SEACAT: AN SDN END-TO-END CONTAINMENT ARCHITECTURE

by

Makito Kano

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computing

School of Computing

The University of Utah

August 2015

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Makito Kano**

has been approved by the following supervisory committee members:

Jacobus van der Merwe	, Chair	**May 21, 2015** Date Approved
Sneha Kasera	, Member	**June 12, 2015** Date Approved
Eric Eide	, Member	**June 13, 2015** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Healthcare organizations heavily rely on networked applications. Many applications used in healthcare settings have different security, privacy, and regulatory requirements. At the same time, users may use their devices with medical applications for non-medical-related purposes. Running arbitrary applications on the same device may affect the healthcare applications in a way that violates their requirements. The ability of using the same device for multiple purposes in an enterprise network presents a challenge to healthcare IT operations. To allow the users to use the same device for both medical and non-medical-related purposes while meeting the set of requirements for medical applications, we present the design and implementation of the SeaCat, an SDN End-to-end Application Containment ArchitecTure, and evaluate the system in a testbed environment. SeaCat has two major components. First is the container technology used in the client device to securely isolate any application. Second is the software-defined networking (SDN) that provides isolated secure network resource access for each application.

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **vi**

LIST OF TABLES . . . . **xi**

ACKNOWLEDGEMENTS . . . . **xii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Motivation 1
1.2 Threat Model 2
1.2.1 Threats and Assumptions on the Client Device 2
1.2.2 Threats and Assumptions on the Enterprise Network 2
1.3 Thesis Statement and Approach 3

2. SECURITY IN HEALTHCARE **4**

2.1 Threats and Regulations 4
2.2 Protecting Security and Privacy 4
2.3 BYOD Trend 5

3. SEACAT ARCHITECTURE **6**

3.1 End-to-end Context 6
3.1.1 Network Context (Forwarding Rules) 7
3.1.2 Network Context (Resource Prioritization) 7
3.1.3 Containers on Client Device 7
3.1.4 802.1X and 802.11i 7
3.1.5 Virtual AP (VAP) 8
3.2 Workflow to Create Contexts 8

4. IMPLEMENTATION **13**

4.1 Container Isolation 13
4.2 Client Hardware Resource Guarantee 15
4.3 Window System Isolation 15
4.4 Networking Configurations 16
4.5 Network Resource Allocation 16
4.6 Workflow with Single Sign On 17
4.7 Securing the Link Between Client and AP 19

5. SECURITY ANALYSIS **28**
5.1 Attacks 28
5.2 Solutions 29
6. EVALUATION **31**
6.1 Network Resource Prioritization 31
6.1.1 How Queues Work 31
6.1.2 Solution 32
6.1.3 Bandwidth Allocation Algorithm 33
6.1.4 Experiment Setup 34
6.1.5 Results 35
6.1.6 Contributions 39
6.2 Hardware Resource Guarantee 40
6.2.1 Experiment Setup 40
6.2.2 CPU Core Allocation 40
6.2.3 CPU Bandwidth Allocation 41
6.2.4 Memory Allocation 42
6.3 Evaluation of Namespace Isolation 42
6.3.1 IPC Namespace 43
6.3.2 Network Namespace 44
6.3.3 Mount Namespace 44
6.3.4 PID Namespace 44
6.3.5 User Namespace 45
6.3.6 UTS Namespace 46
7. FUTURE WORK **62**
7.1 Trusted Platform Module (TPM) 62
8. RELATED WORK **65**
8.1 End-Point Isolation 65
8.2 End-Point SDN 66
8.3 Resource Allocation 66
8.4 Remote Attestation 67
9. CONCLUSION **68**
APPENDIX: PRIORITY FUNCTION ALGORITHM **70**
REFERENCES **73**

LIST OF FIGURES

3.1 SeaCat architecture. Protected application and default applications (user's own and potentially malware) are isolated. SDN-enabled switch sets policies in a way that extends the enterprise security context into the device. Trusted Daemon is responsible for managing the isolation. It also communicates with SeaCat server to tie the two network domains together. The user uses the default context from the default application to authenticate and create the secure context. Upon successful authentication, SeaCat server creates the secure context in the network. Protected data, non-protected data, and a gateway to the public internet exist in the network. 10

3.2 Example of network context. Policies, or rules, applied to the SDN-enabled switch create the logical isolation of the two connections. Queuing mechanism of the switch is used to prioritize the traffic. 10

3.3 Start of the workflow to create the contexts. At the beginning, only the default context exists in the network. 10

3.4 Client device connects to the network with default context. 11

3.5 Step 1: Client uses the default context to access the authentication system. . . 11

3.6 Step 2: SeaCat server creates the secure context for this client in the network. 11

3.7 Step 3: SeaCat server requests the TD to create the context in the device. . . . 11

3.8 Step 4: TD creates the context in the client device. Step 5: TD starts the protected application. 12

4.1 Implementation of the client device. There are two containers. One is for default applications and the other is for each medical application. A separate context is created for each container and separate VIF is assigned to each context. In this case, blue represents the context for the default container and green represents the context for the application container. In addition, TD manages the containers and the OVS on the device. 21

4.2 SDN enabled Wi-Fi AP is the gateway for the client device to the enterprise network. Separate contexts are created in the network as well. As in the client device, blue represents the default context and green represents the secured context. The user is authenticated with the authentication system. Then the SeaCat server requests the controller to create the context in the network and the AP. 21

4.3 Implementation of containers on client device. Separate container is prepared for each medical application. Other applications and jobs that exist in the host in default are contained in the default container. TD runs in the root namespace. Separate set of policies are created in the OVS for each container. 22

4.4 Each container is mounted at different mount point (e.g., `/containers/1`, `/containers/2`). Overlay filesystem is used so that containers can share the necessary system files from the underline kernel. Although they are shared, they cannot be modified from inside the container because they are read only. If they are modified, the changes are kept in the `upper` directory of each container. In Linux, jobs are stored in `/etc/init`. We copy necessary jobs from `/etc/init` (root) to each container's `init` directory. 22

4.5 Red boxes represents containers. The application container, the default container, and the root namespace have different sets of jobs to create the appropriate environment. In addition to create/destroy, start/stop the containers, TD runs the Python program to communicate with SeaCat server and runs the SDN controller. 23

4.6 `Cgroups` allocates hardware resources to each container. We have tested that it can flexibly allocate CPU core, CPU bandwidth, and memory. 23

4.7 Thick black box represents the Xnest isolation of the containers. To display the image, Xserver of Xnest sends image to the Xserver of the root. Dotted arrows represent removed socket. In this diagram, the default container initially had access to the UDS of the application container and the root, which could be the attack vector. 24

4.8 An example of using queues. Queues configured in client device, AP, and the enterprise network impose the maximum bandwidth of medical and non-medical traffic to each direction. 24

4.9 SSO steps that involve client, SP, IdP, and the controller. SP protects the access to a resource. IdP is responsible for checking the integrity of the client. Controller creates the policy enforced context upon request from the SP. 25

