Authentication, Authorization and Mobility in Openflow-enabled Enterprise Wireless Networks

Theresa Enghardt
(tenghardt@net.t-labs.tu-berlin.de)

Master Project Report
FG INET (Internet Network Architectures)
Technische Universität Berlin

WS 11/12 (Version from September 11, 2012)

Abstract

Large-scale 802.11 wireless networks may benefit from Openflow deployment on its Access Points and other forwarding devices combined with centralized management of data flows on an Openflow controller. The reason is that services such as authentication or routing can be provided in an easier way and more efficiently when operating on a full view of the network rather than dealing with distributed state on the individual devices. The interaction of Openflow with mechanisms in wireless networks such as authentication, authorization and mobility of clients may yield new possibilities such as enabling roaming between APs or networks, enhancing handover or providing alternate means of authentication. For instance, computing a set of neighboring APs that the client frequently roams to may enable faster handover due to preauthenticating the station with the neighbors before roaming.
The present student project report aims to explore ways to incorporate Openflow into an enterprise wireless network. It presents three examples of an Openflow-enabled architecture in terms of authentication, authorization and mobility. Furthermore, it describes the deployment of a prototype of one of these architectures in the Berlin Open Wireless Lab (BOWL) testbed, substituting the Linux bridge with the implementation of an Openflow-enabled virtual switch. Correct behavior in terms of authentication, authorization and handover were validated and an unexpected issue of flooding 802.1x frames was observed and corrected.
This work is a proof-of-concept of how Openflow can be deployed in an enterprise Wireless network and proposes alternative architectures that require more implementation work, but enable features such as authentication to an arbitrary server and load balancing.

Contents

1 Introduction

In enterprise wireless networks, multiple Access Points (APs) provide an ideally ubiquitous coverage of a certain area, e.g. a university campus. Broadcasting the same Service Set Identifier (SSID), clients may choose the AP in its range with the best reception and switch to another one if they are mobile. APs are connected via a backbone that can be either wired or wireless.

Due to the wireless nature of the physical channel, it is crucial to verify that only authorized devices have access to the infrastructure, so authentication and access control are vital issues. Furthermore, clients that use this wireless network, such as notebooks and smartphones, are often mobile and can be moved within the network. Due to the limited reach of the individual APs, the mobile device may be forced to perform a handover from one AP to another. This procedure involves scanning for and choosing a new AP to associate with, reauthentication with the new AP and reassociation. Another reason for a handover can be load balancing. However, modifying the behavior of each and every AP is very complicated if all of them have to be configured individually. In the case of load balancing, an AP would have to broadcast its usage statistics to all neighboring APs in order to determine if traffic should continue to be routed over itself or if a handover would be advisable. Centralizing the evaluation of such data and the decision-making may be a better way to provide load balancing. Likewise, authentication is already centralized by using a RADIUS server instead of pre-shared keys. This illustrates some of the benefits centralizing functionality in these networks. Software-Defined Network (SDN) is a mean to implement the centralization of functionality. It aims to provide an abstraction of the forwarding capabilities of all individual devices on a central controller. This means managing them in a centralized way and solving problems like load balancing on a representation of the whole network. A widely used protocol that enables switches to provide an interface to their forwarding capabilities is Openflow. Openflow also provides an open interface to switches which run proprietary code, enabling researchers to deploy alternate protocols without knowledge of the implementation details of the currently running software.
In the present project, different possibilities for deploying Openflow in an enterprise wireless network are considered. Different approaches to the following research questions are compared: How much of the functionality should be centralized and what should remain at the edge of the network? Where should state be maintained, which component should take which decisions and what are the impacts on the other components?

Structure of this report

At first, a selection of related work shows research that already builds on Openflow and/or focuses on the relevant mechanisms in 802.11. The section following the related work gives a brief overview over 802.11 wireless networks and their functionality, focusing on authentication, authorization and mobility. In the next section, Openflow is described in its components and functions. Afterwards three Openflow-enabled architectures for enterprise Wireless Networks are proposed with an assessment of their individual implementation efforts, benefits and drawbacks. The deployment of a prototype of the first of these approaches is subsequently described and some observations are made. Finally, the conclusion is drawn that while Openflow can be successfully deployed in an enterprise wireless network, additional implementation efforts are required for further benefits, which is an incentive for further work.

2 Related work

Wireless networking already has a history of centralization regarding authentication. In most enterprise wireless network, authentication is provided using "WPA-enterprise". This means that each user has an account on a central authentication server and authenticates to this server rather than only to the Access Point (AP). Since 802.11 already supports this [5], it is not necessary to implement it with Openflow.

As Openflow [9] has been intended as a means of enabling researchers to run innovative techniques on commercial switches, there is already much research based on it. NOX [4] is an Openflow controller that has been developed along with Openflow and is widely used. With regards to wireless, Openflow Wireless [20], also called OpenRoads, is an application that aims to facilitate handover between heterogeneous wireless networks. Introduced as an example of the innovation that is possible with Openflow, it provides an API for the migration of devices between an 802.11 network and a UMTS cellular network. Access control is not regarded in OpenRoads, which makes the present work distinct from it. Furthermore, the focus here is on homogeneous (802.11) networks.

Another project related to Openflow in wireless networks is Openwifi [21]. It defines access (i.e. association, forwarding), authentication and accounting (monitoring traffic, billing) as distinct functions and separates them from one another. Open systems authentication or a predefined pre-shared key are used and when an IP address has been obtained via DHCP, traffic is relayed to a dedicated authentication server which usually provides a web interface. Users may now use their credentials from well-known platforms such as Google or Facebook to authenticate using OpenID or OpenAuth and get an access token to be marked as identified at the access point. Openflow is used to relay or block all traffic except ARP, DNS and DHCP and to collect flow statistics for accounting.
Openwifi is similar to the present report since it examines the possibilities of centralizing functionality in a wireless network. However, it does so above the transport layer, so it is not 802.11-specific. Besides, it provides access control by redirecting traffic instead of rejecting it, which is a different approach than mechanisms on the link layer.

There has been much research aiming at improving handover. Several types of handover and the involved trust relationships are examined [1] and taken as a basis to dynamically establish new trust relationships. These can be internal to the Extended Service Set (ESS), e.g. within the enterprise wireless network, or external to it when roaming between two providers. The Inter Access Point Protocol (IAPP) proposed in the IEEE 802.11f draft is

one suggested protocol to facilitate this, but it could also be implemented using Openflow. Thus this work may be a pointer to a useful application of Openflow deployment.

Aiming at the improvement of handover and reducing packet loss, Access Points (APs) may also send buffered data between the discovery and the reauthentication phase [12]. This is due to the observation that scanning for a new AP is a large part of the overall handover delay. Before scanning, a STA may inform an AP that it goes to sleep, so the AP will buffer packets and flush them later when scanning is completed. Furthermore, the new AP listens promiscuously and already handles packets from the STA which are still addressed to the old AP. The first mechanism requires changes on the STA, so it is not feasible by deploying Openflow only on APs and switches while leaving the STA unchanged, which is an assumption of the present report. The second one might be possible to implement using Openflow, provided that a trust relationship has previously been established.

In order to establish the required trust relationships as early as possible, a Frequent Handoff Region (FHR) may be computed [16] [15] [10], which is a set of APs the STA is most likely to roam to next. The required keying material is distributed to them, so the STA is pre-authenticated on them in addition to the currently used one. Furthermore, it is proposed to cache and transfer the session context, e.g. cryptographic keys, counters and sequence spaces, using IAPP. It would be equally possible to use Openflow for this purpose.

There are also other protocols that are intended to help centralize functionality in large-scale wireless networks. For instance, capwap [17] is an IETF proposed standard that specifically aims at managing multiple access points from different vendors.

Onix [6] provides a platform for network-wide control planes and enables them to operate on a global view of the network. It serves to distribute state to the devices, which is a feature also provided by Openflow and controllers such as NOX [4] are comparable to it. It explicitly aims at implementing the SDN paradigm by handling low-level issues and providing a high-level API for programmers to implement control functionality such as authentication, traffic engineering and routing.

Other approaches to centralized management include Dyson [11] which specifically aims at large-scale 802.11 networks. It is a WLAN architecture that consists of wireless clients, APs and a single central controller. The former ones conduct measurements and report them to the controller, enabling it to operate on a view of the whole network, similarly to SDN. Through a scripting API, policies can be implemented which trigger commands to APs and clients. This make it another protocol for centralized control.

3 802.11 Wireless Networks

The IEEE standard 802.11 [5] specifies components, functions and protocols of Wireless Local Area Networks (WLANs). Here their definitions are given and their representations in the Berlin Open Wireless Lab (BOWL) [2] are described. The functions association, authentication and mobility are discussed in more detail.

3.1 Architecture

An 802.11 wireless network typically consists of one or more wireless Access Points (APs) to which clients, commonly called stations (STA) in this context, can associate and with which they can exchange frames containing management information or data. APs are connected to a backbone called distribution network which can be either wired or wireless. Since only authorized hosts should be granted access, STAs typically must authenticate i.e. provide and prove their identity to the AP. In modern large-scale wireless networks, this often involves the use of an Authentication Server (AS). After being granted access to the network, the STA may send and receive data via the AP to communicate with remote hosts.
The BOWL network comprises multiple testbeds, each of which contains multiple wireless nodes that most frequently act as APs, but may also be used as STAs. They are connected to hosts such as a gateway to the exterior network and an AS over a router. Figure 1 contains a generic view of one of the testbeds.

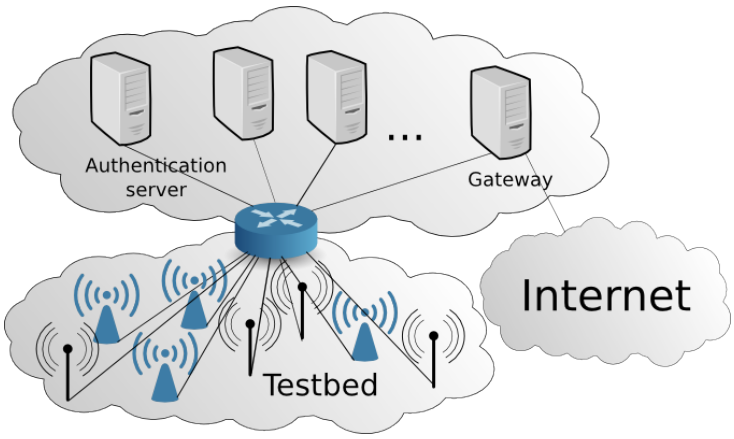

Figure 1: Generic view of a BOWL testbed

3.2 Components

A station (STA) is defined as "any device that contains an IEEE 802.11-compliant medium access control (MAC) and physical layer (PHY) interface to the wireless medium" [5]. Furthermore, every STA runs an implementation of a Port Access Entity (PAE) that controls the forwarding of data to and from the Medium Access Control (MAC). There are two roles for a PAE: authenticator, i.e. the machine granting access to the network, and supplicant, i.e. the machine seeking access.
An Access Point (AP) is a STA that "provides access to the distribution services, via the wireless medium for associated STAs" [5]. It acts as an authenticator by managing a set of associated devices and two virtual ports for each of them: A controlled and an uncontrolled port. The controlled port only accepts 802.1x frames and drops all other traffic, while the uncontrolled port accepts all frames and may bridge them from the wireless interface to the backbone. The latter one can only be used by a STA that has successfully authenticated.
A client is here defined as a non-AP STA which may associate to an AP and act as a supplicant. It can have one of the following states in relation to an AP: not associated, associated, authenticated or active. Authentication to multiple APs at the same time is possible, but association and activity are exclusive. In wireless networks, clients are often mobile, i.e. they can move from the range of one AP to the range of another one, which results in a change of states regarding both. This process is called a handover.
An Authentication Server (AS) is an entity that "determines, from the credentials provided by the supplicant, whether the Supplicant is authorized to access the services provided by the Authenticator." [5] It is often located on a server in the local network, but may also be implemented on the same device as the authenticator.
In the BOWL network, all wireless nodes are STAs and their individual roles are determined by the software executed on them. Devices used as APs typically run hostapd, which implements the authenticator. APs also generally bridge traffic between the wireless and wired interface(s). A device used as a client normally uses wpa_supplicant [8] either with a shared passphrase (WPA-PSK) or with credentials for an account on the RADIUS authentication server (WPA-EAP), which is running freeradius.

