()* calls *hostapd_mgmt_rx()* to process the event.

In *handle_tx_callback()*, it is already clear if it is a failed or ACKed frame. Thus this function assigns an appropriate value to event.tx_status.xxx, and calls *wpa_supplicant_event(drv->hapd, EVENT_TX_STATUS, &event)* directly.

In *handle_frame()*, when it is a data frame, the function calls *handle_data()*. Since it is already known to be a 802.11 data frame, the function will check the frame's header first and then processes the contents of the frame. It puts the 802.11 header in event.rx_from_unknown.frame and calls *wpa_supplicant_event(drv->hapd, EVENT_RX_FROM_UNKNOWN, &event)*. The function *wpa_supplicant_event()* calls hostapd_rx_from_unknown_sta(), which will return if it finds that this message is not destined for any of the BSSes in this AP. If it is destined to a BSS on this AP, then *hostapd_rx_from_unknown_sta()* calls *ieee802_11_rx_from_unknown()* to check if the sender is associated or not and disassociates or disauthenticates with it accordingly. If everything is ok, then *handle_frame()* will move the pointer from the end of header to the start of body. It calls *drv_event_eapol_rx()* to deal with an ethertype packet, which places the Ethernet frame in *event.eapol_rx.xxx* and calls

wpa_supplicant_event(ctx, EVENT_EAPOL_RX, &event). For *EVENT_EAPOL_RX*, *hostapd_event_eapol_rx* is called, which locates the corresponding BSS for the inter face and calls *ieee802_1x_receive()*.

Each incoming EAPOL frame from the wireless interface is processed by *ieee802_1x_receive(struct hostapd_data *hapd, const u8 *sa, const u8 *buf, size_t len)*. The structure *hostapd_data* is a giant structure containing all the information about a BSS. The argument *sa* is the source address (supplicant) of the EAPOL frame. While *buf* contains the EAPOL frame, *len* represents the length of *buf* in octets. Each supplicant's information is stored in a structure *sta_info*. This structure is initialized and assigned during the search phase whenever a supplicant's existence is sensed by the AP. By calling *ap_get_sta(hapd, sa)*, the AP learns which supplicant the EAPOL frame belongs to. Next it will check if the station information is available, whether the whole 802.1X packet is too short, or if the frame (without the IEEE 802.1X header) is too short. After verification, it will update the frame version, WPA or RSN key, and increment the total number of EAPOL frames received. If there is no EAPOL state machine for the supplicant yet (*!sta->eapol_sm*), one will be initiated.

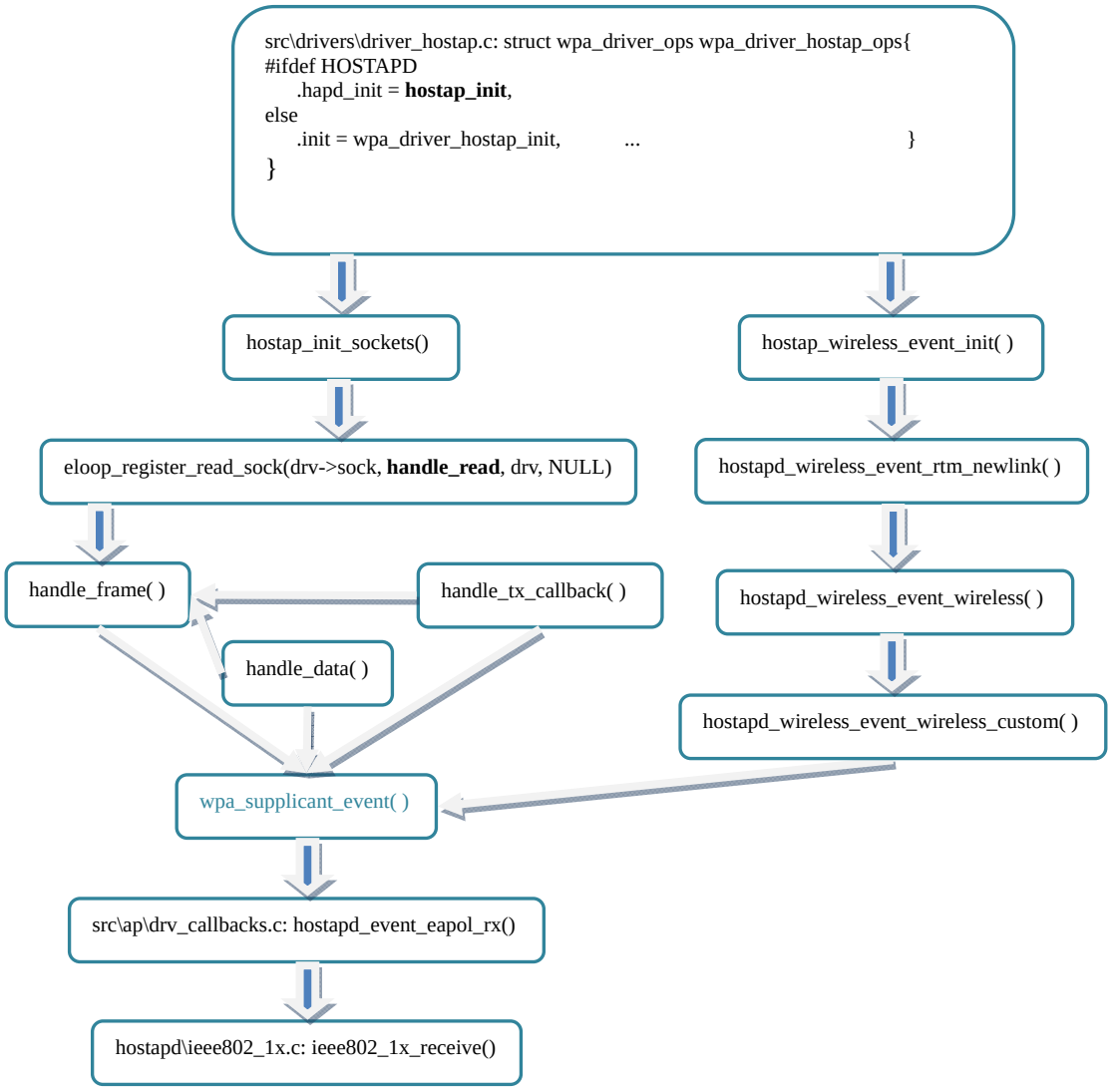

Figure 3-19. EAPOL Receiver

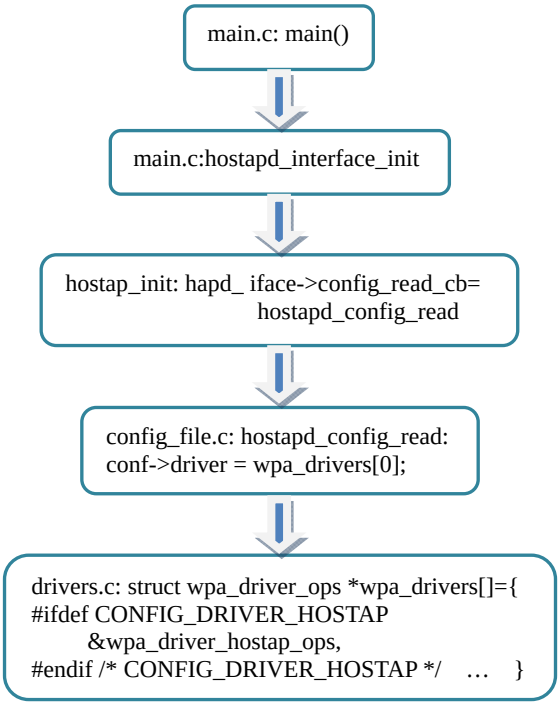

Figure 3-20. wpa_driver_hostap_ops for Initialization

The EAPOL sending process is activated when the Backend Authentication state machine transits to REQUEST and calls txReq(). Figure 3-25 shows in a top-down manner how hostapd sends EAPOL frames. An explanation is omitted since the figure provides sufficient information for our purposes.

2
Method

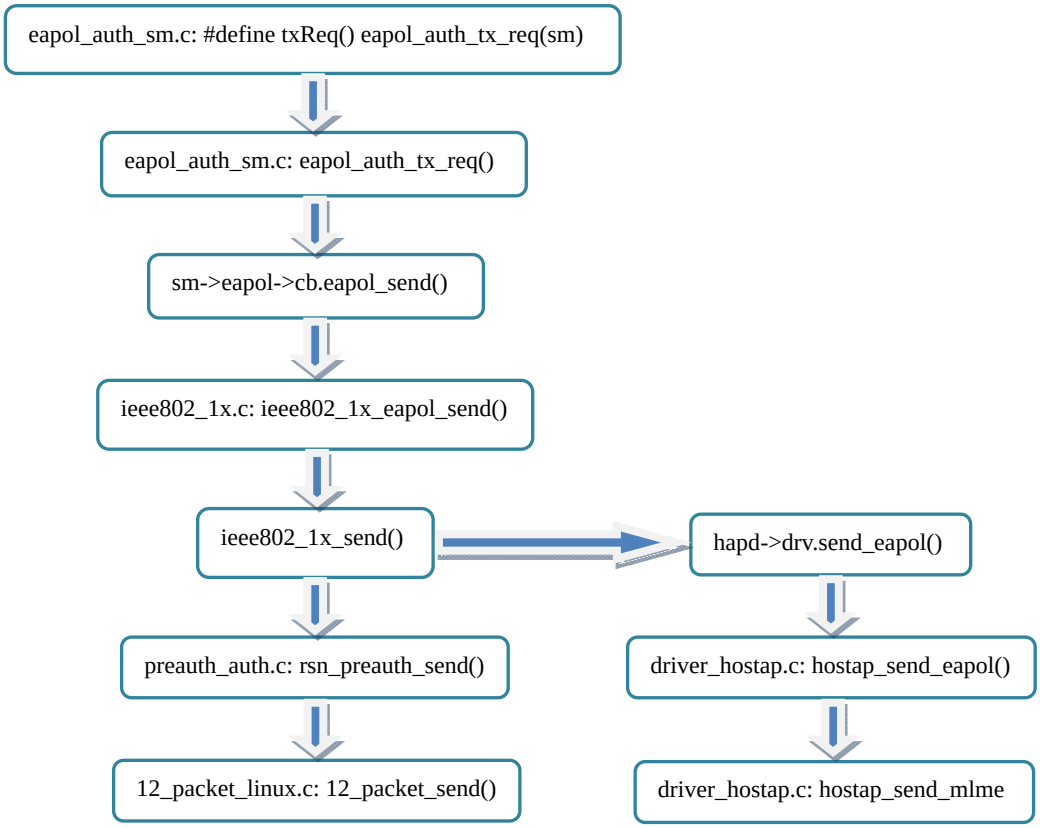

Figure 3-21. EAPOL Sender

3.5 Non-Binary Authenticator

Section 3.5.1 is a comprehensive study of the controlled port. Then based upon the idea of a controlled port Section 3.5.2 will describe the none-binary authenticator.

3.5.1 Port Control in hostapd

In a none-binary authenticator, a new supplicant can enjoy free communication for a certain period of time. This means that the controlled port is kept open from the initialization of the EAPOL state machines till the free open duration times out. Focusing on the key word “port”, it is necessary to answer the following questions before continuing:

1. Which elements can affect the status of the controlled port?
2. How to organize these elements? (To provide logic control of the port)
3. What is the real function of a controlled port?
4. What will happen after the port status changes?

In IEEE 802.1X, the **AuthControlledPortStatus** represents the controlled Port’s status. This is logically viewed as a switch that can be turned on or off, thus permitting or denying the flow of PDUs via that Port. When access is enabled, the status value is “**authorized**”; when it is disabled, its value is “**unauthorized**”.