4.10 SSO steps that shows our contribution. Step 2 to 9 (grayed out) are the standard Shibboleth procedures. Instead of delivering a webpage, our SeaCat program requests the controller and the TD to create context. This shows that Shibboleth can be integrated into SeaCat. Because Shibboleth is widely used open source software, it proves SeaCat's feasibility in terms of SSO. 26

4.11 High-level steps that combine 802.1X (authentication), 802.11i (encryption), and our context. 26

4.12 The architecture that improves the security of the Wi-Fi link. The AP uses the VLAN tag to differentiate the traffic. OVS checks the tag to send packets to the appropriate destination. We combine 802.1X and 802.11i to our solution. 27

6.1 Implemented instance of SeaCat. Client laptop has two containers and the TD. It is connected to the SDN-enabled AP through virtual wireless interfaces. The emulated network laptop runs Mininet to emulate the enterprise network. The controller laptop runs the SDN controller for the enterprise network. 47

6.2 Topology with two servers and two clients. Queues are configured in `s1-eth1`. 47

6.3 Three queues in `s1-eth1` are indicated as Class 1, 2, and 3. Width of the arrow in the bottom of the queue indicates the allocated bandwidth. A colored marble represents a packet. Flow entries direct incoming packets to appropriate queue. Packets are sent out at different rates. 47

6.4 Assume that the algorithm favors the application traffic. Initially, larger traffic is generated from the default hosts; hence, larger number of marbles (packets) are going through the default queue. When the application traffic load increases, the allocated bandwidth is transferred to it, indicated by narrower arrow for the Default queue and wider arrow for the App queue. If both queues have large traffic load, larger bandwidth is allocated to more important traffic. 48

6.5 Initially, larger bandwidth is allocated to the App queue. When the default traffic load starts to catch up, bandwidth is transferred from the App to the Default queue. Although the application traffic is considered more important, the algorithm favors the queue with lower traffic load and likely to transfer available bandwidth to it if it needs more. 48

6.6 Sample priority function for Queue 1 at State 1 (before bandwidth transfer). Numbers indicate the area under the curve, which are the current priority and the prospective priority. Arrow indicates the current throughput. 49

6.7 Sample priority function for Queue 2. 49

6.8 Sample priority function for Queue 3 at State 1 (before bandwidth transfer). . 50

6.9 Sample priority function for Queue 1 at State 2 (after bandwidth transfer). . . 50

6.10 Sample priority function for Queue 3 at State 2 (after bandwidth transfer). . . 50

6.11 Enterprise topology created with Mininet for the experiment. Access switches are connected to the core switches. Edge switch works as the gateway to the Internet. Each client and server represent a set of hosts. Queues are configured in access switch and Wi-Fi APs. 51

6.12 Scenario for Experiment 1. Default Client and Medical 1 Client download at the same time. Default downloads from the server in the Internet while Medical 1 downloads from Medical 1 server in the network. Queues are configured in Queues2 port. 51

6.13 Throughput transition for Experiment 1. It shows the equilibrium between Class 1 and Default. 51

6.14 Priority functions for Default, Class 1, and Class 2 traffic. The ratio of the priority between the Medical 1 ($1500/x$) and the Default ($750/x$) is 2 to 1. . . . 52

6.15 Priority functions for Default, Class 1, and Class 2 traffic. The ratio of the priority between the Class 1 ($1500/x$) and the Default ($50/log(x)$) varies based on the available bandwidth. 52

6.16 Scenario for Experiment 2 and 5. Default, Class 1, and Class 2 upload at the same time. Queues are configured in Queues1 and Queues3 port. 53

6.17 Throughput transition for Experiment 2, which achieves the similar result as WFQ during the equilibrium, or the time when there is no bandwidth re-allocation or transfer. The Class 1 traffic receives twice more bandwidth than the Default traffic at all of the three equilibrium. 53

6.18 Throughput transition for Experiment 2, which achieves the variable ratio between the Class 1 and the Default. During the first two equilibrium period, the ratio is 2 to 1. However, during the third equilibrium period, the ratio is 5 to 4. 54

6.19 Scenario for Experiment 3. Class 1 and 2 download at the same time. Queues are configured in Queues4 port. 54

6.20 Throughput transition for Experiment 3. It shows the equilibrium between Class 1 and Class 2. 55

6.21 Scenario for Experiment 4. Default, Class 1, and Class 2 download at the same time. Default and Class 1 traffic are managed at Queues2 port. Class 1 and Class 2 traffic are managed at Queues4 port. Different queue configurations are applied to the two ports. 55

6.22 Throughput transition for Experiment 4. It shows how Class 1 affects both Default and Class 2 throughput. Default and Class 1 reach their equilibrium when Class 2's throughput goes down. 56

6.23 Throughput transition for Experiment 5. It shows the transition when three classes share the same port. 56

6.24 Scenario for Experiment 6. DoS attack traffic and Medical 2 traffic from the valid user arrive to Medical 2 server at the same time. Queues in Queues5 port allocate enough bandwidth to Medical 2 traffic. 57

6.25 Throughput transition for Experiment 6. It shows how queues can be used to prevent DoS attack. 57

6.26 Throughput transition for Experiment 7, which uses TCP. It shows similar result as in Experiment 1. There is more fluctuation than UDP, but the equilibrium is reached on average. 58

6.27 Priority function for $50/x$. Use the section area under the curve around the maximum bandwidth to find the function that achieves the desired equilibrium. 58

6.28 Priority function for $100/x$. 59

6.29 Separate cores are allocated to separate containers. `Cgroups` handles the allocation. `stress` spawns threads in the default container to repeatedly call `sqrt()` to exhaust the CPU resource. At the same time, `sysbench` in the application container measures the speed to calculate the prime numbers. 59

6.30 Containers share the same core. Same experiment as in the case when the cores are not shared. 59

6.31 Experiment results of CPU core allocation. Blue bars with horizontal stripes show the calculation time without another container. Orange bars with tilted grid show the calculation time when separate cores are allocated. Red solid bars show the time when cores are shared. (e.g., 2 Core N means separate two cores, with total of four cores, are allocated to each container. 2 Core S means two containers share the same two cores.) Sharing the cores with other container significantly impacts the calculation time. 60

6.32 Experiment results of CPU bandwidth allocation. Blue bars with horizontal stripes show the calculation time without another container. Orange bars with tilted grid show the calculation time when the two containers are allocated the same share of bandwidth while `stress` is running on the default. Red solid bars show the time when twice more bandwidth is allocated to the application container. Allocating larger CPU bandwidth significantly reduced the calculation time. 60

6.33 Memory resource is shared between the containers. In the default container, `stress` spawns threads to call `malloc()` that tries to obtain 2GB of memory. While it keeps the memory, `/proc/meminfo` was read from the application container. 61

6.34 Experiment result of memory allocation. Blue bar with horizontal stripes shows the default free memory in the application container. Orange bar with tilted grid shows the free memory when the `stress` in the default container tries to obtain 2GB. Red solid bars show the free memory when `Cgroups` imposes the maximum bandwidth. For example, the leftmost red bar shows the case when `Cgroups` imposed 1.5GB limit on the default container. In this case, `stress` can only obtain up to 1.5GB memory. 61

7.1 TPM checks the integrity (i.e., calculates the hash and sign) of BIOS, kernel, TD, application container, and the application. Client sends the signature to the SeaCat Server to authenticate. SeaCat server stores the files so it can check the signature. When the signature is validated, it signals the client to start the application. Finally, SeaCat server sends the signal to create the policy enforced context. 64

7.2 TrouSerS is the software stack that communicates with the TPM. It provides the interface, which can be used by tpm-tools commands and C programs. `tpm-tools` commands are used to configure the TPM and C programs are used to sign and encrypt. 64

LIST OF TABLES

6.1 Summary of CP and PP values calculated from the sample functions. 48

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor, Dr. Kobus van der Merwe; research staff of the Flux research group; David Johnson; and my project partner, Junguk Cho, for the constant guidance, motivation, and support.