3.3 Functions

In order to connect to a wireless network, a STA has to perform a number of steps. Firstly it either passively sniffs for beacons that each announce an available BSSID or actively probes APs on the wireless channels. By local policy and signal strength, it then decides which of the APs it wants to connect to. Before association, the STA performs a first authentication with the AP by sending an 802.11 authentication request management frame and receiving an authentication response frame in return. Since WEP authentication has been deemed insecure as stated in the 802.11 standard [5], wireless networks almost always use Open System authentication and "real" authentication takes place later, after association.

3.3.1 Association

Association is "the service used to establish access point/station (AP/STA) mapping" [5] and allows the distribution system to send frames destined for the client to the correct AP. It can be seen as equivalent to plugging into a wired network. Having scanned for APs in its range and having decided to associate to one of them, the STA sends an 802.11 management frame called association request which contains the SSID, the supported bit rates and some capability information. If an AP accepts the association request, it responds with an association response 802.11 management frame and internally allocates a data structure for the STA, so it can identify the mobile station when delivering buffered frames.

3.3.2 Authentication, Authorization and Data Confidentiality

Substituting the insecure WEP, most wireless local area networks (WLANs) nowadays support Robust Secure Network (RSN), a security standard that provides authentication and data confidentiality defined in [5]. Authentication service is provided using IEEE 802.1x, data confidentiality via Temporary Key Integrity Protocol (TKIP) and the AES-based encryption protocol CCMP. RSN implementations are commonly called Wi-Fi Protected Access 2 (WPA2).
STAs that support RSN may establish a Robust Secure Network Association (RSNA) in the following way: For each associated client a 802.1x port exists on the AP. This 802.1x port consists of a controlled and an uncontrolled port. Initially, an associated STA can only access the controlled virtual port on the AP which only accepts 802.1x data frames and drops all other traffic. The AP acts as the authenticator and requests credentials from the STA,

which typically is either a pre-shared key (PSK), a certificate or a user name and password on an Authentication Server (AS). In the first case, it decides by itself if the credentials are valid and if so, it authorizes the client to use the network. In the latter cases, an authentication server external to the AP decides whether the client is authorized and the AP only forwards the request and the credentials. In case of a successful authentication the AP opens the uncontrolled port for the STA, authorizing it to access the distribution network. Lastly, session keys are negotiated between the AP and the STA, which then becomes active and can access the network.
802.1x authentication is most commonly provided via the Extensible Authentication Protocol (EAP). Between the supplicant and the authenticator, EAP frames are sent as 802.11 data frames, which is called EAP over wireless (EAPOW). For more confidentiality, Protected Extensible Authentication Protocol (PEAP) may be used. In this case AP and client also establish a Transport Layer Security (TLS) session over EAP before credentials are exchanged. The AP communicates with the AS using the RADIUS protocol, typically over UDP. It is important to note that here the authenticator supplies a mechanism for issuing challenges and confirming or denying access, but does not judge the offered credentials. The latter task is performed by the AS.
When the credentials have been accepted, a shared secret between the STA and the AP called the Pairwise Master Key Security Assertion (PMKSA) is derived. Exchanging EAP-key frames, a four-way handshake is performed between STA and AP to negotiate session keys. These are called the Pairwise Transient Key Security Assertion (PTKSA) and are used to encrypt all data frames on the wireless channel between the STA and the AP. The used encryption algorithm is called Counter Mode with Cipher Block Chaining Message Authentication Code Protocol (CCMP). Furthermore, TKIP ensures that each data frame is secured with a different key to guarantee confidentiality.

3.3.3 Handover

If a STA moves out of the range of the AP it is associated to, it may have to roam to another AP, i.e. perform a handover. It decides to do so when signal strength and thus Signal to Noise Ratio (SNR) become too low. After either passively listening for beacon messages or actively probing on all used channels, it chooses a suitable AP and first performs 802.11 authentication with it, which is nowadays typically open systems authentication. It then reassociates by sending a reassociation frame, which is similar to the association frame, but additionally contains the 6-Byte address of the current AP. This

enables the new AP to contact the previous one, i.e. for transferring cached packets for the STA or session information. The Inter Access Point Protocol (IAPP) proposed in IEEE recommendation 802.11F had been intended for this, but it was withdrawn, so it is not used. Either another protocol must facilitate this, or, more commonly, it is omitted and the AP only responds with an association response frame to the STA.

Then 802.1x authentication is performed. If the new AP has cached a PMKSA for the STA in its PAE, the exchange of credentials is skipped and the 4-way handshake is started immediately to establish the PTKSA, i.e. the session keys. The AP manages the controlled and uncontrolled 802.1x port for the STA in the exact same way as in the initial authentication. After the uncontrolled port is unblocked, the STA becomes active on the new AP and can continue to access the network. However there may be a short additional bridging delay for MAC address updates to the Ethernet switches, which ensure that frames intended for the STA are now sent over the new AP.

For the purpose of handover, there is also the 802.11 amendment called "Fast BSS transition". It relies on an external protocol to distribute keys between the pairwise master key holder on the old AP and the new key holder. This protocol is not further specified in the standard, but should establish the state necessary for data connectivity before reassociation.
The 802.11 authentication methods Open Systems Authentication and WEP are now supplemented by a third one which is called FT authentication. It derives additional security assertions so the necessary state is established before reassociation rather than after. This mechanism is currently not used in the BOWL network.

3.4 Message sequence

The message sequence of a non-AP STA from association to becoming active is shown in figure 2. The AP decides to let the station associate to the network and requests its credentials for authentication on behalf of the AS. The AS then makes the decision to grant full access to the STA, authorizing it to send and receive non-802.1x data frames. After successful authentication, a Primary Master Key Security Assertion (PMKSA) is present on the AP, i.e. a shared secret between client and AS from which keying material is then derived. For data confidentiality, the four-way handshake takes place and negotiates several session keys, the Primary Transient Key Security Assertion (PTKSA). When a PTKSA is present, data can be securely exchanged. The PTKSA is destroyed when the STA logs out and terminates its association, the PMKSA remains. The sending of a disassociation frame is optional.

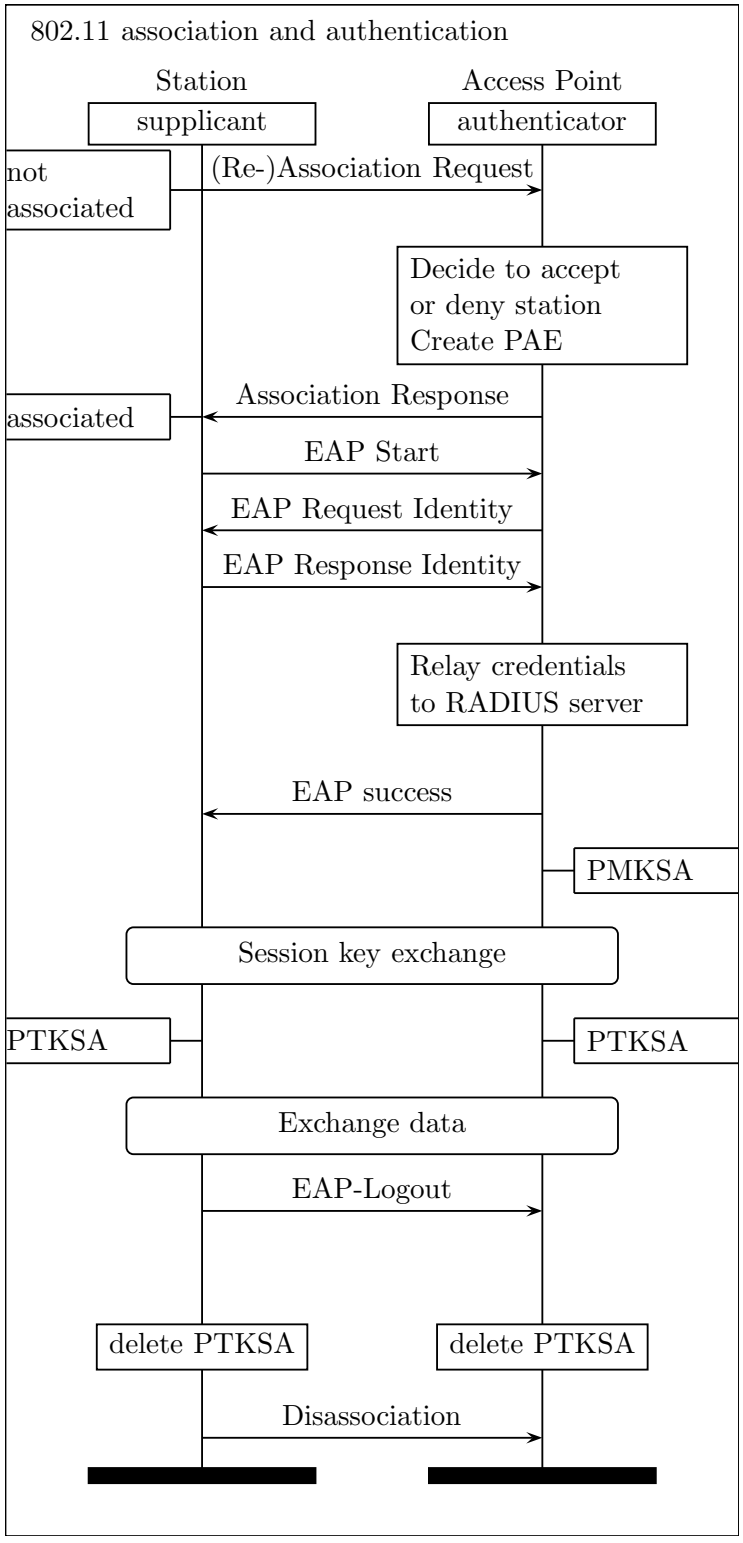

Figure 2: Message sequence chart of 802.1x

4 Openflow

Openflow [9] is a protocol used to manage the forwarding capability of switches. A flow is generally defined as multiple packets that belong together, e.g. due to having the same source/destination address and port. Switches group packets into these flows based on common header fields and store one or more tables of these flows. When the receive a packet, they perform actions on it depending on which flow it matches, e.g. output it on a certain port, flood it to all ports, add or remove a VLAN tag. It is possible to manage the entries of these flow tables and the related actions using the Openflow protocol, e.g. from a central entity on the network called the controller.
Consequently, Openflow can be used to abstract the behavior of the switch on the data plane, i.e. the handling of frames that do not serve to configure the forwarding device but are merely passing through it. Software defined networking is based on this concept.