In addition to the AuthControlledPortStatus, an **AuthControlledPortControl** parameter associated with the controlled Port allows administrative control over the

port's authorization status. This parameter will have one of the values: **ForceUnauthorized**, **Auto** and **ForceAuthorized**; where Auto is the default value. The relationship between the AuthControlledPortStatus and AuthControlledPortControl parameters is:

a) An AuthControlledPortControl value of ForceUnauthorized forces the Authenticator PAE state machine to set the value of AuthControlledPortStatus to be unauthorized; i.e., the Controlled Port is unauthorized unconditionally.
b) An AuthControlledPortControl value of ForceAuthorized forces the Authenticator PAE state machine to set the value of AuthControlledPortStatus to be authorized; i.e., the Controlled Port is authorized unconditionally.
c) An AuthControlledPortControl value of Auto allows the Authenticator PAE state machine to control the value of AuthControlledPortStatus to reflect the outcome of the authentication exchanges between Supplicant PAE, Authenticator PAE, and Authentication Server.

In all three cases, the value of AuthControlledPortStatus directly reflects the value of the **portStatus** variable maintained by the Authenticator and Supplicant PAE state machines. Three factors contribute to the value of the portStatus variable:

a) The authorization state of the Authenticator PAE state machine (assumed to be "Authorized" if the state machine is not implemented for that port).
b) The authorization state of the Supplicant PAE state machine (assumed to be "Authorized" if the state machine is not implemented for that port).
c) The state of the ***Supplicant Access Control With Authenticator*** administrative control parameter. This parameter has two possible values: ***active*** and ***inactive***. The default value of this control parameter is *inactive*; support of the *active* value is optional. The value of this parameter takes effect only if both Authenticator PAE and Supplicant PAE state machines are implemented for that port. If the value of the parameter is *inactive* , then the portStatus parameter value is determined only by the authorization state of the Authenticator PAE state machine. If the value of the parameter is *active*, then the portStatus parameter value is determined by the authorization state of both the Authenticator PAE and Supplicant PAE state machines; if either state machine is in an unauthorized state, then the value of portStatus is unauthorized.

The value of the AuthControlledPortControl parameter for every port of a System can be overridden by means of the ***SystemAuthControl*** parameter for the System. This parameter has one of the values *Enabled* or *Disabled*; its default value is Disabled. If SystemAuthControl is set to Enabled, then authentication is enabled for the System, and each port's authorization status is controlled by the value of the port's AuthControlledPortControl parameter. If SystemAuthControl is set to Disabled, then all ports behave as if their AuthControlledPortControl parameter is set to ForceAuthorized. In effect, setting the SystemAuthControl parameter to Disabled causes authentication to be disabled on all ports, and it forces all controlled ports to be Authorized.

Any access to the LAN is subject to the current administrative and operational state of the MAC (or logical MAC) associated with the port, in addition to AuthControlledPortStatus. If the MAC is physically or administratively inoperable, then no protocol exchanges of any kind can take place using that MAC on either the

controlled or the uncontrolled port. The inoperable state of the MAC has also caused the Authenticator PAE to transit the controlled port to the Unauthorized state.

All the above parameters can directly determine the portStatus. Their relationship and impact are listed in the Table 3-12 below:

Table 3-5. Parameters for portStatus

Relationship of Parameters	Authenticator portStatus	Supplicant portStatus
1 **MAC** → Disabled	Unauthorized	same
→ Enabled → 2		
2 **SystemAuthControl** → Disabled	Authorized	same
→ Enabled → 3	AuthControlledPortStatus = = portStatus of both	
3 **AuthControlledPortControl** → ForceAuthorized	Authorized	same
→ ForceUnauthorized	Unauthorized	same
→ Auto → 4		
4 **Supplicant Access Control With Authenticator** → inactive	Controlled by the Authenticator PAE state machine	
→ active	(authorization state of the Authenticator PAE) || (authorization state of the Supplicant PAE)	

Section 6.6.4 of IEEE 802.1X indicates several mechanisms that can result in the controlled port state changing to unauthorized:

a) The authentication exchanges between the Supplicant and the Authentication Server can result in failure to authorize the port.
b) Management controls can prevent the port from being authorized, regardless of the credentials of the Supplicant.
c) The MAC associated with the port can be non-operational for any reason (including for hardware failure or administrative reasons).
d) Connection failure between the Supplicant and the Authenticator can result in the Authenticator timing out the authorization state.
e) Expiry of a reauthentication timer can occur without successful reauthorization.
f) The Supplicant PAE can fail to respond to a request for authentication information by the Authenticator PAE.
g) The Supplicant PAE can issue an explicit logoff request.

Some of the mechanisms mentioned above are actually duplicates of the parameters in Table 3-12. Still, with so many elements affecting the status of the controlled port, there should be some logic to organize them. The logic lies in the Authenticator PAE state machine, which represents the authentication status of the supplicant.

All the elements and mechanisms above are described as timers and global variables in Section 8.2.2 IEEE802.1X, and they are used later to explain the performance of PACP state machines as well as interstate-machine communication. A comprehensive understanding of them acquaint us with: (1) what effect from outside (of the Authenticator PAE state machine) can cause status change of the port; and (2) what outside will be affected as a result of port's status change in the Authenticator

1

Method

PAE state machine. Later this will help us modify the Authenticator state machine. Timers and global variables related to port are explicitly introduced below.

Mechanism a) occurs in situation 4 in Table 3-12. This is represented by the global variable **authFail**. Mechanism b) means ForceUnauthorized in situation 3 in Table 3-12, which is represented by **portControl**. Mechanism c) is equal to a disabled MAC in situation 1, which is represented by **portEnabled**. Mechanisms d) and f) occur in situation 4 in Table 3-12, and they are represented by the global variables **authTimeout**. Mechanism e) is represented by **reAuthenticate** and Mechanism g) is represented by "eapolLogoff". Table 3-13 explains port-related global variables and timers on the basis of Section 8.2.2 of IEEE802.1X.

Table 3-13. port-related Global Variables and Timers [51]

Variable	**Description**
reAuthWhen	A timer used by the Reauthentication Timer state machine to determine when reauthentication of the Supplicant takes place. The initial value of this timer is reAuthPeriod.
authAbort	This variable is set TRUE by the Authenticator PAE state machine in order to signal to the Backend Authentication state machine to abort its authentication procedure. Its value is set FALSE by the Backend Authentication state machine once the authentication procedure has been aborted.
authFail	This variable is set TRUE if the authentication process (represented by the Backend Authentication state machine) fails. It is set FALSE by the operation of the Authenticator PAE state machine, prior to initiating authentication.
authPortStatus	The current authorization state of the Authenticator PAE state machine. This variable is set to Unauthorized or Authorized by the operation of the state machine. If the Authenticator PAE state machine is not implemented, then this variable has the value Authorized.
authStart	This variable is set TRUE by the Authenticator PAE state machine in order to signal to the Backend Authentication state machine to start its authentication procedure. Its value is set FALSE by the Backend Authentication state machine once the authentication procedure has been started.
authTimeout	This variable is set TRUE if the authentication process (represented by the Backend Authentication state machine) fails to obtain a response from the Supplicant. The variable may be set by management action, or by the operation of a timeout while in the AUTHENTICATED state. This variable is set FALSE by the operation of the Authenticator PAE state machine.
authSuccess	This variable is set TRUE if the authentication process (represented by the Backend Authentication state machine) succeeds. It is set FALSE by the operation of the Authenticator PAE state machine, prior to initiating authentication.
eapFail	This variable is set TRUE by the higher layer if it determines that the authentication has failed.
eapolEap	This variable is set TRUE by an external entity if an EAPOL

2

Method

Variable	**Description**
	PDU carrying a Packet Type of EAP-Packet is received.
eapSuccess	This variable is set TRUE by the higher layer if it determines that the authentication has been successful.
eapTimeout	This variable is set TRUE by the higher layer if it determines that the Supplicant is not responding to requests.
initialize	This variable is externally controlled. When asserted, it forces all EAPOL state machines to their initial state. The PACP state machines are held in their initial state until initialize is deasserted.
portControl	This variable is derived from the current values of the *AuthControlledPortControl* and *SystemAuthControl* parameter for the port. This variable can take the following values:
	1) **ForceUnauthorized.** The controlled port is required to be held in the Unauthorized state.
	2) **ForceAuthorized.** The controlled port is required to be held in the Authorized state.
	3) **Auto.** The controlled port is set to the Authorized or Unauthorized state in accordance with the outcome of an authentication exchange between the Supplicant and the Authentication Server. If SystemAuthControl is set to Enabled, then portControl directly reflects the value of the AuthControlledPortControl parameter. If SystemAuthControl is set to Disabled, then the value of portControl is ForceAuthorized.
portEnabled	This variable is externally controlled. Its value reflects the operational state of the MAC service supporting the port. Its value is TRUE if the MAC service supporting the port is in an operable condition (see 6.4), and it is otherwise FALSE. Both the PAE state machine and the higher layer should be in sync at initialization time. Thus portEnabled is set true together with EAPOL state machine's inialization (*sta->eapol_sm->eap_if->portEnabled = TRUE* , see *ieee802_1x_receive()*). The higher layer is expected to initialize itself when this signal becomes true. The higher layer is expected to reset eapSuccess and eapFail when portEnabled is initially set true.
portStatus	The current authorization state of the controlled port. This variable is set to Unauthorized or Authorized by the operation of the PAE state machines. The value of portStatus directly determines the value of the AuthControlledPortStatus parameter for the port. The value of portStatus is determined from the values of authPortStatus, suppPortStatus, and the *Supplicant Access Control With Authenticator* administrative control parameter, as follows: 1) If both Supplicant PAE and Authenticator PAE state machines are implemented for the port, and the value of the Supplicant Access Control With Authenticator administrative control parameter is *inactive*, then the value of portStatus directly reflects

1
Method

Variable	**Description**
	the value of authPortStatus. Otherwise: 2) If the value of either authPortStatus or suppPortStatus is Unauthorized, then the value of portStatus is Unauthorized. Otherwise: 3) If the values of authPortStatus and suppPortStatus are both Authorized, then the value of portStatus is Authorized.
reAuthenticate	This variable is set TRUE by the Reauthentication Timer state machine on expiry of the reAuthWhen timer. This variable may also be set TRUE by management action. It is set FALSE by the operation of the Authenticator PAE state machine. Reauthentication may not begin immediately. The Authenticator does not interrupt the current authentication, but instead waits for it to complete before beginning a new authentication. Only one pending reauthentication will be tracked.

Question 1 and 2 have already been answered. Actually their explanations are directly from IEEE 802.1X. But question 3 and 4 are implementation dependent.

The standard states that the port is controlled like a tap and there should be no data flow if it is unauthorized. In hostapd, the parameter portStatus is just a symbol that only helps the Authenticator PAE state machine to execute. There is no real port or filtering functions. In short, everything in hostapd is part of the logic of port control. The intrinsic port and the control functions are in the hostap driver.

Nevertheless, there is a connection between the port in hostapd and the real port in the hostap driver. This coupling is used whenever portStatus is changed. This connection is the answer to question 4. Figure 3-26 shows the functions' flow along this connection. Although this figure considers "Authorized" as an example, the flow of "Unauthorized" is the same. The connection ends by calling *ioctl()* to send signals to the driver.