I thank my committee members, Eric Eide and Sneha Kasera, for their suggestions and feedback, which have helped me to improve this work.

This material is based upon work supported by the National Science Foundation under Grant No. 1343713.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Healthcare professionals use many kinds of medical applications including accessing patient records, remote diagnosis and consultations, in-home patient monitoring, healthcare-related analytics, and even remote surgical procedures. These applications are often used from the same device, which could also be used to run non-medical-related applications such as web browsers and games. This diversity of applications and devices that contain multiple applications present a particular challenge to healthcare IT operations. Healthcare organizations face many stringent regulatory, privacy, and security requirements. For example, in-home patient monitoring systems have different privacy and security requirements compared to similar patient monitoring systems that are used within a hospital. Although a web browser does not have any requirements for accessing non-medical-related websites, it should not be able to observe or interfere with any privileged application that have more stringent requirements. In addition, the medical applications in those devices operate on the networks that are under the organization's control.

To address these problems, we propose the SeaCat architecture. It ensures both the security and the resource for healthcare applications in end-to-end fashion. Our approach uses end-point application containment to securely isolate medical applications. Our approach also involves creating a network context in an enterprise network with SDN and extending SDN primitives into end-points. With this approach, we ensure that medical applications are not affected by other applications in the same device. In addition, the network resources are properly isolated and the client device's hardware resources are guaranteed.

1.2 Threat Model

In this section, we provide a high-level overview of the threat model.

1.2.1 Threats and Assumptions on the Client Device

Medical applications must operate on somewhat insecure device where malware could be installed in the same machine. The first threat that SeaCat defends against is malware that is running on the same device as a medical application. SeaCat prevents malware that runs on the same device from stealing sensitive data. The second threat is a DoS attack from the malware in the client device or on the network. Since multiple applications run simultaneously, the entire network bandwidth of the client device may be consumed by the malware. SeaCat prevents this attack by allocating enough network resource for the medical application. The third threat is that non-medical applications (including malware) exhaust the hardware resources such as memory and CPU bandwidth. SeaCat prevents this by allocating enough resources to the medical application. The fourth threat is that the user accidentally copies and pastes sensitive medical data to non-medical applications or malware obtains access to the image display or key strokes. In a Linux environment, SeaCat prevents this by isolating the window system of the medical application.

The fifth threat, which the SeaCat architecture can protect against, is that either hardware, kernel, container, or the medical application is compromised. In this case, nothing in the client device can be trusted. SeaCat can use remote attestation to check the integrity of the device. The solution is described in the Future Work chapter.

1.2.2 Threats and Assumptions on the Enterprise Network

All applications on the client device operate in a hospital enterprise network. Although it is somewhat more secure than the public Internet, we assume the same threat level as using the public Internet where any user can attempt to access the network to eavesdrop or cause a DoS attack. We assume that the wired part of the enterprise network, such as switches and Ethernet links, are better secured than the wireless (Wi-Fi) link. We assume that the medical application server, authentication system, and SDN controller are not compromised. The threat that SeaCat defends against is heavy traffic load on the enterprise network that disrupts the medical application traffic. Such traffic can be intentionally generated by malware or unintentionally generated by legitimate users who download a large file. In addition, SeaCat's authentication system prevents unauthorized access to the medical application server.

1.3 Thesis Statement and Approach

Our thesis is that isolation mechanism on both end-point devices and a software-defined networking can be combined to isolate and secure sensitive data and applications through end-to-end containment in an enterprise network.

We demonstrate that SeaCat provides end-to-end application isolation for a client device and a network that can provide isolation and resource prioritization for the connection. SeaCat can provide security through container isolation within a single device. On the end point, we use container technology to isolate medical application from malware and other non-medical-related applications. SeaCat can prioritize network resources for each application. We use SDN in both the enterprise network and on the client device to prioritize network resources for medical application traffic.

To achieve these goals, we developed a system which uses:

- Linux container [28] (LXC) technology to isolate filesystem and processes between applications.
- LXC and `Cgroups` [7] to guarantee hardware resources for each container.
- Xnest [32] to isolate X window system between containers.
- OpenvSwitch [37] on client device to enforce policies that are consistent with the policies on the enterprise network.
- Single Sign On to authenticate users to create policy enforced context.
- OpenFlow [36] to allocate appropriate bandwidth to prioritize network resources.

We evaluated these features by:

- Showing that protected data cannot be stolen by malware.
- Showing that both network and hardware resources for protected applications are appropriately allocated.

CHAPTER 2

SECURITY IN HEALTHCARE

2.1 Threats and Regulations

Securing healthcare data is critical for both patients and healthcare organizations. In 2014, Chinese hackers broke into one of the largest U.S. hospital operators, Community Health Systems Inc., to steal the personal information of 4.5 million patients [22]. Stolen medical information are typically used to purchase drugs and medical equipment. They are also used to file made-up claims to obtain insurance payment. Protecting medical information is challenging since stolen medical data are usually not immediately identified because the criminals do not use them right away like credit card numbers.

HIPAA (Health Insurance Portability and Accountability Act) was endorsed by the U.S. Congress to protect healthcare data [45]. It regulates the use and disclosure of an individual's health information by covered entities. Covered entities include healthcare providers (e.g., doctors, clinics, psychologists), health insurance companies, and healthcare clearinghouses. Thus, for healthcare organizations, security of healthcare data is critical from a legal perspective too.

2.2 Protecting Security and Privacy

Previous work has identified five methods to protect the security and the privacy of medical records [13].

- Standards and regulations.
- Encryption algorithms.
- Access control or role-based access.
- Recording the communications on EHR (Electronic Healthcare Record) system.
- Human resource security; educating employees to recognize the importance of security and privacy.

In the scope of healthcare security, there are many ongoing research efforts in the field of access control [14, 1, 8]. More medical records are handled electronically and shared among entities who need them. Those entities include doctors and healthcare organizations. Defining the access control policies and designing the infrastructure for EHR-related communications that abide by those policies are becoming important [5].

The approach of SeaCat to protect the security and the privacy is different from all of the above five methods. While it is different, it also enhances the effect of each previous method. Because SeaCat isolates medical application and other non-medical-related applications, it makes users less prone to leak the secure data. The isolation assists both users and hospital system administrators to comply with the regulations. It also frees users who are not security expert from worrying about the security while they use medical applications. In addition, because SeaCat requires users to authenticate before using a medical application, it makes access control simpler. Since SeaCat imposes the security to user's personal device, it will become even more useful due to the recent BYOD trend.