4.1 Components

An Openflow-enabled architecture consists of at least one switch and one controller.

4.1.1 Switch

An Openflow switch is a device or a process in a communication network that performs packet lookups and forwarding while being managed by an Openflow controller. It may also be referred to as data path and has a unique ID. According to the Openflow specification [19], the switch contains one or more flow tables and a group table containing flow entries, i.e. rules to match packets, counters and instructions for actions to be performed on a packet that matches the flow. The connection to the Openflow controller is established over a channel which is assumed to be secure. The switch communicates with the controller over this channel using the Openflow protocol and can receive commands such as inserting, modifying or removing flow entries.
Originally an Openflow switch was mainly an Ethernet switch that supports Openflow as an interface to modify flow tables, so "vendors do not have to expose the internal workings of their switches" [9]. They enable researchers to deploy custom routing protocols on commercial switches and benefit from "real" traffic at line rate. However, an Openflow switch can also be an implementation of a virtual switch running on any device and applying flow rules to packets. Furthermore, there can be multiple virtual switch instances

running on a physical switch. Each of these has their own data path ID.

4.1.2 Controller

An Openflow controller is a process connected to one or more Openflow switches and manages the entries of their flow tables. It can also query them for status updates such as the state of their ports and traffic counters. This enables it to have a view of the whole network and its topology, so forwarding decisions can be made based on a maximum of information and then inserted into the switches by adding, deleting or modifying flow rules.
An evolved version of such a controller is a framework that abstracts mechanisms such as querying individual switches, fetching information from them and distributing configuration settings to them. This type of framework is also frequently called a network operating system [4]. It supports modules or applications that implement features such as routing algorithms which utilize the framework's API and benefits from the provided abstraction. This is also one of the key concepts of Software-Defined Networking.

4.2 Functions

A switch may actively connect to a controller listening somewhere in the network or on a local socket. It is also possible for it to listen passively for a controller connecting to it. Most controllers and switches support connection on the transport layer via TCP, some also support or require SSL.

4.2.1 Controller-to-switch communication

According to the Openflow specification [19], controller and switch first perform an Openflow handshake where the switch announces its data path ID, the maximum number of packets it can buffer, the number of tables it supports, its capability information, its supported actions and its current ports that support Openflow.
The controller may now send configuration requests such as "Flow Modification" (Flow-Mod) to modify flows or groups in the switch's flow tables or Read-State to query it for statistics. Another type of message is called Experimental and provides the possibility to introduce additional functionality. The switch in turn may send asynchronous messages to update the controller of network events. For example, it sends a Port-Status message if an interface is brought down, it may send a Flow-Removed message if a flow times out and it sends a Packet-In message when it receives a packet that does not have a matching flow table entry or when forwarding to the controller has

been specified as the action of a flow table rule. Packet-In messages contain either a fraction of the packet or the entire one. The controller responds with a Packet-Out message containing a reference to the packet buffer from the Packet-In message and an instruction to apply on the packet.
If idle, controller and switch frequently exchange Hello messages to verify that the connection is still present and possibly also to measure latency or bandwidth.

4.2.2 Packet matching and actions

When an Openflow switch receives a packet, it starts comparing it to the flow entry with the highest priority in the first flow table. If it matches, the corresponding instructions are executed, else matching continues with the second flow rule. If no match is found, the packet may continue to the next flow table, be dropped or forwarded to the controller depending on the switch's configuration.
Supported instructions include sending the packet on a specified port which can be either physical or logical (e.g. 'output to the controller', 'flood to all ports'). The instruction may also point to a Group table entry containing an action that may be referenced from multiple flow entries. Optionally, switches may support adding or removing a VLAN tag, modify the packet header or enqueue the packet in a specific queue, e.g. for Quality of Service (QoS) mechanisms. If no action is specified, the packet is dropped.

5 Approaches to an Openflow-enabled Wireless Architecture

There are different possibilities to integrate Openflow into a wireless architecture. One has to decide what functionality should be administered by a central instance and whether state should be kept there or on the individual devices. Representing different degrees of centralization, three approaches are presented in this section. The first one is least centralized and most similar to an architecture without Openflow, the last one is most centralized where almost all functions have been shifted to the Openflow controller. The approaches vary in where information about client devices is kept and by which application it is handled. For instance, Primary Master Key Security Assertions (PMKSAs), which represent a trust relation between the network and a client, may be stored on each individual Access Point and handled by software such as hostapd. However, in a more centralized architecture, PMKSAs may be stored only on a central host and may be handled by a module of an Openflow controller. This also has an influences on where decisions, such as accepting or denying a client access, are being made.
All of the proposed architectures are examined in terms of possible benefits as well as implementation effort and drawbacks. In the final section, an overall comparison is made.

5.1 Approach A: Handle authentication on the AP

This proposed architecture is the least centralized one, where APs retain most of their original functionality and make the relevant decisions themselves. Openflow only plays a minor role.

5.1.1 Components and functionality

The Access Point (AP) is running an implementation of an Openflow switch in addition to its usual software. The switch implementation substitutes the bridge between wired and wireless interfaces. Association and authentication are handled by the same pieces of software as without Openflow, for example by hostapd [7]. Information about the associated client stations (STA) is stored on the AP and managed by them. This means that regarding association and authentication, the AP keeps state just like in an architecture without Openflow.
When a STA becomes active, i.e. after the four-way handshake has been completed and the uncontrolled port has been unblocked, it sends data over

the network which reaches the Openflow switch implementation that has replaced the bridge. Now the controller is notified about a new host using the network and may store the binding of client to AP in its database to display a view of the topology of the whole network including STAs. When the STA's last flow times out, it is probably no longer active, so the controller receives a timeout message and removes the binding from its database.
Here neither the Openflow switch nor the controller need to implement 802.11 functionality. One way to actively influence the AP's behavior is by inserting flow rules to manage the AP's switching behavior. For example, when multiple wired interfaces are present, the AP can balance load between multiple uplinks. In fact, using Openflow it is possible to deploy arbitrary routing algorithms on the controller to control forwarding on the switch. It is also possible for the controller to set policies for marking traffic, such as adding or removing VLAN tags to packets to or from specific clients. In other words, the backbone of the wireless network benefits from Openflow's capabilities just like any wired network would.

An overview over the necessary modifications to the components and their functionality is displayed in table 1, the message flow between them is shown in figure 3.

Component	**Modifications**	**Tasks**
STA	None	Acts as a supplicant
AP	Substitute bridge by Openflow switch implementation	AP software: Decide about STA association, forward 802.1x to AS, manage keys; Openflow switch: Send first data packet to controller to notify it
Controller	Configure flows	Observe STA-AP-bindings, distribute flows to APs
AS	None	Authenticate STA, decide about authorization

Table 1: Approach A: Overview of components

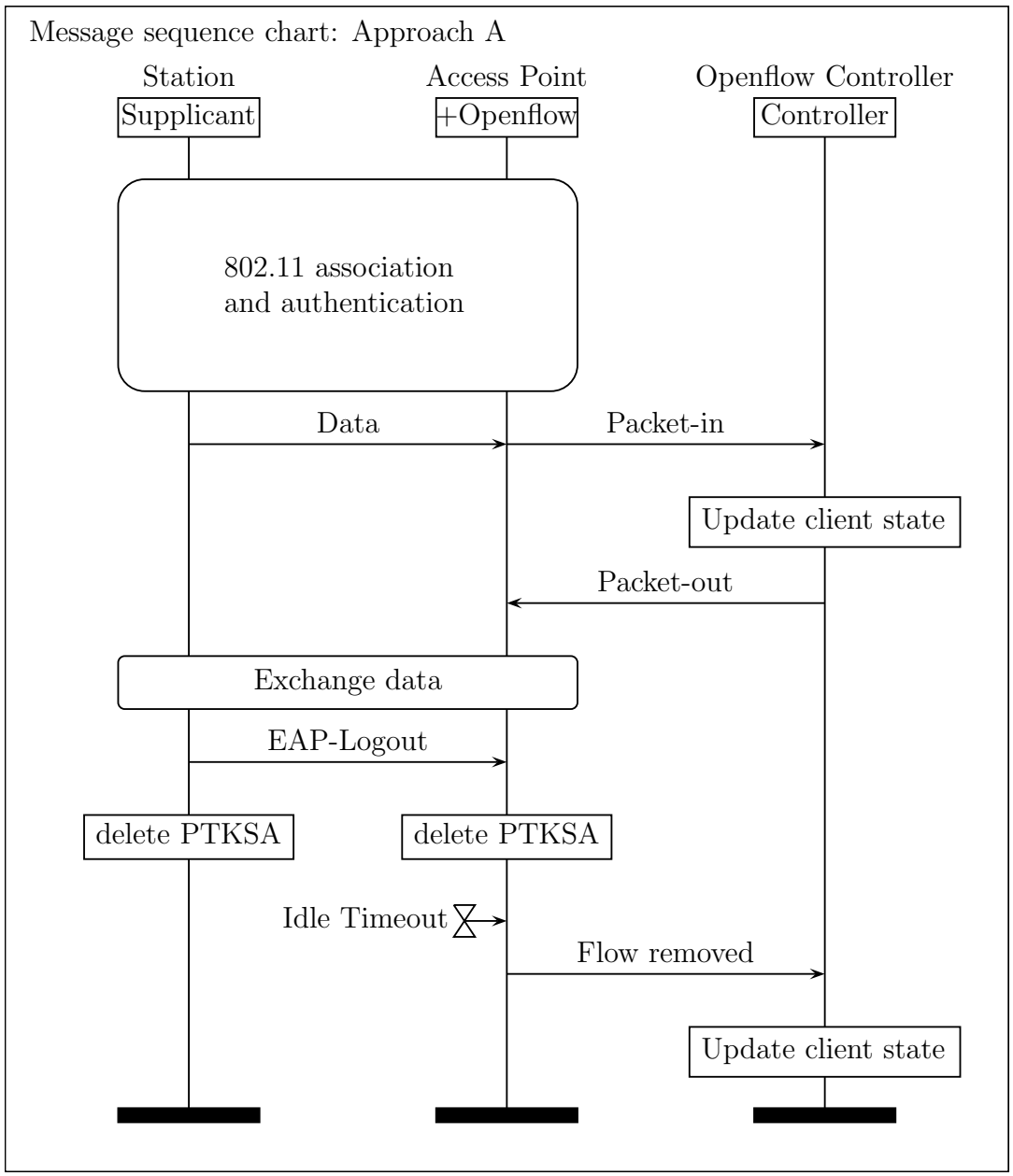

Figure 3: Message flow of proposed architecture A

5.1.2 Advantages and disadvantages

Requiring only minimal changes to the existing infrastructure and using most functionality already present on the APs before Openflow deployment, this approach is fairly easy to incorporate into an existing network. However, the amount of control one has over 802.11-specific functionality is very limited. For instance, admission of clients cannot be managed on the central controller, the decision stays with the existing authentication server. Neither can association be denied to a STA, so load balancing between APs is not possible. With respect to mobility, unfortunately the possibilities for improving handover are very limited. Since neither trust relationships nor keys are managed by or available to the Openflow components, distributing the relevant state to other APs using Openflow is not possible. The controller can merely trace the users' movements from one AP to the next.
However, the controller has an overview over the associated STAs and may record usage statistics. These can help balance the load on switches in the core infrastructure using Openflow. Furthermore, the discovery of highly-used APs may help planning extensions to the network. In addition, this setup yields the "native" benefits of Openflow, e.g. it enables managing VLANs and dynamically assigning clients to them from the central controller using flows.