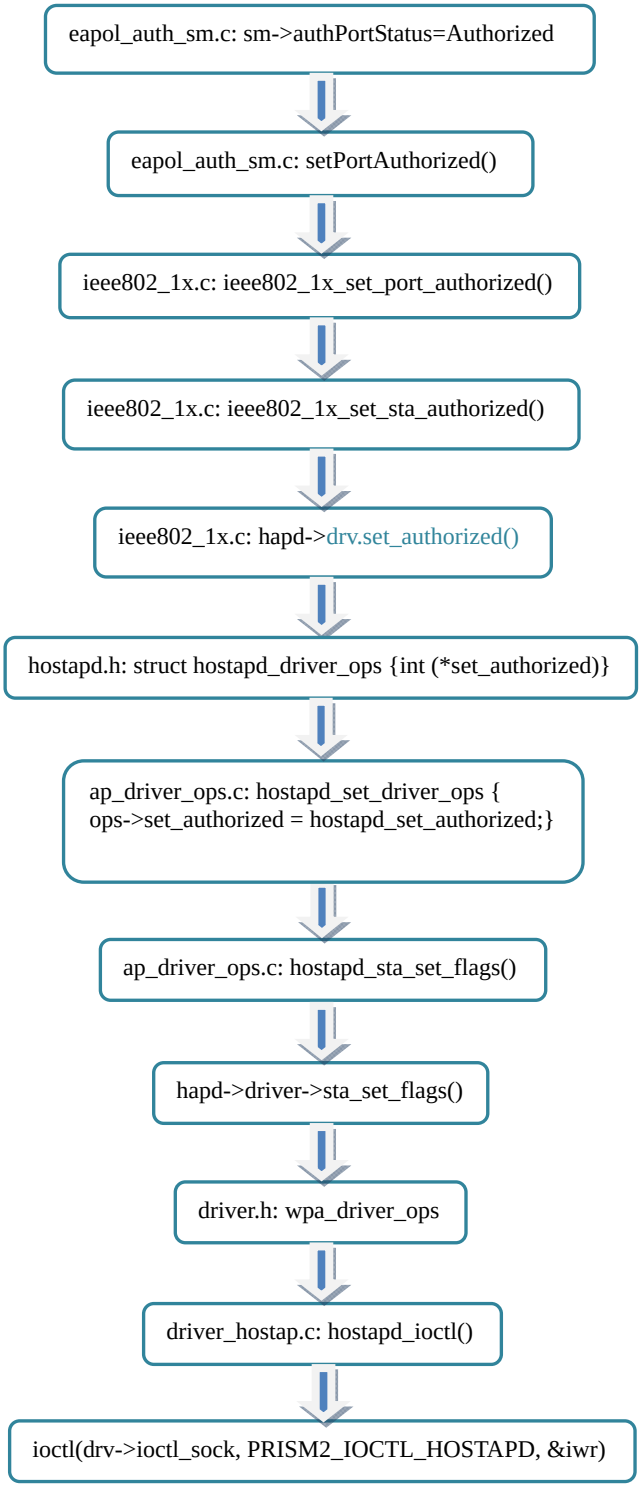

Figure 3-22. After portStatus Change

It is not necessary to consider how the real port is implemented or how the traffic filter is done, as long as we know that the driver will handle it and we know how to inform the driver to do it. The reason for putting port control and traffic filtering in the driver is it takes too much time and requires unnecessary communication between kernel and user space if they are implemented in the user space. If the controlled port was implemented in user space, then all the traffic would need to travel through the

kernel and network stack to the user space where the traffic is finally discarded or redirected. Thus the implementation of the port should be located as close to the hardware as possible. The port should be turned off when the supplicant is first discovered. hostapd deals with 802.11 management and 802.1X authentication frames once the link layer becomes active (see *wpa_supplicant_event()*). Hostap driver never transmits normal data frames to hostapd.

3.5.2 Modification of the Authenticator PAE State Machine

The primary difference between the traditional AP and the non-binary AP is that, the non-binary AP starts authentication with the controlled port open for a period of time. If the supplicant is still not authorized by the time this period expires, then the port will be closed and subsequent authentication will be the same as for IEEE 802.1X. No matter before or after the free open period, the EAP authentication process itself does not change. Therefore the changes should be to the control logic of the port - the Authenticator PAE state machine.

In the new Authenticator PAE state machine, the parameter AuthPortStatus does not change. It is still either Authorized or Unauthorized. Additionally, it is unnecessary to modify the intrinsic port and the control functionalities in the hostap driver. Accordingly what happens after the status change of a port remains the same. The elements affecting PortStatus also remains the same. However, there is a new element "**freeTimeout**" added to the Authenticator PAE state machine, which represents expiration of the initial free open time.

As was mentioned, the whole authentication process will be the same as 802.1X after the free open period expires. Thus it is necessary to preserve all the PACP state machines listed in Section 3.3.2. According to the analysis above, all the state machines remain the same **except** for the Authenticator PAE state machine.

Although the authentication process (represented by the Backend Authentication state machine) and the authentication status (represented by the Authenticator PAE state machine) are separated, the Authenticator PAE state machine (see Figure 3-16, 3-17 and 3-18) still has to be synchronized with the Backend Authentication state machine as well as other PACP state machines. Thus it is impossible to set the port ForceAuthorized in the Authenticator PAE state machine while letting the authentication process go on in the Backend Authentication state machine during free open period. As can be seen in Figure 3-16, FORCE_AUTH is a separate state.

Another design is to add two new states: FREE_DISCONNECTED and FREE_HELD. These two states are clones of DISCONNECTED and HELD except that "*authPortStatus = authorized*" in both of them. The INITIALIZE state unconditionally transits to FREE_DISCONNECTED, where authPortStatus is set authorized and *freeTimeout* is assigned. These clones have exactly the same relationship with other states (except their original ones) as their original ones, which means that the transitioning logic of authPortStatus in the free open time is identical to the traditional system except that the port stays authorized regardless of the authentication result. In order to differentiate the clones from original states, freeTimeout serves as the indicator. If it is not zero, then the states that originally transitted to DISCONNECTED will go to FREE_DISCONNECTED and those states that originally transited to HELD will go to FREE_HELD. When freeTimeout becomes zero, any other state except AUTHENTICATED should transit to

DISCONNECTED unconditionally. However, collisions come after the free open period when freeTimeout keeps zero, since other states cannot decide if they should advance normally or transit to DISCONNECTED unconditionally.

My solution is to introduce a complete colon of the Authenticator PAE state machine, which serves specifically for the free open authentication period. All states in the colon have the prefix “FREE_” added to their original names. The modifications are:

1. In FREE_DISCONNECTED, authPortStatus = Authorized; freeTimeout is set; a script can be called to use IPTable to shape the traffic of the supplicant.
2. In FREE_HELD and FREE_AUTHENTICATED, authPortStatus is left untouched;
3. The INITIALIZE state moves from the original state machine to the clone; it serves as a kick start for the clone state machine -- as well as the whole authentication process.
4. A new state FREE_ENDING replaces the INITIALIZE state in the original state machine. In this state, eapolLogoff is set TRUE and it transits unconditionally to the state ABORTING.
5. When freeTimeout becomes zero, any state in the clone should transit to FREE_ENDING except for INITIALIZE, FREE_DISCONNECTED and FREE_AUTHENTICATED. In this case, FREE_AUTHENTICATED transits to AUTHENTICATED. FREE_DISCONNECTED still transits unditionally to FREE_RESTART.

Figure 3-27 and Figure 3-28 shows the new Authenticator PAE state machine for the non-binary authenticator.