2.3 BYOD Trend

Bring Your Own Device (BYOD) is the trend that employees use their own personal device for work. There are two parts that must be improved before incorporating BYOD in healthcare organizations [34]. The first part is the lack of security in the existing BYOD system. A survey shows that 81% of healthcare organizations allow their employees to store patient data on their mobile device, while 49% of the 81% do not provide any security for those data [30]. Because no security is incorporated to the device, the users are fully responsible for all of their operations. In other words, users should always be careful not to breach any security or privacy regulations when they use their personal device. The second part is the lack of policy for BYOD. The same survey shows that 71% of businesses with BYOD have no official policies. When medical data are stolen from an employee's device, the organization is subject to hefty fines.

In the case of BYOD, the methods listed in the previous section would not protect the security and privacy effectively. If the client device is compromised, the methods to protect data such as encryption and access control may not function as intended. Because SeaCat is the system that runs in the lower level than these methods, it adds security while all the existing methods can still be implemented and function at a higher layer.

CHAPTER 3

SEACAT ARCHITECTURE

Our assumed environment is an enterprise healthcare network where healthcare professionals bring their portable client devices, which uses Wi-Fi to connect to the network. They use these devices to run medical applications as well as applications for non-medical purposes. The first goal of SeaCat is to securely incorporate semi-trusted personal (unmanaged) client devices into a managed enterprise network. SeaCat provides a means for an enterprise network to ensure that it is safe to allow personal client devices to connect to the secured resource. The second goal is to provide enough network and hardware resources to run the medical applications.

Figure 3.1 shows the architecture that achieves these goals. Our system consists of two major components; client device and enterprise network. The client device is potentially the user's personal device, which from an IT perspective cannot be trusted. Therefore, protected (i.e., medical) and default (i.e., non-medical and potentially malware) applications must be isolated. The containment technology and the SDN-enabled switch on the client device not only isolates the traffic for medical and non-medical applications, but also allows separate policies to be applied to each.

In addition to the two sets of applications, there must be a trusted process, which we call Trusted Daemon (TD). The first role of TD is to handle the isolation of the applications. It manages isolation mechanism and medical applications. The second role is to communicate with the SeaCat server to provide appropriate policies to the client device. Before the client device connects to the enterprise network, the policies in the network are managed by the SeaCat server while the policies in the client device are managed by the TD. Thus, after the client device is connected, it is necessary to manage the policies of the two domains together so that there are consistent, end-to-end contexts.

3.1 End-to-end Context

Context is our way of representing the isolation and the resource prioritization applied to secure and default applications as well as the traffic generated from them. We use the following components to create the end-to-end contexts.

3.1.1 Network Context (Forwarding Rules)

Network context provides the isolation of traffic with forwarding rules applied to the SDN-enabled switches. In an SDN-enabled switch, forwarding rules determine which output ports the incoming packets are delivered. Figure 3.2 shows a simple example of SDN-based context. The first forwarding rule in the switch says that packets arriving on port P0 with source IP address of 10.0.0.1 and the destination IP address of 10.0.0.3 must be sent out to the port P2. Similarly, packets from Medical Client that are heading to Medical Server will be sent out to port P3. Default (or non-Medical) Client is not allowed to communicate with Medical Server (and Medical Client is not allowed to communicate with Default Server) because there is no rule that realizes such connections. Hence, the rules provide the logical isolation of connections. The connections are logically isolated in a sense that even though all hosts are physically connected to all other hosts, the rules in the switch work as if two logical links exist to isolate the two connections.

3.1.2 Network Context (Resource Prioritization)

The network context also provides the prioritization of traffic on different contexts. To achieve this, we use the queuing mechanism of SDN-enabled switches. In Figure 3.2, the box with Q2 and three arrows represents the queue that is applied to the outgoing port. As in the forwarding rules, we apply bandwidth allocation rules to the switch to prioritize the traffic. In this example, maximum bandwidth of 10 Mbps is applied to Default Client-Default Server traffic and 30 Mbps is applied to Medical Client-Medical Server traffic. We dynamically change the maximum bandwidth to improve the bandwidth utilization. Our resource prioritization algorithm and the experiment results are described in the Evaluation chapter.

3.1.3 Containers on Client Device

We use Linux Containers (LXC) to isolate medical applications from other applications within the client device. LXC is also used to allocate hardware resources to each context. LXC is described more in detail in Section 4.5.

3.1.4 802.1X and 802.11i

We use 802.1X to authenticate the client when associating to the Wi-Fi AP and 802.11i to encrypt the Wi-Fi traffic. Detailed description of how these two protocols are used is described in Section 4.7.

3.1.5 Virtual AP (VAP)

VAPs create the isolation of Wi-Fi traffic in the similar way as the Ethernet VLAN over the air. This is described more in detail in Section 4.4.

There are three major parts that separate the default and the secure context. The first part is the isolated environment realized by a container on the client device where applications would run. The main purpose of this environment is to isolate default applications from the protected application in the same device, and vice versa. Another purpose is to allow the network administrator to create the custom environment to run default applications or a protected application.

The second part is the network context, which include the forwarding rules and the resource prioritization feature, on the switch of the client device and the switches in the network. The forwarding rules are set so that the traffic for the default applications can only reach the authentication server, the gateway to the Internet, and the non-healthcare-related resources. The rules are also enforced for the traffic of the protected application so that it can only reach the protected data server. The resource prioritization feature prioritizes the network bandwidth based on the type of application that the traffic is generated from. Although the specific configurations are up to the network administrator, it generally assigns higher priority to the protected applications.

The third part is the server or the resources that exist in the network. They are protected data server, which is in the secure context; authentication server, non-healthcare-related server, and the gateway to the Internet, which are in the default context.

3.2 Workflow to Create Contexts

Figure 3.3 shows the start of the workflow to create the contexts. The network only has the default context. Figure 3.4 shows the time when the client device connects to the network. Before the client can access the medical application data, the client must authenticate with the authentication system in the network to create the secure context. Figure 3.5 shows the client using the default context to access the authentication system. The authentication system allows the SeaCat server to authenticate the client. It also allows the SeaCat server to obtain detailed information about the client so that more specific context for a particular client can be created. Once the client is authenticated, SeaCat server dynamically creates the secure context for this client in the network (Figure 3.6). Then the SeaCat server requests the TD to create the context in the device in a way that extends the context from the network to the device (Figure 3.7). Finally, Figure 3.8 shows the time when the TD creates the context in the client device and starts the protected application that uses the end-to-end secure context.