5.2 Approach B: Handle authentication on the controller

The architecture proposed here is more centralized than the previous one, i.e. it extends the control of Openflow components over 802.11 functionality. The controller is extended to handle 802.1x authentication frames and can decide where they should be forwarded, i.e. which Authentication Server (AS) should be used. It also induces the authorization to forward non-802.1x traffic from a particular STA by changing a flow rule. In approach A, the authorization was initiated by AP software such as hostapd, not by the Openflow switch implementation.

5.2.1 Components and functionality

The Access Point (AP) still handles 802.11 association by itself in its wireless driver, which stays unmodified. The Openflow switch implementation substitutes the bridge between wireless and wired interface and gets a flow rule to forward 802.1x frames to the controller. These are data frames within

the 802.11 protocol, so the Openflow switch implementation is aware of them and can match them by their Ethertype (0x888e). Consequently, it treats them as a data flow. In the previous approach, it sent them to the internal openvswitch bridge port, so a locally running application received and processed them. Here, it sends them to the controller instead. Furthermore, to avoid having unauthorized clients use the network it drops all traffic other than 802.1x from not-yet-authenticated STAs.
The controller now performs tasks that were previously done on the APs: It keeps state for each client, storing its authentication status and security assertions. It also forwards authentication messages to the AS, for example encapsulating them in RADIUS and sending them to a RADIUS server which does not have to be modified. The decision to accept or deny the client access is typically still made on the AS, but the controller may also respond by itself or at least choose which AS to use, which does not necessarily have to be the RADIUS server. When the STA has been authenticated, the controller forwards the Primary Master Key Security Assertion of client and network to the AP, but also retains it. AP and STA now derive their session keys in the usual four-way handshake, with the Openflow switch implementation merely passing on these messages to the responsible application on the AP. Authentication and authorization are now handled largely by Openflow components, but data confidentiality stays with the individual APs.
When a handover occurs, the controller can distribute the Primary Master Key Security Assertion to other APs so 802.1x authentication does not have to be performed again.
An overview of the components with the required modifications and performed tasks is given in table 2, the corresponding message flow is shown in figure 4.

5.2.2 Advantages and disadvantages

This approach requires more implementation effort than approach A, since the controller has to be extended by a module that handles 802.1x frames. The module has to store authentication status and security assertions for each STA and must invoke a flow modification to allow the STA to send non-802.1x traffic when authorized. This puts more load on the controller, which may be an issue in networks with a large number of clients. However, handling authentication on Openflow components means that one may choose other authentication servers than RADIUS. For instance, it may enable users to authenticate to a network using their existing accounts from providers such as Google. A different AS may be chosen for each client, greatly enhancing flexibility.

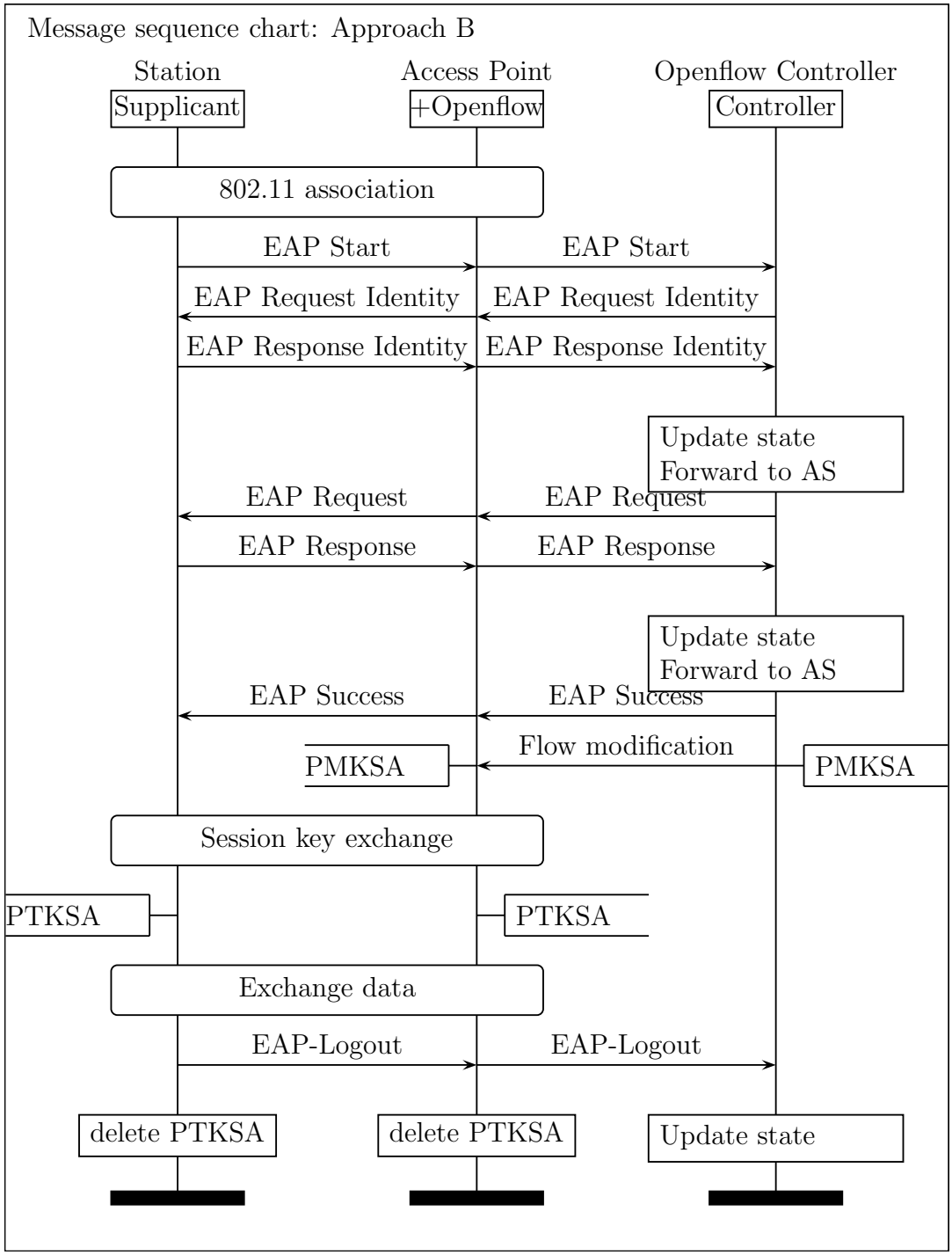

Figure 4: Message flow of proposed architecture B

Component	Modifications	Tasks
STA	None	Acts as a supplicant
AP	Substitute bridge with Openflow switch implementation, modify AP software to disable 802.1x frame handling but leave session key negotiation enabled	AP software: Decide about STA association, negotiate session keys Openflow switch: Forward 802.1x to controller, drop all other traffic unless STA authorized
Controller	Module that handles 802.1x frames, forwards them to the AS and keeps state for each STA, updates flow when STA authorized	Store which STAs are connected to which APs as well as their authentication and authorization status, update flow rule on AP when STA is authorized, optionally distribute PMKSA to APs
AS	None, but does not have to be RADIUS	Authenticate STA, decide whether to authorize it

Table 2: Approach B: Overview of components

Furthermore, PMKSAs are available to the controller, so it can establish trust relationships between STA and other APs necessary for handover before the STA roams, which speeds up the process. Since session keys are still negotiated on the APs, confidentiality is not impaired. However, one must thoroughly secure the controller in order to prevent security risks, e.g. unauthorized clients getting access to the network.
Due to association still being handled by the driver on the individual APs, load balancing is only possible within the core network, not at the edges, similar to proposed architecture A.

5.3 Approach C: Handle all 802.11 frames on the controller

This approach is the most centralized one, having association as well as authentication, authorization, mobility and data confidentiality handled by Openflow components.

5.3.1 Components and functionality

On the AP, the 802.11 driver is modified in such a way that when receiving an association request, it does not send an association response by itself, but hands over the frame to the Openflow switch implementation. Header information of 802.11 must be kept to enable the Openflow switch to identify the frame as a 802.11 management frame, which may require the usage of the monitor interface and/or changes to the operating system of the AP. Matching fields within the 802.11 header may require changes to the Openflow specification, but it could also be implemented as a "vendor-specific" field as a workaround first. Similar to proposed architecture A and B, the Openflow switch implementation replaces the bridge interface between the wireless and the wired interface(s). It forwards the association request to the controller. Having a full view of the network and the number of associated clients to each AP, the controller may now decide to accept the client's association request or to refuse it e.g. due to high load on the chosen AP and other, less busy APs being near the client. It either sends a positive or negative response for the STA to the AP, which forwards it to the client. This requires the controller to generate an 802.11 management frame.
All of the following 802.11 frames are also sent to the controller by the AP, which now acts only as a forwarder and does not keep state of associated or authenticated devices itself. The controller must reimplement not only 802.11 association/disassociation, but also 802.1x authentication, communication with the authentication server and session key negotiation.
Table 3 shows the required changes to the components and the distribution of tasks, figure 5 displays the flow of messages.

5.3.2 Advantages and disadvantages

This being a fully centralized solution, the controller makes all decisions, enabling it to benefit from the full view of the whole network the entire time. However, it requires a huge implementation effort to make the controller aware of 802.11 frames and it might even imply the extension of the Openflow standard. Furthermore, it places a high computational load on the controller, possibly introducing a bottleneck. To ensure scalability, it may be required to deploy multiple controllers, introducing another issue of distributing state between them.
However, it offers the greatest flexibility. For example, load balancing at the edge of the network now becomes possible by denying association to a specific STA on one AP to make it choose another AP. Authentication Servers can be freely chosen and security assertions do not even have to be distributed