1

Method

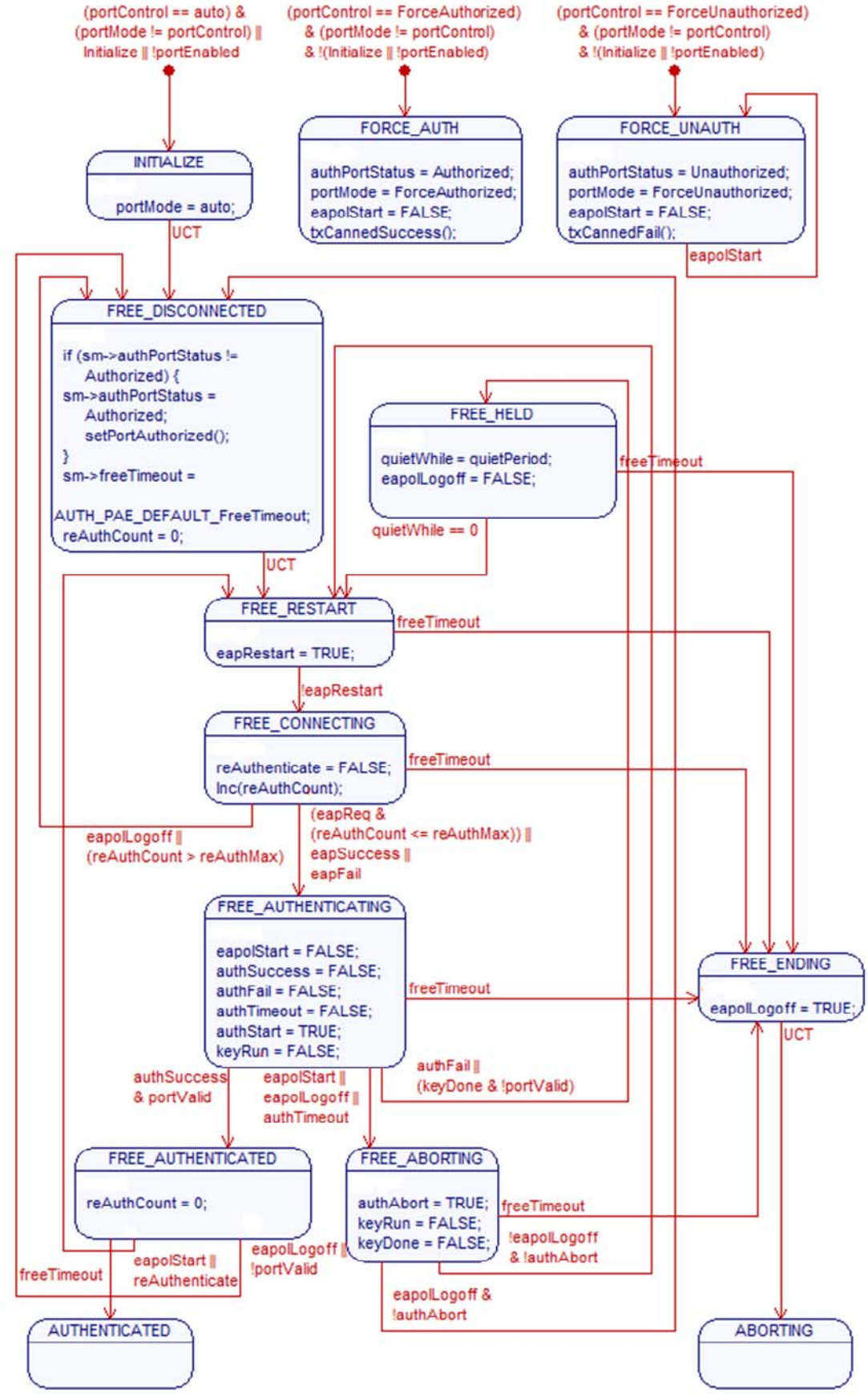

Figure 3-27. non-binary Authenticator PAE state machine - 1

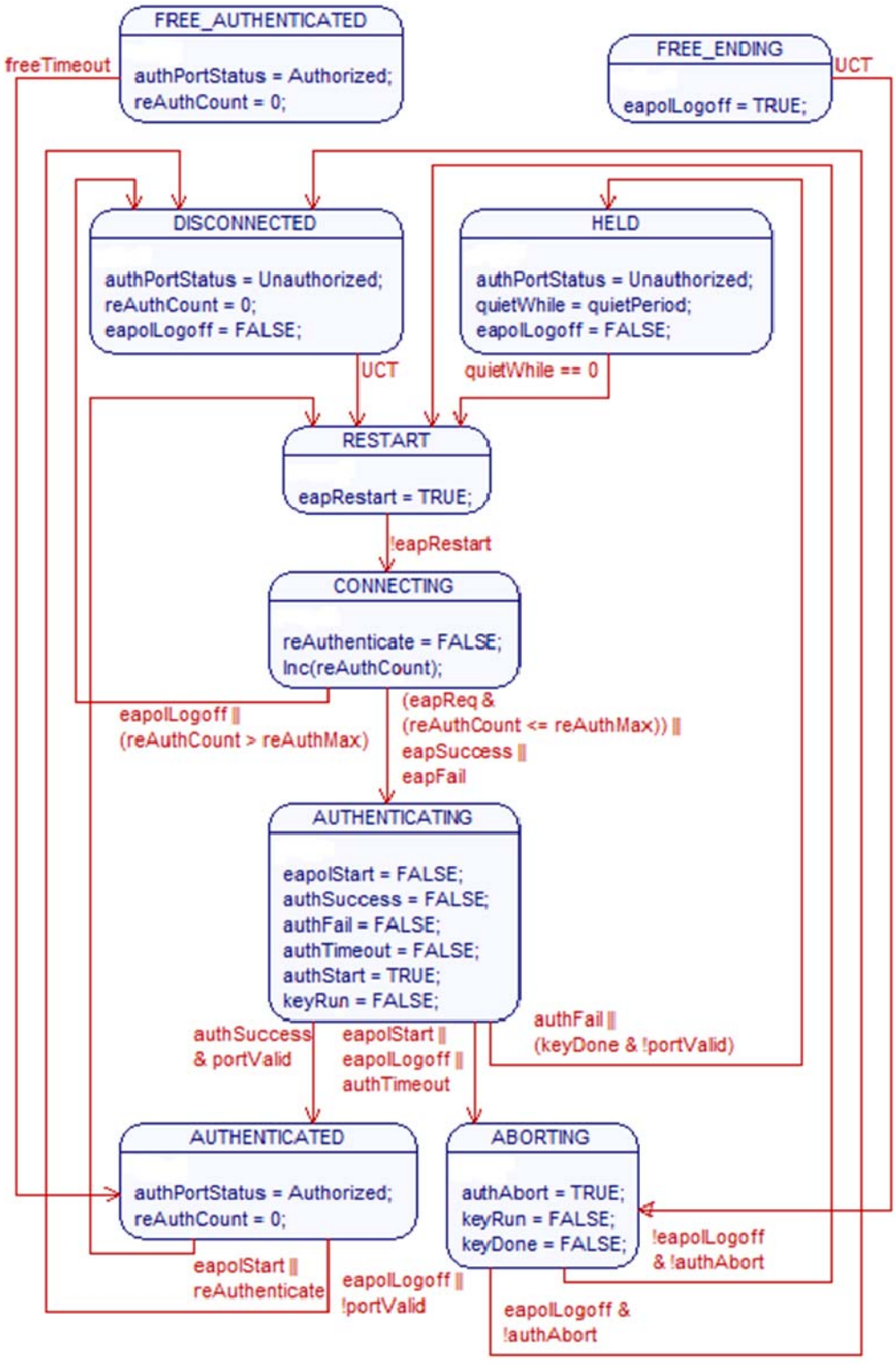

Figure 3-28. non-binary Authenticator PAE state machine - 2

When freeTimeout is zero, it is unnecessary to remember the current state of the clone by transitting to its counterpart in the original state machine in order to avoid a discontinuity in the authentication progress, because it has been given long enough to carry out a legitimate authentication.

Also when freeTimeout becomes zero, the system is not supposed to transit to DISCONNECTED simply to make authPortStatus Unauthorized. The Authenticator PAE must firstly inform the Backend Authentication state machine to terminate the authentication procedure in order to be synchronized for RESTART. Thus the state machine should transit to ABORTING rather than DISCONNCTED. However, exit from ABORING to DISCONNECTED requires EAPOL-Logoff to be TRUE. Otherwise it will transit to RESTART where the authPortStatus remains authorized. So a new state FREE_ENDING is added which sets the variable eapolLogoff FALSE. Sequentially an EAPOL-Logoff causes a transition from FREE_ENDING to ABORING. In ABORTING, authAbort is set TRUE to signal to the Backend Authentication state machine that it should terminate the current authentication procedure. Once the termination is confirmed, the Backend Authentication state machine sets authAbort to FALSE, then it transits to DISCONNECTED. In this case, the port state is forced to Unauthorized and the whole system is successfully synchronized before moving unconditionally to RESTART. In RESTART, the Authenticator PAE informs the higher layer that it has restarted by setting the variable eapRestart TRUE. EAP will acknowledge the restart by resetting eapRestart to FALSE and the Authenticator PAE will advance to CONNECTING as a result.

Appendix B shows the code of the original Authenticator PAE state machine. The following code shows the modification to eapol_state_machine, new Authenticator PAE state machine as well as Port Timers state machine.

```
/**
  * src\eapol_auth\eapol_auth_sm_i.h :
  * struct eapol_state_machine - Per-Supplicant Authenticator state machines
*/
struct eapol_state_machine {
        /* timers */
        …
        /* a new timer */
        int freeTimeout;
        /* global variables */
        …
        /* Port Timers state machine */
        /* 'Boolean tick' implicitly handled as registered timeout */

        /* Authenticator PAE state machine */
        enum {
                /* new states with prefix FREE_ */
                AUTH_PAE_FREE_DISCONNECTED, AUTH_PAE_FREE_CONNECTING,
                AUTH_PAE_FREE_AUTHENTICATING, AUTH_PAE_FREE_AUTHENTICATED,
                AUTH_PAE_FREE_ABORTING, AUTH_PAE_FREE_HELD,
                AUTH_PAE_FREE_RESTART, AUTH_PAE_FREE_ENDING,
                …} auth_pae_state;
        /* variables */
        …
        /* constants */
        …
#define AUTH_PAE_DEFAULT_quietPeriod 60
        unsigned int reAuthMax; /* default 2 */
/* new timeout value for freeTimeout */
#define AUTH_PAE_DEFAULT_freeTimeout 90
#define AUTH_PAE_DEFAULT_reAuthMax 2
        /* counters */
```

```
        …
};

#endif /* EAPOL_AUTH_SM_I_H */

/**
 * src\eapol_auth\eapol_auth_sm.c:
 * eapol_port_timers_tick - Port Timers state machine
 * @eloop_ctx: struct eapol_state_machine *
 * @timeout_ctx: Not used
 *
 * This statemachine is implemented as a function that will be called
 * once a second as a registered event loop timeout.
 */
static void eapol_port_timers_tick(void *eloop_ctx, void *timeout_ctx)
{
        struct eapol_state_machine *state = timeout_ctx;

        if (state->freeTimeout > 0) {
                state-> freeTimeout --;
                if (state->aWhile == 0) {
                        wpa_printf(MSG_DEBUG, "non-binary AP: " MACSTR
                                        " - freeTimeout --> 0",
                                        MAC2STR(state->addr));
                }
        }

        if (state->aWhile > 0) {
                state->aWhile--;
                if (state->aWhile == 0) {
                        wpa_printf(MSG_DEBUG, "IEEE 802.1X: " MACSTR
                                        " - aWhile --> 0",
                                        MAC2STR(state->addr));
                }
        }

        if (state->quietWhile > 0) {
                state->quietWhile--;
                if (state->quietWhile == 0) {
                        wpa_printf(MSG_DEBUG, "IEEE 802.1X: " MACSTR
                                        " - quietWhile --> 0",
                                        MAC2STR(state->addr));
                }
        }

        if (state->reAuthWhen > 0) {
                state->reAuthWhen--;
                if (state->reAuthWhen == 0) {
                        wpa_printf(MSG_DEBUG, "IEEE 802.1X: " MACSTR
                                        " - reAuthWhen --> 0",
                                        MAC2STR(state->addr));
                }
        }

        if (state->eap_if->retransWhile > 0) {
                state->eap_if->retransWhile--;
                if (state->eap_if->retransWhile == 0) {
                        wpa_printf(MSG_DEBUG, "IEEE 802.1X: " MACSTR
                                        " - (EAP) retransWhile --> 0",
```

```
                        MAC2STR(state->addr));
            }
        }

        eapol_sm_step_run(state);

        eloop_register_timeout(1, 0, eapol_port_timers_tick, eloop_ctx, state);
}

/**
 * src\eapol_auth\eapol_auth_sm.c : New Authenticator PAE state machine
 */

/* clone part */

SM_STATE(AUTH_PAE, FREE_DISCONNECTED)
{
        int from_initialize = sm->auth_pae_state == AUTH_PAE_INITIALIZE;

        if (sm->eapolLogoff) {
            if (sm->auth_pae_state == AUTH_PAE_FREE_CONNECTING)
                sm->authEapLogoffsWhileConnecting++;
            else if (sm->auth_pae_state == AUTH_PAE_FREE_AUTHENTICATED)
                sm->authAuthEapLogoffWhileAuthenticated++;
        }

        SM_ENTRY_MA(AUTH_PAE, FREE_DISCONNECTED, auth_pae);

        if (sm->authPortStatus != Authorized) {
            sm->authPortStatus = Authorized;
            setPortAuthorized();
        }
        sm->freeTimeout = AUTH_PAE_DEFAULT_FreeTimeout;
        sm->reAuthCount = 0;
        sm->eapolLogoff = FALSE;

        /*
        * script to call IPTable
        * for traffic shaping
        */

        if (!from_initialize) {
            sm->eapol->cb.finished(sm->eapol->conf.ctx, sm->sta, 0,
                                   sm->flags & EAPOL_SM_PREAUTH);
        }
}


SM_STATE(AUTH_PAE, FREE_RESTART)
{
        if (sm->auth_pae_state == AUTH_PAE_FREE_AUTHENTICATED) {
            if (sm->reAuthenticate)
                sm->authAuthReauthsWhileAuthenticated++;
            if (sm->eapolStart)
                sm->authAuthEapStartsWhileAuthenticated++;
            if (sm->eapolLogoff)
                sm->authAuthEapLogoffWhileAuthenticated++;
        }
```

2
Method

```
        SM_ENTRY_MA(AUTH_PAE, FREE_RESTART, auth_pae);

        sm->eap_if->eapRestart = TRUE;
}


SM_STATE(AUTH_PAE, FREE_CONNECTING)
{
        if (sm->auth_pae_state != AUTH_PAE_FREE_CONNECTING)
                sm->authEntersConnecting++;

        SM_ENTRY_MA(AUTH_PAE, FREE_CONNECTING, auth_pae);

        sm->reAuthenticate = FALSE;
        sm->reAuthCount++;
}


SM_STATE(AUTH_PAE, FREE_HELD)
{
        if (sm->auth_pae_state == AUTH_PAE_FREE_AUTHENTICATING && sm->authFail)
                sm->authAuthFailWhileAuthenticating++;

        SM_ENTRY_MA(AUTH_PAE, FREE_HELD, auth_pae);
      /** obsolete
       *sm->authPortStatus = Authorized;
       *setPortAuthorized();
       */
        sm->quietWhile = sm->quietPeriod;
        sm->eapolLogoff = FALSE;

        eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_WARNING,
                           "authentication failed - EAP type: %d (%s)",
                           sm->eap_type_authsrv,
                           eap_server_get_name(0, sm->eap_type_authsrv));
        if (sm->eap_type_authsrv != sm->eap_type_supp) {
                eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_INFO,
                                   "Supplicant used different EAP type: "
                                   "%d (%s)", sm->eap_type_supp,
                                   eap_server_get_name(0, sm->eap_type_supp));
        }
        sm->eapol->cb.finished(sm->eapol->conf.ctx, sm->sta, 0,
                               sm->flags & EAPOL_SM_PREAUTH);
}


SM_STATE(AUTH_PAE, FREE_AUTHENTICATED)
{
        char *extra = "";

        if (sm->auth_pae_state == AUTH_PAE_FREE_AUTHENTICATING && sm->authSuccess)
                sm->authAuthSuccessesWhileAuthenticating++;

        SM_ENTRY_MA(AUTH_PAE, FREE_AUTHENTICATED, auth_pae);

      /** obsolete
       *sm->authPortStatus = Authorized;
       *setPortAuthorized();
       */
```