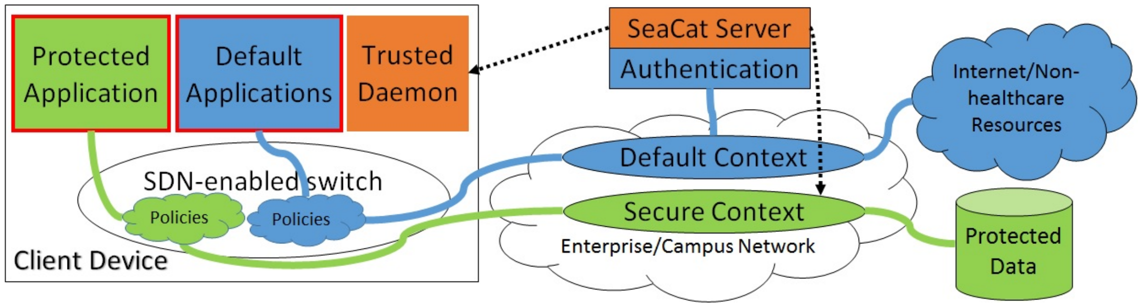

Figure 3.1: SeaCat architecture. Protected application and default applications (user's own and potentially malware) are isolated. SDN-enabled switch sets policies in a way that extends the enterprise security context into the device. Trusted Daemon is responsible for managing the isolation. It also communicates with SeaCat server to tie the two network domains together. The user uses the default context from the default application to authenticate and create the secure context. Upon successful authentication, SeaCat server creates the secure context in the network. Protected data, non-protected data, and a gateway to the public internet exist in the network.

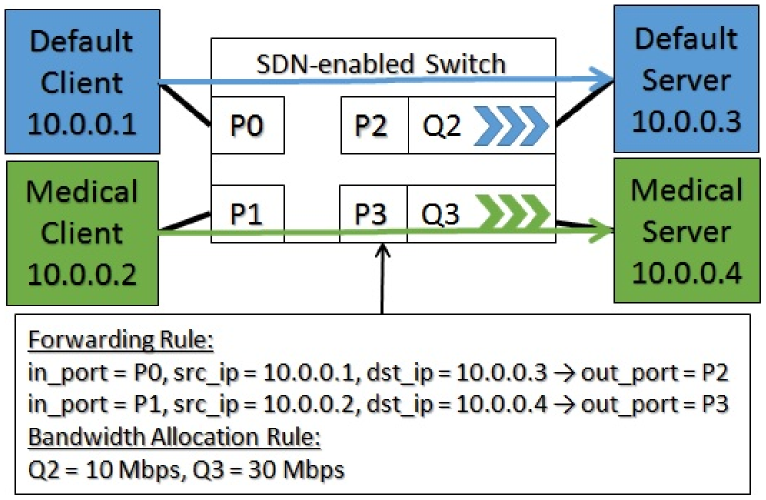
Default Client 10.0.0.1
Medical Client 10.0.0.2
SDN-enabled Switch
P0
P1
P2
Q2
P3
Q3
Default Server 10.0.0.3
Medical Server 10.0.0.4
Forwarding Rule:
in_port = P0, src_ip = 10.0.0.1, dst_ip = 10.0.0.3 → out_port = P2
in_port = P1, src_ip = 10.0.0.2, dst_ip = 10.0.0.4 → out_port = P3
Bandwidth Allocation Rule:
Q2 = 10 Mbps, Q3 = 30 Mbps

Figure 3.2: Example of network context. Policies, or rules, applied to the SDN-enabled switch create the logical isolation of the two connections. Queuing mechanism of the switch is used to prioritize the traffic.

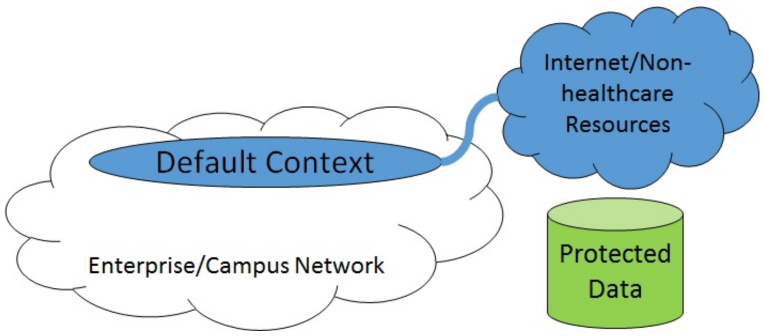

Figure 3.3: Start of the workflow to create the contexts. At the beginning, only the default context exists in the network.

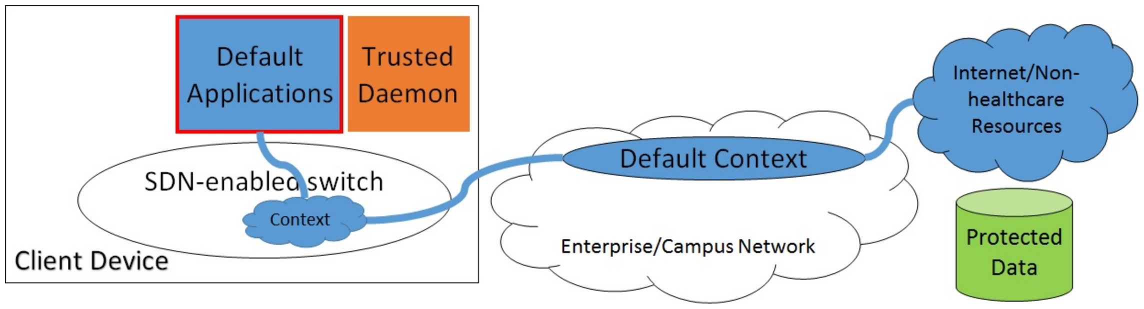

Figure 3.4: Client device connects to the network with default context.

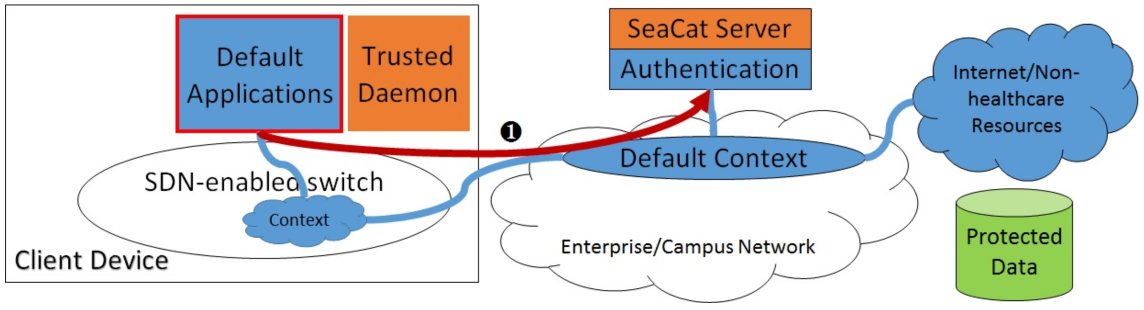

Figure 3.5: Step 1: Client uses the default context to access the authentication system.

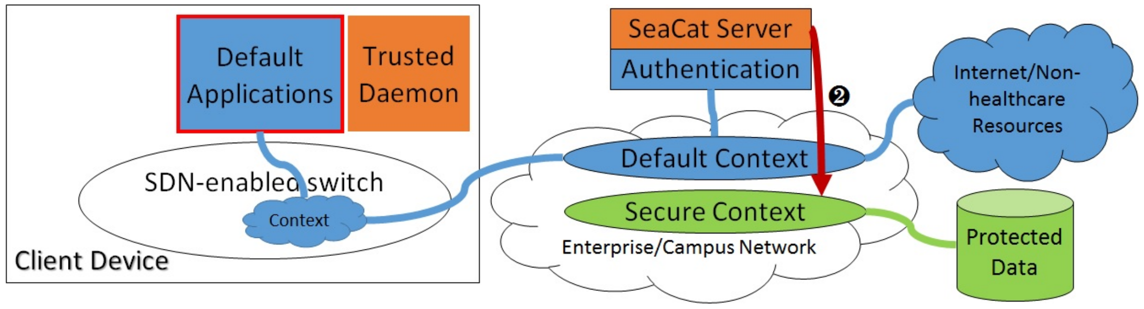

Figure 3.6: Step 2: SeaCat server creates the secure context for this client in the network.