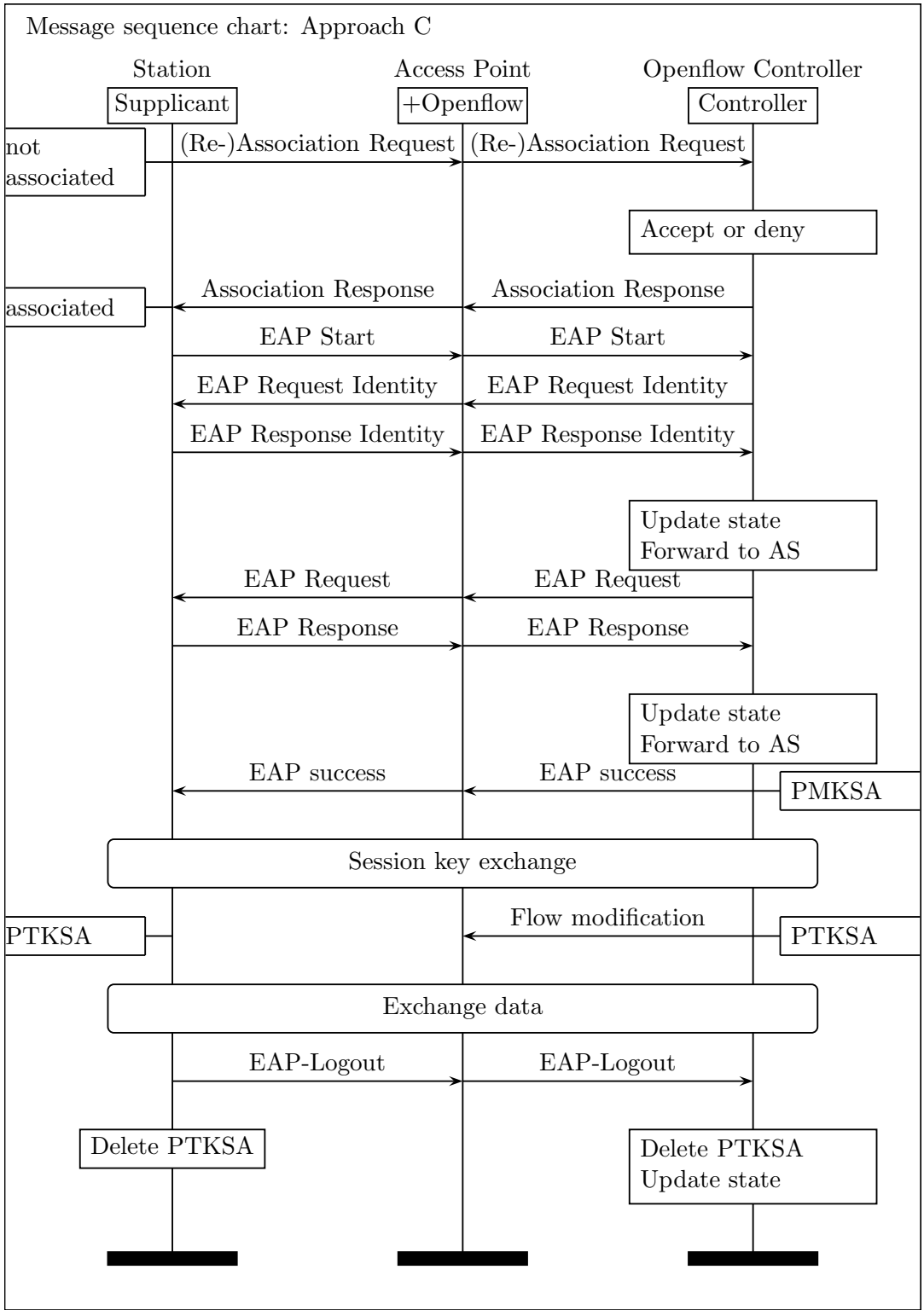

Figure 5: Message flow of proposed architecture C

Component	Modifications	Tasks
STA	None	Acts as a supplicant
AP	Modify 802.11 driver to send raw frames to switch implementation, which matches 802.11 management/data frame header fields	Openflow: Send 802.11 management frames to the controller
Controller	Reimplement 802.11 association handling, authentication frame handling and forwarding to the AS, session key negotiation	Decide about association, keep state for each STA, forward 802.1x to AS, negotiate session keys
AS	None, but does not have to be RADIUS	Authenticate the STA, decide about authorization

Table 3: Approach C: Overview of components

to APs, since authentication is handled at the client. One might even think of improving handover by using the same session keys, reusing the PTKSA and skipping the four-way handshake when a client has previously already been active.

5.4 Comparison of all three approaches

While the first proposed architecture keeps all important state and decision authority within the access point software and out of the Openflow components, the last one basically reimplements 802.11 functionality within an Openflow controller. The latter approach yields additional possibilities and benefits, but it is much harder to deploy, so one must weight the benefits against the implementation complexity and effort.

An overview of the major differences and similarities is provided in Table 4.

Criterion	**A) Authentication on AP**	**B) Authentication on controller**	**C) 802.11 on controller**
Implementation effort	low (install Openflow switch on each AP, install controller, configure flows)	medium (modify authentication software on AP, write authentication module for controller)	huge (modify wireless driver on AP, write association, authentication modules for controller, possibly extend Openflow specification)
Decision about association	by original AP software	by original AP software	by Openflow controller module
State for each STA	by original AP software	trust relation by Openflow controller module, session keys by modified AP software	trust relation by Openflow controller application, session keys by modified AP software or Openflow controller application
Authentication	by original AP and AS software	by Openflow controller module and AS	by Openflow controller module and AS
Decision about authorization	on the original AS, e.g. RADIUS	on a freely chosen AS	on a freely chosen AS
Access Control	by original AP software	by Openflow switch	by Openflow switch
Data confidentiality	by original AP software	by modified AP software	by Openflow controller application or modified AP software
Handover	Controller can observe active STA	Controller can observe associated, authorized and active STA, extend trust relation	Controller can observe associated, authorized and active STA, extend trust relation, manage session keys

Table 4: Comparison of the three proposed architectures

6 Deployment and evaluation

In order to further examine the benefits and pitfalls of Openflow usage in a wireless network as well as its impact in practice, an experimental setup is deployed in the Berlin Open Wireless Lab (BOWL) testbed. The initial prototype consists of an Openflow-enabled Access Point (AP), a wireless client or station (STA) and a virtual machine running an Openflow controller in 'smoketest', a fully-controlled testbed environment. This setup is then expanded to a more advanced prototype in the same testbed featuring two APs. Finally, the setup is deployed in the 'indoor' testbed spanning several floors of a building.
The goal of the evaluation is to determine if authentication, authorization and handover still work as expected when Openflow is deployed on the APs, e.g. if access is still granted to STAs possessing the correct credentials and denied to STAs using the wrong credentials.
The following section first describes the components that are used and their implementation-specific features. Then the individual setups are specified in terms of configuration and validation. It is evaluated whether the behavior of the setup matches the expected message flow previously described in section 5.1. Some observations of the performance (jitter and packet loss) or of unexpected behavior follow.

6.1 Components of the setup in the BOWL testbed

6.1.1 Testbed infrastructure

The BOWL 'smoketest' testbed consists of several wireless nodes located in a server room. Each of them possesses a wired and one or two wireless interfaces, the former one being connected to a switch. This switch in turn connects them to the core infrastructure where several hosts are located, such as the RADIUS server, the gateway from which configuration is orchestrated and the Openflow controller. The relevant devices with their interconnection are depicted in figure 6.

The devices used are three wireless nodes, two as APs and one as STA. They include two Alix boards running OpenWRT Backfire [14], a Linux distribution for embedded devices, with kernel 2.6.32.27 and some BOWL-specific patches. The git commit ID is 9487b07dcd341078f4cac06d175fb20d82af049b. The relevant wireless driver is ath9k, the nodes are running hostapd v2.0-devel when acting as APs and wpa_supplicant v2.0-devel when acting as stations. The environment facilitates 802.1x authentication within the local realm using a RADIUS server running freeradius in the core infrastructure.

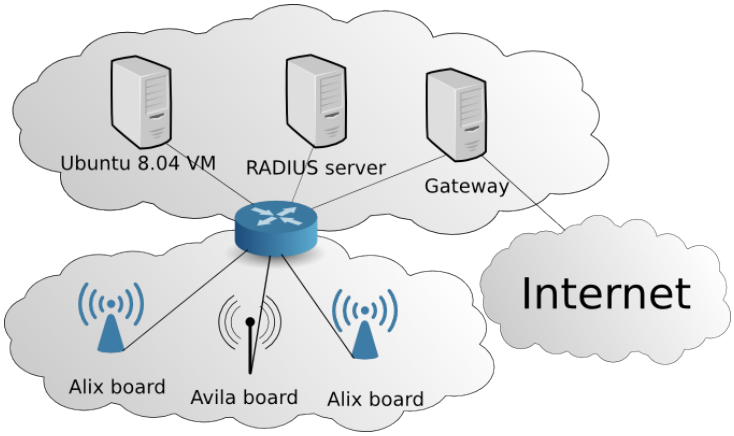

Figure 6: Relevant devices for prototype deployment in the 'smoketest' testbed

The third wireless node is an Avila board running the same operating system, equipped with an atheros card and madwifi driver. Acting as a STA, the same version of wpa_supplicant as on the other nodes was installed.

Initially, a personal notebook PC was used as a STA to connect to the AP, but this had several drawbacks. Due to the physical distance between working desk and testbed, signal strength of the AP was low and at times, a stable association could not be maintained between the client and the AP. Furthermore, the notebook was running several processes which generated packets irrelevant to the Openflow setup, generating unnecessary overhead and 'noise' in the packet capturing traces. As a consequence, it was decided to use another testbed node as the client, which is located next to the AP and which is not running any additional applications.
Generally the nodes in the smoketest testbed can be configured using a command-line interface. There are two options for remote management: To use the terminal server which is reachable from the gateway and connected to each node via a serial line, providing telnet access to them, or to connect to the nodes over the testbed itself via SSH. In the present case, the former option is preferred because it does not require a working node setup with reachable interfaces. This is required because for instance, since the wired interface of an AP is used in the Openflow data plane, it temporarily becomes unreachable prior to running Openflow. Having no access to the AP over an alternate connection, one would be rendered unable to continue configuration and to start Openflow, losing connection to the AP altogether.

Thus the serial line is required for nodes to be accessible at all times, even when the IP setup is not yet complete.
The node also sends its startup log over the serial line, which provides more possibilities for troubleshooting than an SSH session. The only drawback is that typically there can be only one telnet session over a serial line at a time. However it is still possible to use multiple virtual terminals on the node by starting a screen session. This yields the possibility to switch between windows running different programs on the same terminal.
Consequently, on each wireless node a screen session for configuration is started. Both Alix boards can be accessed over a serial line. Unfortunately the third node, the Avila board, is not reachable via the terminal server. Thus there is no telnet connection, which is why an ssh session via the wired interface had to be used for management and configuration. Since the only stable way to reach this node is via the gateway host and the wired interface, routing had to be set up carefully. If traffic was routed over the wireless interface and this interface disconnected, communication with this node would be cut off completely and configuration would become impossible. The ssh connection would break and connectivity would have to be restored by rebooting, i.e. power cycling the node. Thus only traffic to certain hosts could be routed via the wireless interface, not all traffic to the whole core subnet.
Using the Unified Control Interface (UCI) [18], the basic configuration of the interfaces is specified in two configuration files, one for wireless setup and one for network setup. Based on the information provided there, the interfaces are configured when brought up or down by executing the appropriate commands and creating configuration files for applications such as hostapd or wpa_supplicant. Their setup is also restored upon reboot, i.e when power cycling the nodes, making the setup more robust.

6.1.2 Choice of Openflow software

Concerning software, openvswitch [13] was selected to be used as the Openflow switch implementation on the nodes. Openvswitch is an open-source apache-licensed implementation of a virtual switch that supports Openflow. It was chosen because it is Linux-based and there was already a packaged version for OpenWRT at the time of deployment. It provides a kernel module named kmod_openvswitch that can act as a replacement for the linux bridge, daemons and userspace tools to configure the behavior of the switch, all of which are accessed via the command line. These aspects make it suitable for usage in the BOWL testbed. While current releases have a strong focus on Virtual Machines, it can be ported to many more environments due to

being written in platform-independent C. It also includes a simple Openflow controller that can run locally for initial testing.
The packaged version of openvswitch initially deployed on the 'smoketest' nodes is 1.0.3, see table 5. It includes a 'simple switch' reference implementation while there is also a 'full-featured switch' that uses a more sophisticated database for storing flows, which can then be accessed by a daemon that manages multiple virtual switches or data paths. However, the latter implementation could not be run for the time being because there were some userspace tools missing. Namely the tools for database creation ovsdb-tool, the database server ovsdb-server and the daemon ovs-vswitchd were not in the package.
For simplicity and due to the initial focus lying on Openflow functionality and not implementation and performance details, the 'simple switch' reference implementation was used at first. It requires setting up a data path, which corresponds to a 'switch' in terms of Openflow and may contain one or multiple flow tables. When interfaces are added to it, it intercepts all traffic on these interfaces which are then called ports, just like a bridge. One physical switch may contain multiple data paths, e.g. with different ports or on different VLANs. On each of these the Openflow daemon may then be started, connect to a controller and manage flows on this data path, i.e. handle all packets on it. If it is reachable via one of the interfaces that are part of the same data path, it uses in-band mode, else it connects in out-of-band mode. It is possible to connect to Openflow controllers using either TCP, SSL or a Unix domain socket file. There is also the possibility of running the daemon in passive mode, listening for a controller connecting to it.
Unfortunately, the configuration of the used openvswitch version is not supported by UCI. There is a userspace only implementation of openvswitch called Pantou which can be configured via UCI, but it is not applicable to the version that uses the kernel module. Consequently, configuration of the access points has to be done manually via the command line interface.