1

Method

```
        sm->reAuthCount = 0;
        if (sm->flags & EAPOL_SM_PREAUTH)
                extra = " (pre-authentication)";
        else if (sm->flags & EAPOL_SM_FROM_PMKSA_CACHE)
                extra = " (PMKSA cache)";
        eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_INFO,
                           "authenticated - EAP type: %d (%s)%s",
                           sm->eap_type_authsrv,
                           eap_server_get_name(0, sm->eap_type_authsrv),
                           extra);
        sm->eapol->cb.finished(sm->eapol->conf.ctx, sm->sta, 1,
                               sm->flags & EAPOL_SM_PREAUTH);
}


SM_STATE(AUTH_PAE, FREE_AUTHENTICATING)
{
        SM_ENTRY_MA(AUTH_PAE, FREE_AUTHENTICATING, auth_pae);

        sm->eapolStart = FALSE;
        sm->authSuccess = FALSE;
        sm->authFail = FALSE;
        sm->authTimeout = FALSE;
        sm->authStart = TRUE;
        sm->keyRun = FALSE;
        sm->keyDone = FALSE;
}


SM_STATE(AUTH_PAE, FREE_ABORTING)
{
        if (sm->auth_pae_state == AUTH_PAE_FREE_AUTHENTICATING) {
                if (sm->authTimeout)
                        sm->authAuthTimeoutsWhileAuthenticating++;
                if (sm->eapolStart)
                        sm->authAuthEapStartsWhileAuthenticating++;
                if (sm->eapolLogoff)
                        sm->authAuthEapLogoffWhileAuthenticating++;
        }

        SM_ENTRY_MA(AUTH_PAE, FREE_ABORTING, auth_pae);

        sm->authAbort = TRUE;
        sm->keyRun = FALSE;
        sm->keyDone = FALSE;
}

SM_STATE(AUTH_PAE, FREE_ENDING)
{
        SM_ENTRY_MA(AUTH_PAE, FREE_ENDING, auth_pae);
        sm->eapolLogoff = TRUE;
}

/* orignial part */

SM_STATE(AUTH_PAE, FORCE_AUTH)
{
        ...
}
```

2
Method

1
Method

```
SM_STATE(AUTH_PAE, FORCE_UNAUTH)
{
	...
}

SM_STATE(AUTH_PAE, INITIALIZE)
{
	...
}

SM_STATE(AUTH_PAE, DISCONNECTED)
{
	...
}

SM_STATE(AUTH_PAE, RESTART)
{
	...
}


SM_STATE(AUTH_PAE, CONNECTING)
{
	...
}


SM_STATE(AUTH_PAE, HELD)
{
	...
}


SM_STATE(AUTH_PAE, AUTHENTICATED)
{
	char *extra = "";

	if (sm->auth_pae_state == AUTH_PAE_AUTHENTICATING && sm->authSuccess)
		sm->authAuthSuccessesWhileAuthenticating++;

	SM_ENTRY_MA(AUTH_PAE, AUTHENTICATED, auth_pae);

	if (sm->authPortStatus != Authorized) {
		sm->authPortStatus = Authorized;
		setPortAuthorized();
	}
	sm->reAuthCount = 0;
	if (sm->flags & EAPOL_SM_PREAUTH)
		extra = " (pre-authentication)";
	else if (sm->flags & EAPOL_SM_FROM_PMKSA_CACHE)
		extra = " (PMKSA cache)";
	eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_INFO,
			   "authenticated - EAP type: %d (%s)%s",
			   sm->eap_type_authsrv,
			   eap_server_get_name(0, sm->eap_type_authsrv),
			   extra);
	sm->eapol->cb.finished(sm->eapol->conf.ctx, sm->sta, 1,
				sm->flags & EAPOL_SM_PREAUTH);
```

```
}
SM_STATE(AUTH_PAE, AUTHENTICATING)
{
	...
}

SM_STATE(AUTH_PAE, ABORTING)
{
	...
}

SM_STEP(AUTH_PAE)
{
	if ((sm->portControl == Auto && sm->portMode != sm->portControl) ||
	    sm->initialize || !sm->eap_if->portEnabled)
		SM_ENTER_GLOBAL(AUTH_PAE, INITIALIZE);
	else if (sm->portControl == ForceAuthorized &&
		 sm->portMode != sm->portControl &&
		 !(sm->initialize || !sm->eap_if->portEnabled))
		SM_ENTER_GLOBAL(AUTH_PAE, FORCE_AUTH);
	else if (sm->portControl == ForceUnauthorized &&
		 sm->portMode != sm->portControl &&
		 !(sm->initialize || !sm->eap_if->portEnabled))
		SM_ENTER_GLOBAL(AUTH_PAE, FORCE_UNAUTH);
	else {
		switch (sm->auth_pae_state) {
		case AUTH_PAE_INITIALIZE:
			SM_ENTER(AUTH_PAE, FREE_DISCONNECTED);
			break;
		case AUTH_PAE_FREE_DISCONNECTED:
			SM_ENTER(AUTH_PAE, FREE_RESTART);
			break;
		case AUTH_PAE_FREE_RESTART:
			if (sm->freeTimeout == 0) {
				SM_ENTER(AUTH_PAE, FREE_ENDING);
				break;
			}
			if (!sm->eap_if->eapRestart)
				SM_ENTER(AUTH_PAE, FREE_CONNECTING);
			break;
		case AUTH_PAE_FREE_HELD:
			if (sm->freeTimeout == 0) {
				SM_ENTER(AUTH_PAE, FREE_ENDING);
				break;
			}
			if (sm->quietWhile == 0)
				SM_ENTER(AUTH_PAE, FREE_RESTART);
			break;
		case AUTH_PAE_FREE_CONNECTING:
			if (sm->freeTimeout == 0) {
				SM_ENTER(AUTH_PAE, FREE_ENDING);
				break;
			}
			if (sm->eapolLogoff || sm->reAuthCount > sm->reAuthMax)
				SM_ENTER(AUTH_PAE, FREE_DISCONNECTED);
			else if ((sm->eap_if->eapReq &&
				  sm->reAuthCount <= sm->reAuthMax) ||
				 sm->eap_if->eapSuccess || sm->eap_if->eapFail)
				SM_ENTER(AUTH_PAE, FREE_AUTHENTICATING);
```

```
		break;
	case AUTH_PAE_FREE_AUTHENTICATED:
		if (sm->freeTimeout == 0) {
			SM_ENTER(AUTH_PAE, AUTHENTICATED);
			break;
		}
		if (sm->eapolStart || sm->reAuthenticate)
			SM_ENTER(AUTH_PAE, FREE_RESTART);
		else if (sm->eapolLogoff || !sm->portValid)
			SM_ENTER(AUTH_PAE, FREE_DISCONNECTED);
		break;
	case AUTH_PAE_FREE_AUTHENTICATING:
		if (sm->freeTimeout == 0) {
			SM_ENTER(AUTH_PAE, FREE_ENDING);
			break;
		}
		if (sm->authSuccess && sm->portValid)
			SM_ENTER(AUTH_PAE, FREE_AUTHENTICATED);
		else if (sm->authFail ||
			 (sm->keyDone && !sm->portValid))
			SM_ENTER(AUTH_PAE, FREE_HELD);
		else if (sm->eapolStart || sm->eapolLogoff ||
			 sm->authTimeout)
			SM_ENTER(AUTH_PAE, FREE_ABORTING);
		break;
	case AUTH_PAE_FREE_ABORTING:
		if (sm->freeTimeout == 0) {
			SM_ENTER(AUTH_PAE, FREE_ENDING);
			break;
		}
		if (sm->eapolLogoff && !sm->authAbort)
			SM_ENTER(AUTH_PAE, FREE_DISCONNECTED);
		else if (!sm->eapolLogoff && !sm->authAbort)
			SM_ENTER(AUTH_PAE, FREE_RESTART);
		break;
	case AUTH_PAE_FREE_ENDING:
		SM_ENTER(AUTH_PAE, ABORTING);
		break;
	case AUTH_PAE_DISCONNECTED:
		SM_ENTER(AUTH_PAE, RESTART);
		break;
	case AUTH_PAE_RESTART:
		if (!sm->eap_if->eapRestart)
			SM_ENTER(AUTH_PAE, CONNECTING);
		break;
	case AUTH_PAE_HELD:
		if (sm->quietWhile == 0)
			SM_ENTER(AUTH_PAE, RESTART);
		break;
	case AUTH_PAE_CONNECTING:
		if (sm->eapolLogoff || sm->reAuthCount > sm->reAuthMax)
			SM_ENTER(AUTH_PAE, DISCONNECTED);
		else if ((sm->eap_if->eapReq &&
			  sm->reAuthCount <= sm->reAuthMax) ||
			 sm->eap_if->eapSuccess || sm->eap_if->eapFail)
			SM_ENTER(AUTH_PAE, AUTHENTICATING);
		break;
	case AUTH_PAE_AUTHENTICATED:
		if (sm->eapolStart || sm->reAuthenticate)
```

```
                    SM_ENTER(AUTH_PAE, RESTART);
                else if (sm->eapolLogoff || !sm->portValid)
                    SM_ENTER(AUTH_PAE, DISCONNECTED);
                break;
            case AUTH_PAE_AUTHENTICATING:
                if (sm->authSuccess && sm->portValid)
                    SM_ENTER(AUTH_PAE, AUTHENTICATED);
                else if (sm->authFail ||
                     (sm->keyDone && !sm->portValid))
                    SM_ENTER(AUTH_PAE, HELD);
                else if (sm->eapolStart || sm->eapolLogoff ||
                     sm->authTimeout)
                    SM_ENTER(AUTH_PAE, ABORTING);
                break;
            case AUTH_PAE_ABORTING:
                if (sm->eapolLogoff && !sm->authAbort)
                    SM_ENTER(AUTH_PAE, DISCONNECTED);
                else if (!sm->eapolLogoff && !sm->authAbort)
                    SM_ENTER(AUTH_PAE, RESTART);
                break;
            case AUTH_PAE_FORCE_AUTH:
                if (sm->eapolStart)
                    SM_ENTER(AUTH_PAE, FORCE_AUTH);
                break;
            case AUTH_PAE_FORCE_UNAUTH:
                if (sm->eapolStart)
                    SM_ENTER(AUTH_PAE, FORCE_UNAUTH);
                break;
            }
        }
}
```

3.6 Another Method by Using Linux Firewall

Another thought of implementing none binary authenticator is to utilize Linux firewall system – netfilter. In this scenario, port control is completely disabled. Thus it relies on netfilter to do user access control. In the incoming hook, functions can be created to differentiate authentication packets from normal packets. In the local hook, functions can be made to pass authentication packets to the RADIUS client. There is a white list for legitimate users, a black list keeps track of malicious supplcants, and an on-going list is used to keep track of user addresses that are being authenticated. A new match function can be made to differentiate packets whose source addresses are not found in any of the above lists as coming from new supplicants. IPSet is used to set a timer for each new IP (corresponding to a new supplciant). IPTable is used for traffic filtering if the supplicant fails authentication.

The problem is, if there is no 802.1X context sensed by the supplicant, it would treat its own port as ForceAuthorized, which means that 802.1X and EAP state machines stop working. However, in this scenario, although without port control, EAP state machines are needed for both the supplicant and AP. Thus it is necessary for AP to write a new target function that send out the first 802.1X start message when it detects a new supplicant. So a simplified scenario could be:

1. Write a new match that finds out new MAC addresses.
2. Write a new target that sends out the first 802.1X start.
3. IPSet helps automate steps 1 and 2. If 1 is matched, then go to step 2, with an

upper time limit.