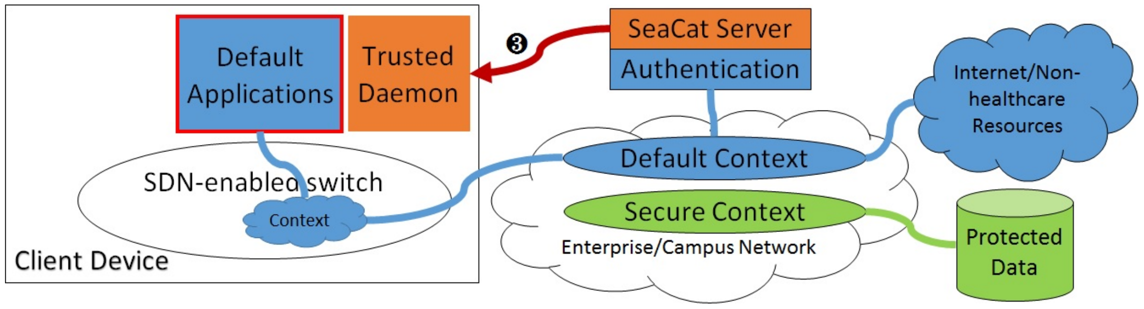

Figure 3.7: Step 3: SeaCat server requests the TD to create the context in the device.

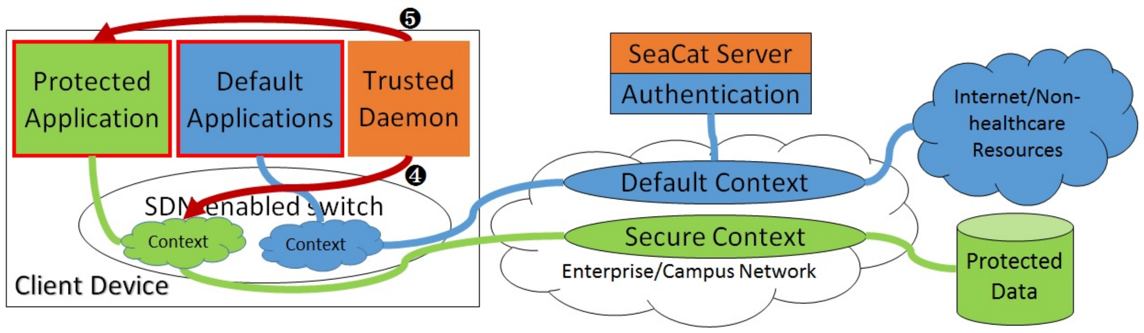

Figure 3.8: Step 4: TD creates the context in the client device. Step 5: TD starts the protected application.

CHAPTER 4

IMPLEMENTATION

Figure 4.1 shows the implementation of the client device. There are two containers; one for the medical application and the other for non-medical applications, which may include malware. It also has TD, which manages the containers and sets policies to the OpenvSwitch (OVS), which is the SDN-enabled switch on the endpoint (client device). OVS is used to create a separate context for each container. Green context, which connects the application container on the bottom, is for medical applications. Blue context, which connects the default container on the top, is for non-medical applications. Note that we used OVS for our SDN realization, but another SDN-enabled switch can also be used. Finally, we focus on the devices that access the enterprise network using Wi-Fi. For this type of device, two virtual Wi-Fi interfaces (VIF) are used to separate the contexts over the Wi-Fi link.

Figure 4.2 shows the implementation of the enterprise network. It has the Wi-Fi AP for the client to connect to the network. The AP has an OVS instance so that the policy can be created. It has multiple virtual APs (VAP) and each VAP can support a separate context. In this case, two contexts are supported; green is for medical application and blue is for default as in the client device. Each VAP is associated to the appropriate VIF of the client device. The contexts extend to the enterprise network where there could be many switches, application servers, and non-medical resources. The network also has the authentication system, SeaCat server, and the controller. The controller creates the contexts in the network and the AP.

The following sections describe the key components in detail.

4.1 Container Isolation

We use Linux Containers (LXC) to isolate medical applications from other applications within the client device. LXC is the kernel containment technology to create an isolated environment (container) to run processes. It uses `Cgroups` and namespace isolation to achieve this. `Cgroups` is used to allocate hardware resources such as CPU bandwidth and memory to each container. A namespace creates an isolated environment for processes running inside. There are six namespaces; IPC (Inter-Process Communication), network, mount, PID (Process ID), user, and UTS. For example, the processes in different IPC namespaces cannot communicate through IPC. LXC uses the kernel to enforce these six types of isolation to create containers. We used Ubuntu OS to implement and evaluate the containers.

The main advantage of using LXC is that it is light-weight. Creating, destroying, starting, and stopping the containers are faster and much less resource-intensive than using virtual machines. Figure 4.3 shows our implementation of a client device. To reduce the trusted computing base (TCB), our goal is to migrate all the default processes from the root namespace to the default container so that only the bare minimum set of processes runs on the root namespace. Namespaces are created in hierarchy. The process that creates a (child) namespace can see all the processes running there. Thus, the processes in the root namespace can see all the processes running in the containers. We run the special process in the root namespace; i.e., Trusted Daemon (TD). The main role of the TD in terms of application containment is to create the new container where a medical application (e.g., OpenMRS [35]) would run.

To achieve the process isolation efficiently, we first mount each container's filesystem to a different mount point on top of the overlay filesystem. Then we create the container at a separate mount point. Figure 4.4 shows how the containers are mounted in terms of the filesystem structure. First we create `/containers` directory. In this directory, we create a directory for each container. In this case, `/containers/1` is created for the default container and `/containers/2` for the application container. Under these directories, there are three directories called `overlay`, `upper`, and `lower`. All the default jobs and the applications are moved to `/containers/1/overlay`. The jobs must be stored in `/etc/init` to run the container so we copy all the necessary jobs from `/etc/init` to `/containers/1/overlay/default/etc/init` and remove the unnecessary ones from `/etc/init`. This method creates separate and custom environments for each container.

To run the operating system inside the container, many other files are necessary. Because the overlay filesystem is used, the containers can access the necessary files in the root directory. More precisely, the processes in the container can *read* the files in the root directory through `lower` directory, but cannot *write* to them. If a process tries to modify or delete any file in the root, the change is stored in the `upper` directory. The same idea applies to mounting the application container at `/containers/2`. Thus, we use the overlay filesystem to efficiently copy the configuration files from the root to the containers.