Package:	openvswitch
Version:	1.0.3-1
Architecture:	x86
Installed time:	1326467667

Table 5: Package information for openvswitch

Regarding the Openflow controller, multiple implementations were considered, notably NOX [4] and floodlight [3]. The former one is a platform

for building network control applications, which provides a C++ API supporting Openflow 1.0. Having been developed by Nicira Networks alongside Openflow, being targeted at recent Linux distributions and having been used in some research projects as described in [4], it seemed to be the obvious choice. Unfortunately, at the time of prototype deployment compiling NOX did not work as expected on some Debian and Ubuntu machines it was tested on. Even on Debian stable (version 5.0 squeeze) there were some issues with the cmake dependency. Since the virtual machine to be used as the Openflow controller in the BOWL testbed was running Ubuntu 8.04 at that time, deploying NOX on it appeared to be a tough challenge. Furthermore, the API was not too well documented and there were no plans for substantial further development. Due to time constraints an alternative had to be found.
Floodlight [3] is an open-source implementation of an Openflow controller which is released under the Apache license. Most parts are written in Java, so it can run largely platform-independent and is easily extensible by writing Java classes. Since it requires only a Java Virtual Machine, it is easy to deploy. Development is ongoing and there is an active community. These are the reasons for which floodlight was finally chosen. However, in order to run floodlight properly the virtual machine had to be upgraded to Ubuntu 10.04 after all.
Floodlight's core package exposes an interface for other classes that allows them to communicate with connected switches. The main class loads floodlight modules which implement functions such as forwarding, topology updates and managing information about devices, i.e. hosts connecting to the network. Some important modules are shown in table 6. One important module is LearningSwitch, which first floods unknown packets to all interfaces and stores which network address (MAC or IP) is behind which port. Once it identifies a pair of network addresses communicating, it inserts a flow into the switch so packets are sent only to the appropriate port rather than flooded on all ports. Another module, the DeviceManager, keeps track of all hosts communicating via the switches and stores their point of attachment to the network, i.e. data path ID and port.

By default, floodlight listens for connections from Openflow switches on port 6633 and communicates with them using the Openflow protocol. It also provides a REST API which can be accessed on port 8080 and queried for information such as the connected switches, their status and their flows. Furthermore, it is possible to insert flows into them using the Static Flow Pusher over the same API.
Due to floodlight being a relatively new project at the time of prototype deployment, releases were frequent and different versions were used throughout

Module	Description
LearningSwitch	Store mappings of MAC address or VLAN to Switch Port in a host table, write flow modifications based on this
DeviceManangerImpl	Observe hosts whose MAC address is seen on the network and track their location in the network as well as their network addresses
Forwarding	Look up a packet's destination in the DeviceManager, find a path via the TopologyManager and send the packet to the correct port, else flood
StaticFlowEntryPusher	Periodically add static flows to switch tables, query switch tables
TopologyManager	Keep track of the switches and their interconnections

Table 6: Important floodlight modules

the project. Initially, version 0.8 was deployed and later it was upgraded to version 0.82. At that time the API was to be considered non-finalized, so documentation did not necessarily correspond to the actual code running. Thus it was necessary to read the code in order to understand how to specify a static flow and how packets can be matched. To have an overview, a table of all possible options in version 0.82 was written in the Floodlight documentation wiki. By subscribing to the mailing list, it was possible to keep track of the important changes. There was a minor change in the URI of the API, a 'core/' being added, which was also communicated over the list. A typo in the implementation was reported so it could be corrected. Furthermore, some issues with the newly-developed module loading system were reported and fixed within a short time frame.
The modules loaded by default in version 0.8 are LearningSwitch, DeviceManagerImpl, PktInProcessingTime, CounterStore and StaticFlowEntryPusher. This is why these modules were used in the first prototype, since for using

other modules, it would have been necessary to modify the code. Version 0.82 introduced a module loading system and thus made it possible to specify the modules to be loaded by writing a configuration file prior to running the controller. By default, the LearningSwitch module was substituted by the newly-developed Forwarding Module. In order to produce a similar behavior with both versions, the configuration file for version 0.82 was set to the modules that had been used in version 0.8, making the controller act like a learning switch.

6.2 Initial prototype deployment

6.2.1 Configuration

The setup of the initial prototype is shown in figure 7. The first Alix board was configured as an AP by setting its wireless interface to the appropriate mode in the UCI configuration file and giving it an SSID. Initially WPA2 with a pre-shared key was used for authentication to test whether the STA properly associates to the AP without 802.1x and because the account on the RADIUS server was not yet present. Later WPA2 was used in conjunction with the RADIUS server, so 802.1x authentication would take place. The parameter for encryption mechanisms was set to CCMP and TKIP. An exemplary configuration file is provided in the appendix A.1.1.

On the node acting as a station, the wireless interface was set to STA mode in the UCI configuration file. It was configured to associate to the SSID advertised by the AP and to authenticate using WPA2, at first using the passphrase. Later with 802.1x the encryption method was explicitly set to CCMP/TKIP and the credentials for authenticating with the local RADIUS server were provided. Furthermore, the wireless interface was given an IP address from the subnet of the wireless nodes in the testbed. The default route was set via this interface. Since access to the node was provided by the terminal server over a serial line, the wired interface was not necessary for the setup, so it was brought down. Two example configuration files are provided in appendix A.1.2.

An instance of an Openflow-enabled virtual switch, which is called data path, was created using the openvswitch tool ovs-dpctl on the command line. Both the wireless and the wired interface were added to this bridge-like interface, since it was intended as a substitution for the Linux bridge. They were both brought up without an IP. In order for the switch to be reachable on the data path interface, an IP address was assigned to it. A route to the subnet where the controller and the RADIUS server are located had to be added through the data path interface.

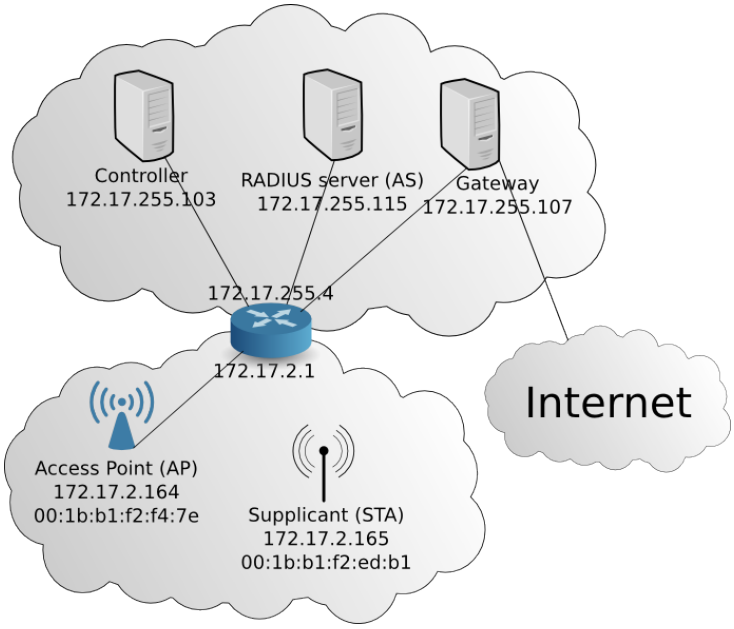

Figure 7: Setup of the initial prototype

It is important to note that while the Openflow daemon, which is called secchan in the present setup, is not yet running, the data path interface is not yet active and the controller is not yet reachable from the access point. Since the wired interface does not have an IP address anymore, it is also not reachable from the controller. This causes the problem that the watchdog which is running on the access points can no longer ping a host on the network, so it assumes that there is a problem with the setup and reboots the node. Having the access point reboot every five minutes due to secchan not running was unacceptable, so watchdog was reconfigured to ping the local loopback interface, effectively disabling it. Similar to the AP, due to the STA not being reachable at all times its watchdog configuration had to be altered in a similar way.
As expected, this prevented the nodes from rebooting, but introduced the problem that a hung-up node now had to be power-cycled manually. However, at that point UCI was not supported for openvswitch setup, so it was impossible to restore a working setup only by rebooting anyway. Manually rebooting it was not a big additional effort compared to manually configuring it. Before running secchan, the directory `/var/run/openvswitch` had to be created. The long-term solution is that when automatic Openflow setup is

supported by UCI, the directory will be created automatically upon configuration of an Openflow data path. By running the simple Openflow controller application that is shipped with openvswitch and connecting to it on the local loopback interface in out-of-band mode, it could be confirmed that secchan was now ready to connect to a controller.

The Openflow controller floodlight was started in its default configuration on the designated virtual machine, acting as a learning switch. On the intermediate router, a firewall rule had to be set up to allow traffic from/to port 6633 for the controller to be reachable from the switch.

Now secchan was started on the Access Points, set up to connect to the remote controller in in-band mode via TCP since SSL was not supported by floodlight. In-band mode is used because the controller is connected via the wired interface which is also part of the data path. As a result, Openflow control traffic is sent on the same interface as data traffic rather than on a separate interface.

6.2.2 Validation

With the setup being completed, a first connection to the AP was attempted, the scan results and the state of the authentication being managed using wpa_cli on the STA, which connects to the running supplicant daemon via the available control interface. Other network configuration tools such as iw were used to display the network state. Packet capturing traces were made with tcpdump on the AP on interfaces wlan0, eth0, mon.wlan0 and dp0. These traces were copied to a notebook PC external to the testbed after each connection attempt. A wireshark plugin provided by Stanford university was installed for decoding Openflow packets. Logging output of the controller and the switch was also observed. Trying to access the local flow table on the switch from the command line using the ovs-dpctl dump-flows dp0 command showed only the currently active flows. Another means of verification was querying the REST API for a list of the connected switches, which were returned as JSON data. Further information could also be obtained by subsequent queries for detailed information on flows and connected devices. The queries were first issued via curl, then using a small script written in python. The script is included in appendix B.1.

The logging output and captured packet trace confirmed that the secchan daemon running on the AP had successfully performed a TCP and Openflow handshake with the controller. It was noted that the first connection attempt often timed out for unknown reasons and only the second one was successful, but openvswitch made several connection attempts and continuously retried in case of failure, so this was not a problem in terms of stability of the setup.