4. If the timer expired, put that user into the black list.

For hook functions, match and target, IPTable, and IPSet refer to Section 2.4, 2.5 and 2.6. For questions regarding Linux kernel, please refer to “Hacking the Linux Kernel Network Stack” [14].

To enhance this system, state machines are needed to handle various time out events, direction control issues, communications between netfilter system and upper layers (EAP & RADIUS Client), and to perform decision making for user access control. This process is more or less equal to rewritting 802.1X, thus the method proposed in the previous section is recommended.

4 Testing and Evaluation

As is mentioned, the real port in hostap driver is closed at the very start. To support seamless roaming, it needs a signal *authorized* from the state FREE_DISCONNECTED in the new Authenticator PAE state machine. It is desirable to modify the driver itself to set the port open by default so that there is not delay.

Moreover, during free open phase the behavior of the new Authenticator PAE state machine can be further simplified. It can focus on ensuring that the supplicant is authentcated and ignore the statistics and signals related to failures. However, if we keep the statistics and signals untouched, which is like Figure 3-27, it is more identical to the original state machine, thus it is easier to understand and has less potentical mistakes.

According to the source code given in Section 3.5.2, to achieve link layer mobility, we only need to add one new integer (freeTimeout), one macro definition (AUTH_PAE_DEFAULT_freeTimeout) and eight new enumertaion members for the new states. Total code lines added are 215.

Besides link layer mobility mechanisms, the new none binary authenticator should also support IP mobility to cover the latency introduced by DHCP [1]. This can be achieved by Mobile IP and SIP. Relevant information can be found in in Chapter 6 of J-O Vatn's dissertation [20].

Nevertheless, there is a big risk for such a new non-binary authenticator. A malicious user can utilize the free access resource as long as he wants by simply wandering back and forth at the boundary of the BSS. The fundamental question regarding this risk is how to define a new supplicant in the AP. As was introduced, a supplicant' information is initialized and accumulated in sta_info since it is first sensed by the AP. All recognized supplicants are put in a list. If a user cannot be found in that list, it is viewed as a new supplicant. For a supplicant that already exists in the BSS, it can become a new supplicant **only** if its EAPOL state machines are destroyed. In hostapd, this process can be simplified as following: *ap_sta_disassociate()* or *ap_sta_deauthenticate()* → *ieee802_1x_free_station()* → *os_free(sm->identity)* & *eapol_auth_free(sm)* → *eap_server-sm_deinit(sm->eap)* & *os_free(sm)*. Note that *eapol_auth_free()* is called to destroy the supplicant's information while *eapol_auth_deinit()* destroys the structure eapol_authenticator -- which is used as system configuration by the structure hostapd. No matter *ap_sta_disassociate()* or *ap_sta_deauthenticate(),* they are called as a result of a system call from hostap driver. Thus it is up to the driver to decide which supplicant should be exterminated (for example, when the wireless signal degrades below a certain threshold).

There are two basic solutions to this problem. The first solution is to make the variable freeTimeout a random value rather than a constant. Since it is not possible to change the diameter of BSS by modifying the radio strength in real life, it is feasible to change the duration of the free authentication. In order to get constant service, a malicious user's behavior is restricted in both location and time. This user needs to move from the boundary after timing out and to reenter immediately afterwards. By giving freeTimeout a random value, the malicious user will not know when they should simulate departing and returning. This random value should range between a value that is neither too short (so that normal supplicants would fail to be

authenticated) nor too long (so that malicious supplicants may count on frequently enjoying free network access).

However, if the malicious user does not need constant service, then he is quite immune to the first solution. For this reason we introduce a second solution, to add those users who are requested to be destroyed by hostap driver, but not yet authenticated in the 802.1X state machine in hostapd to a list named suspicious_list. As was mentioned, the supplicant information is put in a structure sta_info, thus to keep a suspicious_list similar to sta_list would be a heavy burden for the AP. Instead we can copy sm->identity to suspicious_list before os_free(sm->identity) in the step of *ieee802_1x_free_station()*. After that, the user information will be totally destroyed and resources are released. Accordingly, the AP has to check not only sta_list, but also suspicious_list. If the node is in suspicious_list, then it must jump over the clone state machine and go directly to the state DISCONNECTED.

The last question is that, during free open period, users may suffer from communication over an unsafe network (802.1X is a mutual authentication scheme). Since most applications provide higher layer security protection, it will remain safe as long as IP layer mobility is provided.

5 Conclusions and Future work

5.1 Conclusions

For a long time security and flexibility were seen as mutually exclusive in wireless IP access networks. The existence of IEEE 802.1X is crucial for security, but introduces a big hurdle for WiFi users to get out of their zoo. This thesis proposed an innovative method to facilitate seamless WiFi roaming while maintaining the security context required by Robust Security Network. Moreover, it has successfully achieved the goals listed at start: (1) allow the network to provide continued services to supplicants, so that customer's communications will not be cut-off during handover; (2) be completely acceptable to existing supplicant devices and authentication servers, so that both customers and service providers preserve their investments; (3) be compatible with the most commonly used authentication protocol, i.e., IEEE 802.1X; (4) leverage existing access points by a upgrading software rather than requiring installation of special firmware or an upgrade of the hardware. In short our solution increases system functionality remarkably while keeping the cost low.

5.2 Future work

Potential future work would be to address those pitfalls mentioned in Chapter 4. Moreover a test bed for IP Mobility and SIP are required to test this new none-binary authenticator. Wireless IP access will continue to rapidly advancing with additional new standards, thus it is necessary to match our method to both existing and future standard. Ideally there should be a standardization effort to add non-binary authentication to the IEEE 802.1X standard.

References

[1] Thomas E. Eastep, DHCP explanation and operation, 12 January 2010, http://www.shorewall.net/dhcp.htm

[2] Iptables Guide 1.1.19 (Chinese Version) Q&A, 27 May 2007 (based upon Iptables Tutorial by Oskar Andreasson),

http://www.lupaworld.com/action_viewstutorial_itemid_3277.html

[3] Chapter : RADIUS Attribute Descriptions, JUNOSe™ Internet Software for E-series™ Routing Platforms Broadband Access Configuration Guide, last updated 23 February 2005,

http://www.juniper.net/techpubs/software/erx/junose61/swconfig-broadband/html/radius-attributes.html

[4] Yoshihiro Ohba, Mahesh V Kelkar, and Pasi Eronen, Question on EAP state machine, eap Mailing List, 29 June 2005

http://lists.frascone.com/pipermail/eap/msg03467.html

[5] IEEE 802.11-2007, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE 802.11-2007, June 2007.

http://standards.ieee.org/getieee802/download/802.11-2007.pdf

[6] PACKET(7), Linux Programmer's Manual, 29 April 1999, http://swoolley.org/man.cgi/7/packet

[7] Manual of AF_PACKET,6 October 1999, http://bbs.openlab.net.cn/forums/threads/166.aspx

[8] Ma Ying-jeou rstevens rstevens2008@hotmail.com, sk_buff Detailed Analysis, ChinaUnix blog, 18 July 2007

http://linux.chinaunix.net/techdoc/net/2007/07/18/962950.shtml

[9] Jia Zhou, “Adding bandwidth specification to a AAA Sever”, Masters Thesis, Department of Communication Systems, School of Information and Communication Technology, Royal Institute of Technology (KTH); Stockholm, Sweden, COS/CCS 2008-19, September 2008, available at: http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/080914-zhoujia-with-cover.pdf

[10] Zhang Hengchong, “Non-binary authentication: supplicant”, Masters thesis, Department of Communication Systems, School of Information and Communication Technology, Royal Institute of Technology, Stockholm, Sweden, COS/CCS 2009-1, February 2009, available at: http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/090224-Hengchong_Zhang-with-cover.pdf

[11] Kwang-Hyun Baek, Sean W. Smith, and David Kotz, “A Survey of WPA and 802.11i RSN Authentication Protocols”, Technial Report TR2004-524, Dartmouth College Computer Science, November 2004.

[12] William A. Arbaugh, Narendar Shankar, Y. C. Justin, and Kan Zhang, “Your 802.11 Wireless Network Has No Clothes”, IEEE Wireless Communications, 9(6): pp. 44-51, December 2002.

[13] C. Rigney, RADIUS Accounting, RFC Editor, Internet Request for Comments, ISSN 2070-1721, RFC 2866, June 2000, Updated by RFCs 2867, 5080, http://www.rfc-editor.org/rfc/rfc2866.txt

[14] Bioforge, "Hacking the Linux Kernel Network Stack", Phrack, Inc. Volume 0x0b, Issue 0x3d, Phile #0x0d of 0x0f, 13 August 2003, http://www.phrack.org/issues.html?issue=61&id=13

[15] Harald Welte and Pablo Neira Ayuso, Netfilter webpage, last modified November 29, 2008. http://www.netfilter.org/

[16] H3C (IToIP Solutions Expert), "AAA&RADIUS&HWTACACS Introduction",http://www.h3c.com/portal/Products___Solutions/Technology/Security_and_VPN/AAA_RADIUS_HWTACACS/200701/195605_57_0.htm, May 9th.2008

[17] Roaming in general, last modified 20 November 2010, http://en.wikipedia.org/wiki/Roaming

[18] IEEE Std 802.11, 1999 Edition (ISO/IEC 8802-11: 1999) IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Network – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

[19] Vic Yeo, "802.11 Tutorial", updated on 20 July 2005, http://spacehopper.org/mirrors/www.geocities.com/backgndtest/wlan_tut.html

[20] Jon-Olov Vatn, "IP telephony: Mobility and security", Dissertation, Royal Institute of Technology (KTH), School of Information and Communication Technology (ICT), Microelectronics and Information Technology (IMIT), Stockholm, Sweden, TRITA-IMIT-TSLAB AVH:05:01, 2005, 166 pages. http://kth.diva-portal.org/smash/get/diva2:8244/

[21] IEEE 802.11 working group, http://www.ieee802.org/11/

[22] Wi-Fi Alliance, http://www.wi-fi.org/

[23] Jesse Walker. Unsafe at any key size; An analysis of the WEP encapsulation, October 2000. Accessed March 2002 http://www.drizzle.com/~aboba/IEEE/0-362.zip

[24] Jesse Walker. Developer Services - 802.11 Security Series. Part I: The Wired Equivalent Privacy (WEP), 19 June 2002, http://jcbserver.uwaterloo.ca/cs436/handouts/miscellaneous/Intel_Wireless_1.pdf

[25] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In Seventh Annual International Conference on Mobile Computing And Networking, July 2001. Also http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html.

[26] IEEE Computer Society, *IEEE Standard 802.1x-2001, IEEE Standard for Local and Metropolitan Area Networks - Port-Based Network Access Control*, IEEE 802.1X-2001, June, 2001 http://www.ieee802.org/1/pages/802.1x-2001.html

[27] "802.1X White Paper", Allied Telesis, C613-08003-00 Rev. C, 23 May 2006
http://www.alliedtelesyn.com/media/pdf/8021x_wp.pdf

[28] Krishna Sankar, Sri Sundaralingam, Darrin Miller, and Andrew Balinsky, Chapter 7: EAP Authentication Protocols for WLANs, Cisco Wireless LAN Security, Series: Networking Technology, Cisco Press., 2005
www.ciscopress.com/content/images/1587051540/sample**chapter**/1587051540content.pdf.