Mounting each container at different mount point prevents processes to access directory higher than `overlay`. This mechanism provides the filesystem isolation between containers. Finally, when the container is started, `/sbin/init`, which is spawned from the root, will start the jobs that are moved to the container.

Figure 4.5 shows more detailed container isolation structure. The application and the default containers share the same kernel. TD and SDN controller run in the root namespace. TD receives the signal from the SeaCat server. It also runs the SDN controller to set the flow entries on the OpenvSwitch.

In order to realize SeaCat's isolated containers, we have selected appropriate jobs that are started by Upstart for each container and for the root namespace to create the appropriate environment. Upstart is the `init` daemon that handles starting the jobs during the system boot [44]. In general, only the minimum jobs must run in the application container to run the medical application. Because the default container represents the environment where everything else runs, most of the jobs in the original kernel configuration run in this container. For example, `procps` is the utility to browse `/proc` filesystem. Although it is helpful to have this job in the default container, it is unnecessary in the application container. Finally, jobs that relate to system configurations and daemons run in the root namespace. One example is `cgroup-lite`, which is the package to set up `Cgroups` at system boot. `Cgroup` is the tool to isolate hardware resources in LXC.

4.2 Client Hardware Resource Guarantee

Two of the most critical hardware resources for any kind of application are CPU and memory. We use Linux's `Cgroups` to allocate these resources to each container to provide hardware resource guarantees for a medical application. Figure 4.6 shows the role of `Cgroups`. `Cgroups` is one of the main technologies used in LXC to allocate, prioritize, deny, manage, and monitor such hardware resources. By default, all the hardware resources are shared among containers. We use `Cgroups` to allocate CPU cores, CPU bandwidth, and memory to each container. This adds additional guaranteed resources to the client device on top of the network bandwidth. We have tested to show that it is possible to allocate these resources to our containers.

4.3 Window System Isolation

Figure 4.7 shows the Xserver isolation between containers and the root namespace. Xserver is the X Window System display server used in Linux. It receives images from the client application (e.g., browser, console) and sends the information to the display. It also handles the input from the user such as keyboard press and mouse movement. We use Xnest [32] to isolate the containers in terms of their I/O operations.

We run the containers inside their own Xnest so that separate Xserver is used. This construction prevents the ability to copy and paste or keyboard logging between the containers and the root namespace. Thus, even if the user opens browsers from different containers, the user will not be able to copy sensitive information from the application container to the other. In addition, the malware in the default container will not be able to obtain keyboard logging or display information of the application container. Xnest uses Unix Domain Socket (UDS) to communicate to the root Xserver. We make sure that the containers do not share the UDS of other containers or the root. This prevents access to outside of the container through UDS.

4.4 Networking Configurations

We use OVS on the network, the AP, and the client device to create contexts. The advantage of OVS is that it has a well-defined API and it is easy to set complex policies dynamically. We also virtualize the physical Wi-Fi interface of the client device and the AP to support multiple contexts. The advantage of having virtual interfaces is that they create the isolation of Wi-Fi traffic in the similar way as the Ethernet VLAN over the air.

4.5 Network Resource Allocation

The main advantage of constructing the SDN context in an enterprise network is its simplicity. The SDN makes routing packets and allocating bandwidth simple and dynamic. Hence, it is the perfect tool to isolate the traffic and prioritize the network resource for different types of applications.

When the medical application traffic needs to share the same switches and links with other traffic (e.g., non-medical-related traffic such as video streaming), the user may experience slow connection or it may even be disconnected. SeaCat allocates the network resource to different types of traffic in a way that does not cause waste. It allows the network administrator to set the priority on each type of traffic to achieve flexible and fine-grained resource allocation.

The challenge is that the throughput of various types of traffic keeps changing. It is not possible to meet this changing throughput demands with the static bandwidth configurations. In addition, there may be multiple classes of medical traffic. For example, there may be two levels of priorities, or classes, where one is the medical traffic that does not require a large bandwidth, such as EHR application traffic. The other class could be a remote diagnosis application traffic that requires larger bandwidth. Finally, there is the default class that represents the non-medical-related traffic. The problem is to allocate appropriate amount of bandwidth to each class of traffic dynamically. The solution is to use the queuing mechanism of the SDN-enabled switches.

SeaCat uses the network queuing of the OVS in the client device and the network. Figure 4.8 shows an example. The client device, the AP, and the enterprise network have queues that impose different bandwidth to different type of traffic in both directions. In this case, a separate set of queues is created for medical and non-medical traffic. Our solution is to dynamically change the maximum bandwidth of each queue based on the priority and the throughput of traffic. In our experiments, we were able to show that our bandwidth allocation algorithm appropriately allocated bandwidth to achieve the desired level of network resource allocation.

4.6 Workflow with Single Sign On

Shibboleth [41] is the open source single sign on system that allows clients to authenticate before accessing a protected resource such as a website. It consists of two components; Service Provider (SP) and Identity Provider (IdP). SP is typically installed on the Apache server. Its role is to intercept the HTTP request from the client to the protected resource and redirect it to IdP. The role of IdP is to authenticate the client by providing the login page. When the username and password are entered, IdP checks them with the information stored in the database. If those information are valid, IdP redirects back to the Apache server where SP grants the access to it.

We leverage the authentication mechanism of Shibboleth by allowing it to store policies associated with each application so that application specific policies are applied to contexts. When the user finishes using the application, the user can use Shibboleth's logout feature to remove the context. Another contribution is that Shibboleth is open source software that are widely deployed so it is easy to implement and configure to a particular system. Finally, although we integrated Shibboleth into SeaCat to create the secure context behind the scene, the authentication steps that a user needs to take is very similar to what they are used to with other services that use Shibboleth.

Figure 4.9 shows the Shibboleth SSO steps to prepare the policy enforced network context. Following is the description of each step:

1. TD starts the default container.

2. User accesses (i.e., enters URL to the browser) the Service Provider (SP), which protects the access to the SeaCat server.
3. User is redirected to the Identity Provider (IdP), which shows the page to enter username and password.
4. User enters the correct username and password.
5. IdP replies SAML assertion to the browser. It contains the authorization decision that allows the client to access specified resource; SeaCat server in this case.
6. Browser redirects SAML assertion to SP.
7. SP queries IdP for more specific information about the client (optional).
8. IdP replies the stored client information (optional).
9. User is authenticated. SP allows the client to access the webpage, which executes the SeaCat program. SP sends Session ID to the browser.
10. The program in SP notifies the controller to create a particular context for this client (medical application). Controller creates policy enforced context in the enterprise network.
11. SeaCat program signals the TD that the policy enforced context is ready. It tells the TD to use the set of policies that are consistent with the context that will be created by the controller at Step 12.
12. Controller creates the context in the network.
13. TD adds flow entries to OVS for policy enforced context.
14. TD starts the application container.
15. User can use the medical application in the application container to connect the server using the policy enforced context.

Figure 4.10 shows our contribution to the SSO process. Step 2 to 9, which are grayed out, are the standard Shibboleth procedures. In the standard procedures, the goal of the client is to view the webpage after authenticating with the SP. Thus, after Step 9, the standard HTTP communication will begin. Instead of providing the webpage to the client, SeaCat runs the SeaCat program, which is written in Python, to notify the controller to create the context for this client (Step 10) and notifies the TD to create the same context in the client device (Step 11). Accomplishing both of these tasks from one program creates the centralized context management entity that manages the context in the network and all the client devices.