A call to the REST API showed that the list of connected switches now featured the AP's data path ID.
In the Openflow handshake, openvswitch sends its switch capabilities to the controller. For example, the number of tables is set to 2, the maximum number of packets buffered is 256, it cannot reassemble IP fragments or provide queue statistics, but it supports all actions, i.e. output on any port, set MAC address/IP/TCP/UDP source or destination or set/strip VLAN headers. Openvswitch also provided information on its ports, i.e. the managed interfaces being the internal data path, the wired and the wireless interface. After the handshake, the controller sent a flow modification request to delete all flows on the switch. Without any flows, any packet received on one of the interfaces of the AP is sent to the controller as a 'Packet in' Openflow message. The response is always a 'Packet out' message with the instruction to flood the packet to all ports of the AP.

The behavior of the APs was indeed validated to be successful association and authentication. The station was granted access and hostapd unblocked its uncontrolled port when the correct password was used. It also succeeded in pinging first the AP, then the gateway router, then the hosts in the other subnet such as the host on which the Openflow controller was running. An incorrect password resulted in the station being denied access and all non-802.1x frames sent by it were ignored by the AP just as expected. Any non-802.1x traffic from the STA that had been sent to the AP before successfully authenticating was not seen on the wireless interface. As expected, when authenticated the STA can ping all reachable hosts on the network from the client. Its ARP requests were treated like any other frames that have no matching flows: They were sent to the controller as Packet-In and flooded to all interfaces after the controller's response. The response received on eth0 was also first sent to the controller and then flooded, but after this, a Flow modification was received to set up flows for communication between the client and the pinged host. The newly inserted flow rule matches input port, Ethernet source/destination and IP source/destination and outputs packets only on the correct switch port. This causes ICMP messages to be forwarded directly without a Packet-In and Packet-Out event. After some idle time, the flow times out.

6.2.3 Observation

The setup appeared to serve its purpose of having Openflow as a substitution of the linux bridge and being able to manage the AP's flows from the central controller. However, an unexpected phenomenon was observed.

When examining the packet traces, it was noted that all EAP frames had been flooded to the wired network. After arriving on the wireless interface, they were recognized as unknown packets with no matching flows, and sent to the controller in a 'Packet-In' message. The controller responded with a 'Packet-Out' instruction to flood the frame, just like any other packet. It was then forwarded to the local data path and wired interface, producing the expected response frame, which was sent directly on the wireless interface without flooding. RADIUS packets were handled in a similar manner. In spite of being originated locally, they were first handed to the controller and only upon its instruction to flood the packet, they were sent to the RADIUS server on the wired interface, but also on the wireless interface. When the RADIUS server responded, its response was also first sent to the controller as a Packet-In message. After one packet to and one packet from the RADIUS server had been processed, the controller finally sent a Flow Modification, setting up the appropriate output ports for packets in both directions. However, there were no Flow Modifications for EAP frames.

This handling of EAP frames was unexpected. It had been assumed that they are treated differently than other packets. 802.1x frames should only be sent to hostapd, which should handle authentication with the RADIUS server and key negotiation. Flooding these frames, especially session key negotiation, is definitely not a desired behavior. Security risks such as the impact of this on the session's confidentiality have not been examined in further detail since key exchange frames could also be sniffed on the wireless interface by any attacker. Nonetheless it is unnecessary to leak this kind of information to the whole network. While examining the code of the Linux bridge in OpenWRT and comparing it to the openvswitch code, it was found that the commonly used Linux bridge would drop 802.1x frames. It had been hard coded that any frame from a specific type of socket (ETH_P_PAE) should not be forwarded by the bridge, so these packets are handled by hostapd which also listens on that socket. This is specified in an OpenWRT patch on the linux kernel and corresponds to the expected behavior. The openvswitch bridging kernel module does not contain such a rule, so it simply matches 802.1x frames like any other. Since openvswitch currently has no specific focus on wireless networks and focuses on virtualized environments instead, this is not surprising.
For altering this behavior, two possibilities were considered. Firstly, one may patch the kernel module itself, adding a similar rule to the one in the Linux bridge. Secondly, one may proactively insert a flow to the switch which instructs it to drop 802.1x frames. The latter was chosen because it makes use of native capabilities of the Openflow protocol and avoids the complication

of managing a modified openvswitch version.

6.3 Advanced setup

The initially deployed prototype in the fully-controlled testbed environment 'smoketest' is extended to facilitate handover. Figure 8 shows the new setup. The node that had been used as a client so far is now configured as an Access Point following the same steps outlined in the previous setup described on page 6.2.1. Using secchan on their respective data paths, both Access Points connect to the same controller.
Furthermore, the unexpected handling of EAP frames is corrected by inserting static flow rules into the switches.

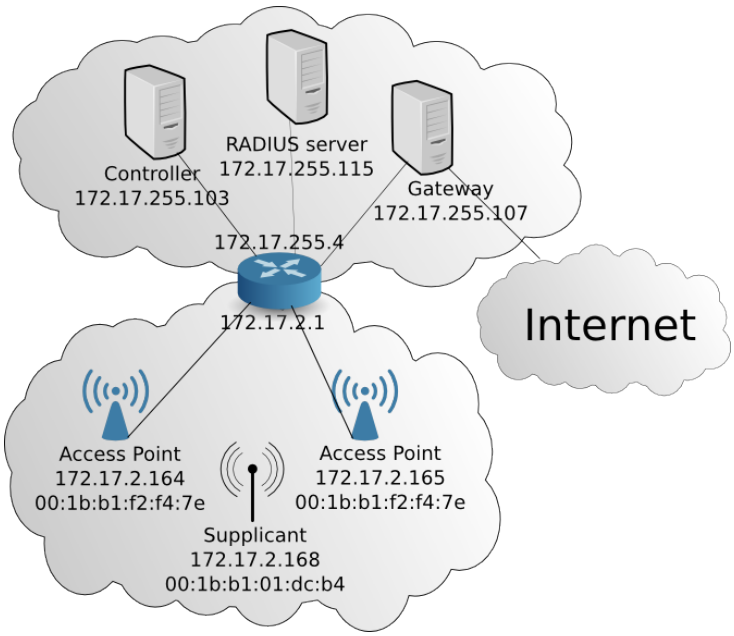

Figure 8: Advanced setup

6.4 Configuration

A third node was added to the setup and configured as the client. Using UCI, the wireless interface was configured in client mode and an IP address was assigned to it similarly to the client configuration in the previous setup. The wired interface stayed up and kept its IP, which was in a different subnet

that the Access Points, but also connected to the core infrastructure. A host route to the machine running the controller was set, so traffic to this host would be routed over the wireless interface. The machine would later be used as a corresponding host to test performance over the wireless network.

Using a python script similar to the one used for querying the REST API for information, a HTTP POST request was made for every inserted flow to every switch connected to the controller. The script is included in appendix B.2 for further reference. 802.1x frames were matched by their Ethertype (0x888e) and were at first dropped in an attempt to simulate bridge behavior. This means that for these packets, no action was specified. Assuming that RADIUS packets are also authentication frames, another flow was added which matched packets with destination port 1812 and specified output on the wired port as action.

The first attempts to connect the client to one of the Access Points failed. They associated without problems, the EAP Request Identity was sent and EAP Response Identity was received on the wireless interface of the AP, but authentication timed out afterwards. After debugging this issue using packet captures and the hostapd log, it was found that there were no RADIUS packets in any of the traces and no mention of it. Instead, the EAP Response Identity frame apparently had not been received by hostapd, so it was impossible to start the RADIUS handshake.

Apparently the EAP Response Identity frame had not been passed through from the wireless interface to hostapd. It was concluded that dropping EAP frames was not the appropriate action. Instead, 'output=local' was specified as the new action for frames whose Ethertype matches 0x888e, so they would be output to the local switch port, the data path interface. The assumption that packets would now be passed to hostapd correctly was supported by the fact that flows destined for the switch itself which had been set up automatically by floodlight always specified this action rather than dropping the packets.
In the next attempt, RADIUS packets were being generated, but authentication still timed out. This was due to a small bug in the Floodlight Static Flow Pusher API: Floodlight received a POST request which contained a flow matching the destination port, but it sent an Openflow Flow Modification matching the source port due to a typo. After fixing this and recompiling the controller, authentication finally worked. The bug was also reported over the mailing list and corrected upstream. However, it is also possible to delete the RADIUS flow rules entirely and treat them as any other IP packets, resulting in the usual flooding. Once a more intelligent behavior for forwarding is implemented on the controller, this may be a good alternative to setting

static rules for controllers. One reason is that one has to hard code the correct output port for RADIUS packets, which is only identified by a number corresponding to the order of addition to the data path. For example, the local port is referenced by 0, the first added interface is referenced by 1, the second one by 2 and so on. If interfaces are added to or removed from the data path, it is possible that this output number changes, so RADIUS packets would be forwarded to the wrong port. Essentially, it is advisable to avoid specific static flow rules for RADIUS packets and just handle them like any other IP packets.

6.4.1 Validation

Once again the behavior of the components was validated using packet capturing traces, logging output and the network testing tools ping and iperf.
Association and authentication to both APs sharing the same SSID was now possible with the appropriate credentials, and using a wrong password resulted in being denied authentication by both APs. EAP frames were no longer flooded, being visible only on the wireless interface and on the AP's local data path interface just as expected. On the data path interface the first RADIUS access request and response were seen as Packet In and Packet Out Openflow messages, but after his, the controller automatically sent an Openflow flow modification message. It contains two flows: One from the RADIUS server to the AP which is output on the local port, and one from the AP to the RADIUS server which is sent to switch port 1, the wired interface. Then RADIUS handshake messages were only seen on these two interfaces and shortly after the handshake, a flow removed message was received. ARP packets were also seen as Openflow Packet-In and Packet-out messages and eventually resulted in a flow modification being added.
However it was noted that when performing a test with wrong credentials after a test with successful authentication, 802.1x authentication would be skipped due to the STA's PMKSA being cached in the AP's hostapd. To delete this Security Assertion, the AP's hostapd had to be restarted. To guarantee that only the correct flows are inserted into the data paths, secchan was also frequently restarted between tests. This process was automated using a shell script which restarted all relevant services, deleted old packet capturing traces and started new ones. One example of such a script can be found in the appendix B.3.
This implies that PMKSAs should be assured to time out after a certain time. However in practice devices that have once been successfully authorized are often unlikely to lose their authorization in the near future, i.e. before the timeout. Consequently, this caching issue was perceived as a testing error

rather than a security flaw.

It was attempted to simulate a handover by bringing down the AP's wireless interface that the STA is currently associated with, forcing the client to switch to the other AP. As expected, after a short time the client associated to the new AP. At the beginning a ping had been started and sent ICMP messages to and from the host where the controller was running over the wireless interface. After the connection to the old AP was cut, it showed a few missed packets but then continued, having found a new path over the new AP.

6.4.2 Observation

In addition to validating that the devices were communicating correctly, a performance test was carried out to observe handover delay.
Using the network measurement tool iperf, an iperf server was started on the virtual machine running the controller. The server opened a datagram socket to receive a constant stream of UDP packets and reported jitter and packet loss every 1 second. After the STA had associated to an AP, an iperf client was started on it and sent UDP packets with 512 kbit/s for 60 seconds. On the server it could be seen that there was no packet loss and jitter was between 1 and 5ms. A handover was then forced by bringing down the current AP's wireless interface. This resulted in no datagrams being received for a certain time, which was measured and recorded in each run of the experiment. The station eventually associated to the next AP and continued sending datagrams, so packet loss dropped to 0% again, marking the end of the delay period. In total, the experiment was conducted 10 times. The median value of the delay in which nothing was received is 7 seconds and the mean value is 9.6 seconds, the minimum value being 4 seconds and the maximum value 36 seconds.
The delay between the last received datagram on the old AP and the first datagram on the new AP could also be observed in the packet capturing traces. The delay between the start of the 802.1x authentication and the sending of the first packet was very short, only a few milliseconds. This means that probably other activities account for the long handover delay, for example scanning delay.