[29] IEEE 802 LAN/MAN Standards Committee, last updated 10 August 2010, http://grouper.ieee.org/groups/802/

[30] IEEE, "802.11a-1999 High-speed Physical Layer in the 5 GHz band", IEEE, 11 February 1999,
http://standards.ieee.org/getieee802/download/802.11a-1999.pdf

[31] IEEE, "802.11b-1999 Higher Speed Physical Layer Extension in the 2.4 GHz band", IEEE, 2003
http://standards.ieee.org/getieee802/download/802.11b-1999.pdf

[32] "IEEE 802.11g-2003: Further Higher Data Rate Extension in the 2.4 GHz Band", 27 June 2003, (copy of the standard)
http://www.ahltek.com/WhitePaperspdf/802.11-20%20specs/802.11g-2003.pdf

[33] "IEEE Standards for Local and Metropolitan Area Networks: Standard for Port Based Network Access Control", IEEE Std 802.1X-2004, October 2004, http://standards.ieee.org/getieee802/download/802.1X-2004.pdf

[34] Pablo Brenner, A Technical Tutorial on the IEEE 802.11 Protocol, BreezeCOM, 1997, http://www.sss-mag.com/pdf/802_11tut.pdf

[35] CWAP - Certified Wireless Analysis Professional Official Study Guide, Exam PW0-205, McGraw-Hill Osborne Media, 2004, 432 pages, ISBN-10: 0072255854, ISBN-13: 978-0072255850

[36] Pejman Roshan and Jonathan Leary, 802.11 Wireless LAN Fundamentals: A practical guide to understanding, designing, and operating 802.11 WLANs, Cisco Press, 2004, 312 pages, ISBN-10: 1587050773, ISBN-13: 978-1587050770

[37] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz, Extensible Authentication Protocol (EAP), RFC Editor, Internet Request for Comments, ISSN 2070-1721, RFC 3748, June 2004, Updated by RFC 5247, http://www.rfc-editor.org/rfc/rfc3748.txt

[38] Introduction to 802.1X for Wireless Local Area Networks, Revision A, Interlink Networks Inc., 2002, 11 pages,
http://www.lucidlink.com/media/pdf_autogen/802_1X_for_Wireless_LAN.pdf

[39] About RADIUS, Patton Tech Notes, Patton Electronics, Gaithersburg, MD USA, 16 January 2003, 5 pages
http://www.patton.com/technotes/ras_about_radius.pdf

[40] B. Aboba and P. Calhoun, RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP), RFC

Editor, Internet Request for Comments, ISSN 2070-1721, RFC 3579, september 2003, Updated by RFC 5080, http://www.rfc-editor.org/rfc/rfc3579.txt

[41] H3C (IToIP Solutions Expert), “AAA & RADIUS & HWTACACS Introduction”, http://www.h3c.com/portal/Products___Solutions/Technology/Security_and_VPN/AAA_RADIUS_HWTACACS/200701/195605_57_0.htm

[42] Microsoft TechNet, “RADIUS Packet Format”, http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/intwork/inbc_ias_wfwi.mspx?mfr=true

[43] Jonathan Hassell, RADIUS, O’Reilly Media, October 2002, 206 pages, ISBN-10: 0596003226, ISBN-13: 978-0596003227

[44] Deutsche Telekom, WiFi Roaming Solution. World wide wireless, Deutsche Telekom International Carrier Sales & Solutions, 11 April 2008, http://www.telekom-icss.com/dtag/cms/content/ICSS/en/330538

[45] Arunesh Mishra, Minho Shin, and William Arbaugh. An Empirical Analysis of the IEEE 802.11 MAC Layer Handoff Process. Technical Report CS-TR-4395 UMIACS-TR-2002-75, University of Maryland, 2002. Available at http://www.cs.umd.edu/~mhshin/paper/ACMCCRMishra.Shin.Arbaugh.ps (Accessed January 2003).

[46] Héctor Velayos and Gunnar Karlsson. Techniques to Reduce IEEE 802.11b MAC Layer Handover Time. Technical Report TRITA-IMIT-LCN R 03:02, ISSN 1651-7717, ISRN KTH/IMIT/LCN/R-03/02–SE, KTH, Stockholm, Sweden, April 2003.

[47] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. Diameter Base Protocol, RFC Editor, Internet Request for Comments, ISSN 2070-1721, RFC 3588, September 2003, http://www.rfc-editor.org/rfc/rfc3588.txt

[48] IEEE standard 802.11F, IEEE Trial-Use Recommended Practice for Multi-Vendor Access Point Interoperability via an Inter-Access Point Protocol Across Distribution Systems Supporting IEEE 802.11 Operation, June 2003.

[49] Host AP project introduction, http://hostap.epitest.fi/

[50] Developers' documentation for wpa_supplicant and hostapd, last access 28 November 2010, http://hostap.epitest.fi/wpa_supplicant/devel/

[51] J. Vollbrecht and P. Eronen and N. Petroni and Y. Ohba, State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator, RFC Editor, Internet Request for Comments, ISSN 2070-1721, RFC 4137, August 2005, http://www.rfc-editor.org/rfc/rfc4137.txt

[52] IP sets project introduction, last modified 18 April 2005, http://ipset.netfilter.org/

[53] L. Blunk and J. Vollbrecht, PPP Extensible Authentication Protocol (EAP), RFC Editor, Internet Request for Comments, ISSN 2070-1721, RFC 2284,

March 1998, Obsoleted by RFC 3748, updated by RFC 2484, http://www.rfc-editor.org/rfc/rfc2284.txt

[54] C. Rigney, S. Willens, A. Rubens, and W. Simpson, Remote Authentication Dial In User Service (RADIUS), RFC Editor, Internet Request for Comments, ISSN 2070-1721, RFC 2865, June 2000, Updated by RFCs 2868, 3575, 5080, http://www.rfc-editor.org/rfc/rfc2865.txt

Appendix A. sta_info

```
/**
  * ap.h : structure sta_info – It stores all the information of a supplicant
  */
struct sta_info {
        struct sta_info *next; /* next entry in sta list */
        struct sta_info *hnext; /* next entry in hash table list */
        u8 addr[6];
        u16 aid; /* STA's unique AID (1 .. 2007) or 0 if not yet assigned */
        u32 flags;
        u16 capability;
        u16 listen_interval; /* or beacon_int for APs */
        u8 supported_rates[WLAN_SUPP_RATES_MAX];
        int supported_rates_len;

        unsigned int nonerp_set:1;
        unsigned int no_short_slot_time_set:1;
        unsigned int no_short_preamble_set:1;
        unsigned int no_ht_gf_set:1;
        unsigned int no_ht_set:1;
        unsigned int ht_20mhz_set:1;

        u16 auth_alg;
        u8 previous_ap[6];

        enum {
                STA_NULLFUNC = 0, STA_DISASSOC, STA_DEAUTH, STA_REMOVE
        } timeout_next;

        /* IEEE 802.1X related data */
        struct eapol_state_machine *eapol_sm;

        /* IEEE 802.11f (IAPP) related data */
        struct ieee80211_mgmt *last_assoc_req;

        u32 acct_session_id_hi;
        u32 acct_session_id_lo;
        time_t acct_session_start;
        int acct_session_started;
        int acct_terminate_cause; /* Acct-Terminate-Cause */
        int acct_interim_interval; /* Acct-Interim-Interval */

        unsigned long last_rx_bytes;
        unsigned long last_tx_bytes;
        u32 acct_input_gigawords; /* Acct-Input-Gigawords */
        u32 acct_output_gigawords; /* Acct-Output-Gigawords */

        u8 *challenge; /* IEEE 802.11 Shared Key Authentication Challenge */

        struct wpa_state_machine *wpa_sm;
        struct rsn_preauth_interface *preauth_iface;

        struct hostapd_ssid *ssid; /* SSID selection based on (Re)AssocReq */
        struct hostapd_ssid *ssid_probe; /* SSID selection based on ProbeReq */

        int vlan_id;

#ifdef CONFIG_IEEE80211N
        struct ht_cap_ie ht_capabilities; /* IEEE 802.11n capabilities */
```

```
#endif /* CONFIG_IEEE80211N */

#ifdef CONFIG_IEEE80211W
	int sa_query_count; /* number of pending SA Query requests;
				* 0 = no SA Query in progress */
	int sa_query_timed_out;
	u8 *sa_query_trans_id; /* buffer of WLAN_SA_QUERY_TR_ID_LEN *
				* sa_query_count octets of pending SA Query
				* transaction identifiers */
	struct os_time sa_query_start;
#endif /* CONFIG_IEEE80211W */

	struct wpabuf *wps_ie; /* WPS IE from (Re)Association Request */
};
```

Appendix B. Original Authenticator PAE SM

```
/**
 * src\eapol_auth\eapol_auth_sm.c : Authenticator PAE state machine
 */
SM_STATE(AUTH_PAE, INITIALIZE)
{
	SM_ENTRY_MA(AUTH_PAE, INITIALIZE, auth_pae);
	sm->portMode = Auto;
}


SM_STATE(AUTH_PAE, DISCONNECTED)
{
	int from_initialize = sm->auth_pae_state == AUTH_PAE_INITIALIZE;

	if (sm->eapolLogoff) {
		if (sm->auth_pae_state == AUTH_PAE_CONNECTING)
			sm->authEapLogoffsWhileConnecting++;
		else if (sm->auth_pae_state == AUTH_PAE_AUTHENTICATED)
			sm->authAuthEapLogoffWhileAuthenticated++;
	}

	SM_ENTRY_MA(AUTH_PAE, DISCONNECTED, auth_pae);

	sm->authPortStatus = Unauthorized;
	setPortUnauthorized();
	sm->reAuthCount = 0;
	sm->eapolLogoff = FALSE;
	if (!from_initialize) {
		sm->eapol->cb.finished(sm->hapd, sm->sta, 0,
				       sm->flags & EAPOL_SM_PREAUTH);
	}
}


SM_STATE(AUTH_PAE, RESTART)
{
	if (sm->auth_pae_state == AUTH_PAE_AUTHENTICATED) {
		if (sm->reAuthenticate)
			sm->authAuthReauthsWhileAuthenticated++;
		if (sm->eapolStart)
			sm->authAuthEapStartsWhileAuthenticated++;
		if (sm->eapolLogoff)
			sm->authAuthEapLogoffWhileAuthenticated++;
	}

	SM_ENTRY_MA(AUTH_PAE, RESTART, auth_pae);

	sm->eap_if->eapRestart = TRUE;
}


SM_STATE(AUTH_PAE, CONNECTING)
{
	if (sm->auth_pae_state != AUTH_PAE_CONNECTING)
		sm->authEntersConnecting++;

	SM_ENTRY_MA(AUTH_PAE, CONNECTING, auth_pae);
```

```
	sm->reAuthenticate = FALSE;
	sm->reAuthCount++;
}


SM_STATE(AUTH_PAE, HELD)
{
	if (sm->auth_pae_state == AUTH_PAE_AUTHENTICATING && sm->authFail)
		sm->authAuthFailWhileAuthenticating++;

	SM_ENTRY_MA(AUTH_PAE, HELD, auth_pae);

	sm->authPortStatus = Unauthorized;
	setPortUnauthorized();
	sm->quietWhile = sm->quietPeriod;
	sm->eapolLogoff = FALSE;

	eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_WARNING,
			   "authentication failed - EAP type: %d (%s)",
			   sm->eap_type_authsrv,
			   eap_type_text(sm->eap_type_authsrv));
	if (sm->eap_type_authsrv != sm->eap_type_supp) {
		eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_INFO,
				   "Supplicant used different EAP type: "
				   "%d (%s)", sm->eap_type_supp,
				   eap_type_text(sm->eap_type_supp));
	}
	sm->eapol->cb.finished(sm->hapd, sm->sta, 0,
				   sm->flags & EAPOL_SM_PREAUTH);
}


SM_STATE(AUTH_PAE, AUTHENTICATED)
{
	char *extra = "";

	if (sm->auth_pae_state == AUTH_PAE_AUTHENTICATING && sm->authSuccess)
		sm->authAuthSuccessesWhileAuthenticating++;

	SM_ENTRY_MA(AUTH_PAE, AUTHENTICATED, auth_pae);

	sm->authPortStatus = Authorized;
	setPortAuthorized();
	sm->reAuthCount = 0;
	if (sm->flags & EAPOL_SM_PREAUTH)
		extra = " (pre-authentication)";
	else if (wpa_auth_sta_get_pmksa(sm->sta->wpa_sm))
		extra = " (PMKSA cache)";
	eapol_auth_vlogger(sm->eapol, sm->addr, EAPOL_LOGGER_INFO,
			   "authenticated - EAP type: %d (%s)%s",
			   sm->eap_type_authsrv,
			   eap_type_text(sm->eap_type_authsrv), extra);
	sm->eapol->cb.finished(sm->hapd, sm->sta, 1,
				   sm->flags & EAPOL_SM_PREAUTH);
}


SM_STATE(AUTH_PAE, AUTHENTICATING)
{
```

```
	SM_ENTRY_MA(AUTH_PAE, AUTHENTICATING, auth_pae);

	sm->eapolStart = FALSE;
	sm->authSuccess = FALSE;
	sm->authFail = FALSE;
	sm->authTimeout = FALSE;
	sm->authStart = TRUE;
	sm->keyRun = FALSE;
	sm->keyDone = FALSE;
}


SM_STATE(AUTH_PAE, ABORTING)
{
	if (sm->auth_pae_state == AUTH_PAE_AUTHENTICATING) {
		if (sm->authTimeout)
			sm->authAuthTimeoutsWhileAuthenticating++;
		if (sm->eapolStart)
			sm->authAuthEapStartsWhileAuthenticating++;
		if (sm->eapolLogoff)
			sm->authAuthEapLogoffWhileAuthenticating++;
	}

	SM_ENTRY_MA(AUTH_PAE, ABORTING, auth_pae);

	sm->authAbort = TRUE;
	sm->keyRun = FALSE;
	sm->keyDone = FALSE;
}


SM_STATE(AUTH_PAE, FORCE_AUTH)
{
	SM_ENTRY_MA(AUTH_PAE, FORCE_AUTH, auth_pae);

	sm->authPortStatus = Authorized;
	setPortAuthorized();
	sm->portMode = ForceAuthorized;
	sm->eapolStart = FALSE;
	txCannedSuccess();
}


SM_STATE(AUTH_PAE, FORCE_UNAUTH)
{
	SM_ENTRY_MA(AUTH_PAE, FORCE_UNAUTH, auth_pae);

	sm->authPortStatus = Unauthorized;
	setPortUnauthorized();
	sm->portMode = ForceUnauthorized;
	sm->eapolStart = FALSE;
	txCannedFail();
}


SM_STEP(AUTH_PAE)
{
	if ((sm->portControl == Auto && sm->portMode != sm->portControl) ||
		sm->initialize || !sm->eap_if->portEnabled)
		SM_ENTER_GLOBAL(AUTH_PAE, INITIALIZE);
```

```
else if (sm->portControl == ForceAuthorized &&
         sm->portMode != sm->portControl &&
         !(sm->initialize || !sm->eap_if->portEnabled))
        SM_ENTER_GLOBAL(AUTH_PAE, FORCE_AUTH);
else if (sm->portControl == ForceUnauthorized &&
         sm->portMode != sm->portControl &&
         !(sm->initialize || !sm->eap_if->portEnabled))
        SM_ENTER_GLOBAL(AUTH_PAE, FORCE_UNAUTH);
else {
        switch (sm->auth_pae_state) {
        case AUTH_PAE_INITIALIZE:
                SM_ENTER(AUTH_PAE, DISCONNECTED);
                break;
        case AUTH_PAE_DISCONNECTED:
                SM_ENTER(AUTH_PAE, RESTART);
                break;
        case AUTH_PAE_RESTART:
                if (!sm->eap_if->eapRestart)
                        SM_ENTER(AUTH_PAE, CONNECTING);
                break;
        case AUTH_PAE_HELD:
                if (sm->quietWhile == 0)
                        SM_ENTER(AUTH_PAE, RESTART);
                break;
        case AUTH_PAE_CONNECTING:
                if (sm->eapolLogoff || sm->reAuthCount > sm->reAuthMax)
                        SM_ENTER(AUTH_PAE, DISCONNECTED);
                else if ((sm->eap_if->eapReq &&
                          sm->reAuthCount <= sm->reAuthMax) ||
                         sm->eap_if->eapSuccess || sm->eap_if->eapFail)
                        SM_ENTER(AUTH_PAE, AUTHENTICATING);
                break;
        case AUTH_PAE_AUTHENTICATED:
                if (sm->eapolStart || sm->reAuthenticate)
                        SM_ENTER(AUTH_PAE, RESTART);
                else if (sm->eapolLogoff || !sm->portValid)
                        SM_ENTER(AUTH_PAE, DISCONNECTED);
                break;
        case AUTH_PAE_AUTHENTICATING:
                if (sm->authSuccess && sm->portValid)
                        SM_ENTER(AUTH_PAE, AUTHENTICATED);
                else if (sm->authFail ||
                         (sm->keyDone && !sm->portValid))
                        SM_ENTER(AUTH_PAE, HELD);
                else if (sm->eapolStart || sm->eapolLogoff ||
                         sm->authTimeout)
                        SM_ENTER(AUTH_PAE, ABORTING);
                break;
        case AUTH_PAE_ABORTING:
                if (sm->eapolLogoff && !sm->authAbort)
                        SM_ENTER(AUTH_PAE, DISCONNECTED);
                else if (!sm->eapolLogoff && !sm->authAbort)
                        SM_ENTER(AUTH_PAE, RESTART);
                break;
        case AUTH_PAE_FORCE_AUTH:
                if (sm->eapolStart)
                        SM_ENTER(AUTH_PAE, FORCE_AUTH);
                break;
        case AUTH_PAE_FORCE_UNAUTH:
                if (sm->eapolStart)
```

```
                    SM_ENTER(AUTH_PAE, FORCE_UNAUTH);
                break;
            }
        }
    }
```

Appendix C. Original eapol_state_machine

```
/**
  * src\eapol_auth\eapol_auth_sm_i.h :
  * struct eapol_state_machine - Per-Supplicant Authenticator state machines
  */
struct eapol_state_machine {
        /* timers */
        int aWhile;
        int quietWhile;
        int reAuthWhen;

        /* global variables */
        Boolean authAbort;
        Boolean authFail;
        PortState authPortStatus;
        Boolean authStart;
        Boolean authTimeout;
        Boolean authSuccess;
        Boolean eapolEap;
        Boolean initialize;
        Boolean keyDone;
        Boolean keyRun;
        Boolean keyTxEnabled;
        PortTypes portControl;
        Boolean portValid;
        Boolean reAuthenticate;

        /* Port Timers state machine */
        /* 'Boolean tick' implicitly handled as registered timeout */

        /* Authenticator PAE state machine */
        enum { AUTH_PAE_INITIALIZE,             AUTH_PAE_DISCONNECTED,
                AUTH_PAE_CONNECTING,            AUTH_PAE_AUTHENTICATING,
                AUTH_PAE_AUTHENTICATED,         AUTH_PAE_ABORTING,
                AUTH_PAE_HELD,                  AUTH_PAE_FORCE_AUTH,
                AUTH_PAE_FORCE_UNAUTH,          AUTH_PAE_RESTART } auth_pae_state;
        /* variables */
        Boolean eapolLogoff;
        Boolean eapolStart;
        PortTypes portMode;
        unsigned int reAuthCount;
        /* constants */
        unsigned int quietPeriod; /* default 60; 0..65535 */
#define AUTH_PAE_DEFAULT_quietPeriod 60
        unsigned int reAuthMax; /* default 2 */
#define AUTH_PAE_DEFAULT_reAuthMax 2
        /* counters */
        Counter authEntersConnecting;
        Counter authEapLogoffsWhileConnecting;
        Counter authEntersAuthenticating;
        Counter authAuthSuccessesWhileAuthenticating;
        Counter authAuthTimeoutsWhileAuthenticating;
        Counter authAuthFailWhileAuthenticating;
        Counter authAuthEapStartsWhileAuthenticating;
        Counter authAuthEapLogoffWhileAuthenticating;
        Counter authAuthReauthsWhileAuthenticated;
        Counter authAuthEapStartsWhileAuthenticated;
        Counter authAuthEapLogoffWhileAuthenticated;
```

```
	/* Backend Authentication state machine */
	enum { BE_AUTH_REQUEST, BE_AUTH_RESPONSE, BE_AUTH_SUCCESS,
		BE_AUTH_FAIL, BE_AUTH_TIMEOUT, BE_AUTH_IDLE,
		BE_AUTH_INITIALIZE, BE_AUTH_IGNORE} be_auth_state;
	/* constants */
	unsigned int serverTimeout; /* default 30; 1..X */
#define BE_AUTH_DEFAULT_serverTimeout 30
	/* counters */
	Counter backendResponses;
	Counter backendAccessChallenges;
	Counter backendOtherRequestsToSupplicant;
	Counter backendAuthSuccesses;
	Counter backendAuthFails;

	/* Reauthentication Timer state machine */
	enum { REAUTH_TIMER_INITIALIZE, REAUTH_TIMER_REAUTHENTICATE
	} reauth_timer_state;
	/* constants */
	unsigned int reAuthPeriod; /* default 3600 s */
	Boolean reAuthEnabled;

	/* Authenticator Key Transmit state machine */
	enum { AUTH_KEY_TX_NO_KEY_TRANSMIT, AUTH_KEY_TX_KEY_TRANSMIT
	} auth_key_tx_state;

	/* Key Receive state machine */
	enum { KEY_RX_NO_KEY_RECEIVE, KEY_RX_KEY_RECEIVE } key_rx_state;
	/* variables */
	Boolean rxKey;

	/* Controlled Directions state machine */
	enum { CTRL_DIR_FORCE_BOTH, CTRL_DIR_IN_OR_BOTH } ctrl_dir_state;
	/* variables */
	ControlledDirection adminControlledDirections;
	ControlledDirection operControlledDirections;
	Boolean operEdge;

	/* Authenticator Statistics Table */
	Counter dot1xAuthEapolFramesRx;
	Counter dot1xAuthEapolFramesTx;
	Counter dot1xAuthEapolStartFramesRx;
	Counter dot1xAuthEapolLogoffFramesRx;
	Counter dot1xAuthEapolRespIdFramesRx;
	Counter dot1xAuthEapolRespFramesRx;
	Counter dot1xAuthEapolReqIdFramesTx;
	Counter dot1xAuthEapolReqFramesTx;
	Counter dot1xAuthInvalidEapolFramesRx;
	Counter dot1xAuthEapLengthErrorFramesRx;
	Counter dot1xAuthLastEapolFrameVersion;

	/* Other variables - not defined in IEEE 802.1X */
	u8 addr[ETH_ALEN]; /* Supplicant address */
	int flags; /* EAPOL_SM_* */

	/* EAPOL/AAA <-> EAP full authenticator interface */
	struct eap_eapol_interface *eap_if;

	int radius_identifier;
	/* TODO: check when the last messages can be released */
```

```
	struct radius_msg *last_recv_radius;
	u8 last_eap_id; /* last used EAP Identifier */
	u8 *identity;
	size_t identity_len;
	u8 eap_type_authsrv; /* EAP type of the last EAP packet from Authentication server */
	u8 eap_type_supp; /* EAP type of the last EAP packet from Supplicant */
	struct radius_class_data radius_class;

	/* Keys for encrypting and signing EAPOL-Key frames */
	u8 *eapol_key_sign;
	size_t eapol_key_sign_len;
	u8 *eapol_key_crypt;
	size_t eapol_key_crypt_len;

	struct eap_sm *eap;

	Boolean initializing; /* in process of initializing state machines */
	Boolean changed;

	struct eapol_authenticator *eapol;

	void *sta; /* station context pointer to use in callbacks */
};

#endif /* EAPOL_AUTH_SM_I_H */
```

TRITA-ICT-EX-2010:299

www.kth.se