6.5 Final deployment

The setup is adapted for deployment in the 'indoor' testbed of the BOWL network, a network of some APs located on several floors of a building.
A more up-to-date version of openvswitch is used as Openflow implementa-

tion on the APs, namely version 1.4.0 which provides a full-featured switch with a database and userspace management tools. Openvswitch configuration is in the process of being automated using the UCI interface, so that Openflow data paths can be added as a special kind of bridge and all relevant setup measures are taken by a script.
Access points are connected to the controller using a GRE tunnel, so the control channel is out of band in this case. GRE tunnel setup is also automated along with specifying the controller.

7 Conclusion

It is possible to deploy Openflow in a 802.11 network on the wireless Access Points, specifically in the BOWL network. Several switch implementations exist and openvswitch appears to be particularly suited. It can run as replacement for a regular Linux bridge and is compatible with hostapd usage. As a 802.11 data frame, 802.1x frames are is handled by Openflow switch and have to be forwarded to the local data path interface to be received by hostapd. It is advisable to set up a static Openflow rule for this, so they are received by hostapd, but not leaked to the wired network. Authorization is still working, since unauthorized non-802.1x packets from wireless stations arriving on the wireless interface are dropped automatically thanks to hostapd. It is not necessary to set up an Openflow rule for this purpose when hostapd is running.

Deploying Openflow in a wireless network now has the benefit that traffic can be sliced just like in a wired network. Traffic can be matched by any header field and custom routing protocols can be developed. Also, dynamic VLAN tagging is possible. One can also monitor the number of devices communicating over each AP from the controller. However, load balancing is only possible in the core network, since in this setup Openflow does not influence 802.11 management traffic so it cannot send disassociation frames.

It would be conceivable to develop an application or module for the Openflow controller that handles some of the hostapd functionality such as 802.1x authentication using credentials on an arbitrary server. This would also require changes to the AP, e.g. patching hostapd to handle only key negotiation. Another possibility is centralizing most 802.11 functionality and having the AP only forward frames, while association and authentication are fully handled by the controller. However, this is a great effort since it requires changes to the Openflow standard and its implementation in order to be able to match and directly forward the raw 802.11 frames to the controller.

References

[1] M. S. Bargh, R. J. Hulsebosch, and E. H. Eertink. Fast authentication methods for handovers between ieee 802.11 wireless lans. In *IEEE 802.11 Wireless LANs, Proceedings of the 2nd ACM international workshop on Wireless mobile applications and services on WLAN hotspots*, pages 51–60. ACM Press, 2004.

[2] TU Berlin. Berlin open wireless lab. http://www.bowl.tu-berlin.de/, 2012.

[3] Floodlight. A java-based openflow controller. http://floodlight.openflowhub.org, 2012.

[4] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *Computer Communication Review*, 38(3):105–110, 2008.

[5] IEEE Computer Society. IEEE standard for information technology — telecommunications and information exchange between systems — local and metropolitan area networks — specific requirements — part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, June 2007.

[6] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *In Proc. OSDI*, 2010.

[7] Jouni Malinen. Ieee 802.11 ap, 802.1x authenticator. http://hostap.epitest.fi/hostapd/, 2012.

[8] Jouni Malinen. Linux wpa/wpa2 ieee 802.1x supplicant. http://hostap.epitest.fi/wpa_supplicant/, 2012.

[9] Nick Mckeown, Scott Shenker, Tom Anderson, Larry Peterson, Jonathan Turner, Hari Balakrishnan, and Jennifer Rexford. Openflow: Enabling innovation in campus networks, 2008.

[10] Arunesh Mishra, Min ho Shin, and William A. Arbaugh. Context caching using neighbor graphs for fast handoffs in a wireless network, 2003.

[11] Rohan Murty, Jitendra Padhye, Alec Wolman, and Matt Welsh. Dyson: An architecture for extensible wireless lans, 2009.

[12] Jon olov Vatn. An experimental study of ieee 802.11b handover performance and its effect on voice traffic, 2003.

[13] Openvswitch. A production quality, multilayer open virtual switch. http://openvswitch.org, 2012.

[14] OpenWRT. A linux distribution for embedded devices. http://openwrt.org, 2012.

[15] Sangheon Pack and Yanghee Choi. Fast inter-ap handoff using predictive authentication scheme in a public wireless lan. In *In Proceedings of IEEE Networks Conference (confunction of IEEE ICN and IEEE ICWLHN*, 2002.

[16] Sangheon Pack and Yanghee Choi. Pre-authenticated fast handoff in a public wireless lan based on ieee 802.1x model. In *IEEE 802.1x Model," IFIP TC6 Personal Wireless Communications 2002 (To Appear*, pages 175–182, 2002.

[17] Dorothy Stanley, Michael Montemuro, and Pat Calhoun. Control And Provisioning of Wireless Access Points (CAPWAP) Protocol Specification. RFC 5415, 2009.

[18] The UCI System. Unified control interface. http://wiki.openwrt.org/doc/uci, 2012.

[19] Stanford OpenFlow Team. Openflow switch specification version 1.1.0, 2011.

[20] Kok K. Yap, Rob Sherwood, Masayoshi Kobayashi, Te Y. Huang, Michael Chan, Nikhil Handigol, Nick McKeown, and Guru Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *Proceedings of the 2th ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, VISA '10, pages 25–32. ACM, 2010.

[21] Kok-Kiong Yap, Yiannis Yiakoumis, Masayoshi Kobayashi, Sachin Katti, Guru Parulkar, and Nick McKeown. Separating authentication, access and accounting: A case study with openwifi. 2011.

A Example configuration files

A.1 Initial prototype

A.1.1 Access Point

```
config wifi-device  radio0
        option type     mac80211
        option channel  11
        option macaddr  00:1b:b1:f2:f4:7e
        option hwmode   11ng
        option htmode   HT20
        list ht_capab   SHORT-GI-40
        list ht_capab   TX-STBC
        list ht_capab   RX-STBC1
        list ht_capab   DSSS_CCK-40
        # REMOVE THIS LINE TO ENABLE WIFI:
#       option disabled 1

config wifi-iface
        option device   radio0
        option mode     ap
        option ssid     smoketest-24.theresa.bowl

# PSK Mode
#       option encryption psk2
#       option key      maunzmaunz

# 802.1x mode
        option encryption wpa2/tkip+aes
        option server   172.17.255.115
        option key      testing123
```

A.1.2 Station

```
config wifi-device  radio0
        option type     mac80211
        option channel  11
        option macaddr  00:1b:b1:f2:ed:b1
        option hwmode   11ng
        option htmode   HT20
        list ht_capab   SHORT-GI-40
```

```
        list ht_capab   TX-STBC
        list ht_capab   RX-STBC1
        list ht_capab   DSSS_CCK-40
        # REMOVE THIS LINE TO ENABLE WIFI:
#       option disabled 1

config wifi-iface
        option device   radio0
option mode     sta
        option ssid     smoketest-24.theresa.bowl

# WPA Preshared Key
#       option encryption psk2
#       option key      maunzmaunz

# 802.1x RADIUS
       option encryption wpa2/tkip+aes
       option eap_type peap
       option auth     mschapv2
       option identity theri@bowl.tu-berlin.de
       option password *********
```

B Scripts

B.1 Querying the Floodlight REST API

```
#!/usr/bin/python

import sys
import json
import urllib
import pprint

queries = {'switches':'wm/core/controller/switches/json', 'aggregate':
'wm/core/switch/all/','perswitch':'wm/core/switch/', 'counters':
'wm/core/counter/', 'memory':'wm/core/memory/json'}

stats = ['port', 'queue', 'flow', 'aggregate', 'desc', 'table',
 'features', 'host']
```

```
def query_switches(controller):
        return json.loads(urllib.urlopen(controller +
queries['switches']).read())

def query_memory(controller):
        return json.loads(urllib.urlopen(controller +
queries['memory']).read())

def query_stats(controller, switch):
        result = {}
        for stat in stats:
                print "Query " + switch + " for " + stat
                result[stat] = json.loads(urllib.urlopen(controller +
queries['perswitch'] + switch + '/' + stat + '/json').read())
        return result

def query_aggregate(controller):
        result = {}
        for stat in stats:
                print "Query for aggregate " + stat
                result[stat] = json.loads(urllib.urlopen(controller +
queries['aggregate'] + stat + '/json').read())
        return result

if __name__ == "__main__":

        controller='http://172.17.255.103:8080/'
        switches = query_switches(controller)
        print switches

       for s in switches:
               print "Switch: " + s["dpid"]
               stats = query_stats(controller,s["dpid"])
               print "Done querying " + s["dpid"]

        pp = pprint.PrettyPrinter(indent=2)
        pp.pprint(stats)
```

B.2 Using Floodlight's Static Flow Pusher API

```
#!/usr/bin/python
import sys
import json
import urllib
import pprint

flows = [{ # 802.1x from wireless to local
                'switch':'0',
                'name':'8021x-',
                'ether-type' : '0x888e',
                'active' : 'true',
                'actions': 'output=local'},
        { # RADIUS to server
                'switch':'0',
                'name':'torad-',
                'dst-port' : '1812',
                'active' : 'true',
                'actions': 'output=1'},
        { # RADIUS from server
                'switch':'0',
                'name':'frorad-',
                'src-port' : '1812',
                'active' : 'true',
                'actions': 'output=local'},
]

queries = {'switches':'wm/core/controller/switches/json', 'push':
'wm/staticflowentrypusher/json', 'perswitch':'wm/core/switch/'}

stats = ['port', 'queue', 'flow', 'aggregate', 'desc', 'table',
'features', 'host']

def addflow(params, controller="http://172.17.255.103:8080/"):
return urllib.urlopen(controller +
queries['push'], params).read()

def query_switches(controller="http://172.17.255.103:8080/"):
return json.loads(urllib.urlopen(controller +
queries['switches']).read())
```

```
def query_stat(switch, stat, controller="http://172.17.255.103:8080/"):
        result = {}
        print "Query " + switch + " for " + stat
        result[stat] = json.loads(urllib.urlopen(
controller + queries['perswitch'] + switch + '/' + stat + '/json').read())
        return result

if __name__ == "__main__":

switches = query_switches()
print switches

# Static parameters for the new flow
ret = ''
for switch in switches:
for flow in flows:
flow['switch'] = switch['dpid']
flow['name'] += switch['dpid']
print "Adding flow: " + json.dumps(flow)
ret = addflow(json.dumps(flow))
```

B.3 Script for automating the test

```
#!/bin/ash
# Script for test 2 on alix-24

/sbin/wifi
ovs-dpctl add-if dp0 wlan0
killall secchan
killall tcpdump
cd ~/pcap
rm *.pcap
rm *log*
tcpdump -i wlan0 -s 0 -w test-24-wlan0.pcap&
tcpdump -i dp0 -s 0 -w test-24-dp0.pcap&
tcpdump -i eth0 -s 0 -w test-24-eth0.pcap&
secchan dp0 tcp:172.17.255.103 --log-file=secchanlog-24 -v&
```