4.7 Securing the Link Between Client and AP

Potential security hole exists in the place where the two network domains connect; that is, the link between the client device and the Wi-Fi AP. Suppose a legitimate client follows our procedure and creates the secure context. Then a malicious client associates to the same AP as the legitimate client. This malicious client can sniff the packets for secure context. If the malicious user finds out the policies for the secure context (e.g., IP address of the medical application server for destination IP), the malicious user can access the secure context without being authenticated.

Our solution is to use 802.1X [24] and 802.11i [23] in the client-AP link. We use 802.1X to authenticate the client before allowing the access to the default context. Then 802.11i is used to encrypt the connection. Figure 4.11 shows the high-level steps. When the client, or Station (STA), associates with the AP and tries to use the default context, STA is required to provide the credential, usually the username and the password. This credential is sent to the authentication server (AS), which is typically implemented with Remote Authentication Dial In User Service (RADIUS). RADIUS server usually has the database that stores the credential of the STAs. STA and AS generate Master Key (MK) from the credential. Then they generate the Pairwise Master Key (PMK) from MK. After successful authentication, AS sends Success message to the AP, which is forwarded to the STA.

When the authentication is completed, the STA and the AP conduct the 4-way handshake to derive the Pairwise Transient Key (PTK). Part of the PTK are used to encrypt and decrypt the data sent between the STA and the AP. The encryption method includes CCM mode Protocol (CCMP), which uses AES. Since the AP knows that the STA is authenticated (i.e., AP received the Success message from the AS), its data are sent to the OVS of the edge of the network. At this point, the connection to the default context is established. After this point, if the client wants to use the secure context, the client follows our procedure. Authenticating the client prevents the malicious client to send data past the AP. Encrypting the client-AP connection prevents the malicious client to sniff the packets for either context.

Figure 4.12 shows the implementation that combines this solution and the rest of SeaCat's implementation. As part of 802.1X, the AP uses the VLAN tag to direct packets to the appropriate destination. Initially, packets for the unauthorized traffic are tagged as unauthorized. These packets are directed to the RADIUS by the OVS using the tag. After successful authentication, the AP assigns the (Default, Authenticated) tag to the authenticated packets, which are directed to the default context. At this point, the client can use the default context. If the client wants to use the secure context, the client proceeds with the SeaCat's authentication procedure. Finally, the medical traffic packets are tagged as (Secure, Authenticated) and forwarded to the secure context. The link between Virtual Interface (VIF) and Virtual AP (VAP) are encrypted with 802.11i.

This is a feasible solution because 802.1X and 802.11i are widely used in many enterprise networks. Note that this solution cannot prevent the malicious client that tries to interfere with the physical wireless link, for example causing a DoS attack. However, this solution protects against malicious clients that follow the standard protocols.

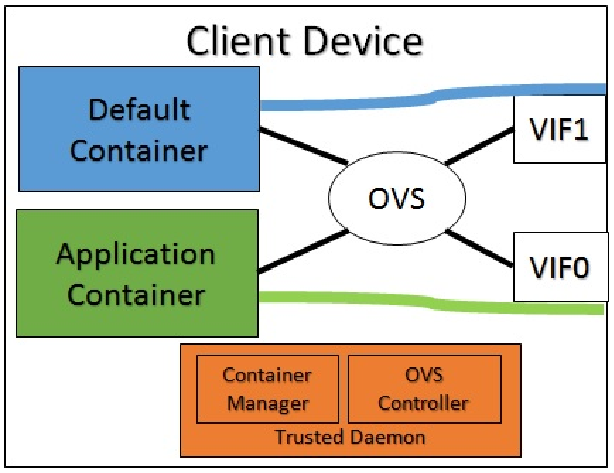

Figure 4.1: Implementation of the client device. There are two containers. One is for default applications and the other is for each medical application. A separate context is created for each container and separate VIF is assigned to each context. In this case, blue represents the context for the default container and green represents the context for the application container. In addition, TD manages the containers and the OVS on the device.

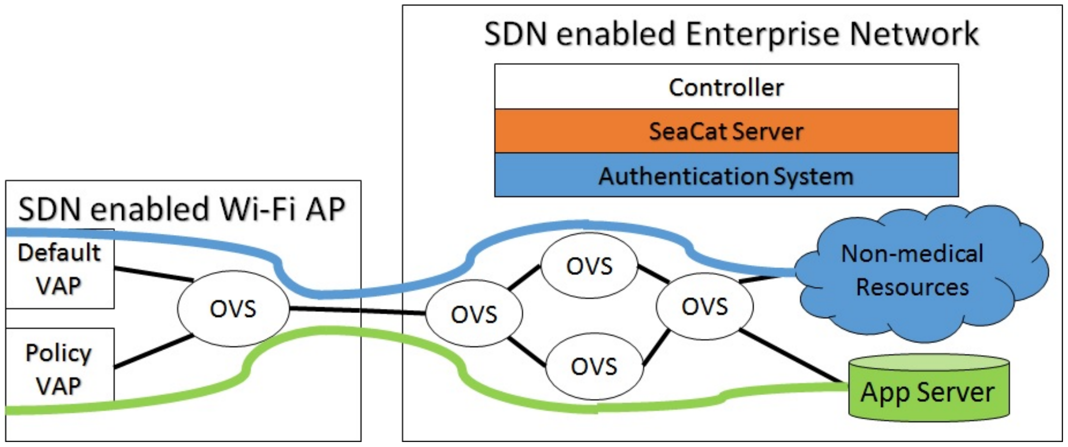

Figure 4.2: SDN enabled Wi-Fi AP is the gateway for the client device to the enterprise network. Separate contexts are created in the network as well. As in the client device, blue represents the default context and green represents the secured context. The user is authenticated with the authentication system. Then the SeaCat server requests the controller to create the context in the network and the AP.

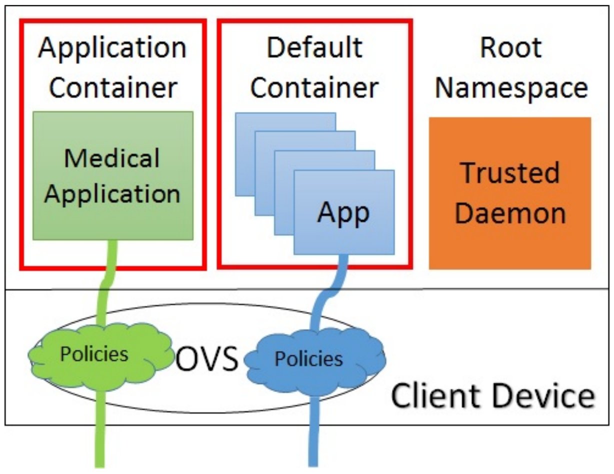

Figure 4.3: Implementation of containers on client device. Separate container is prepared for each medical application. Other applications and jobs that exist in the host in default are contained in the default container. TD runs in the root namespace. Separate set of policies are created in the OVS for each container.

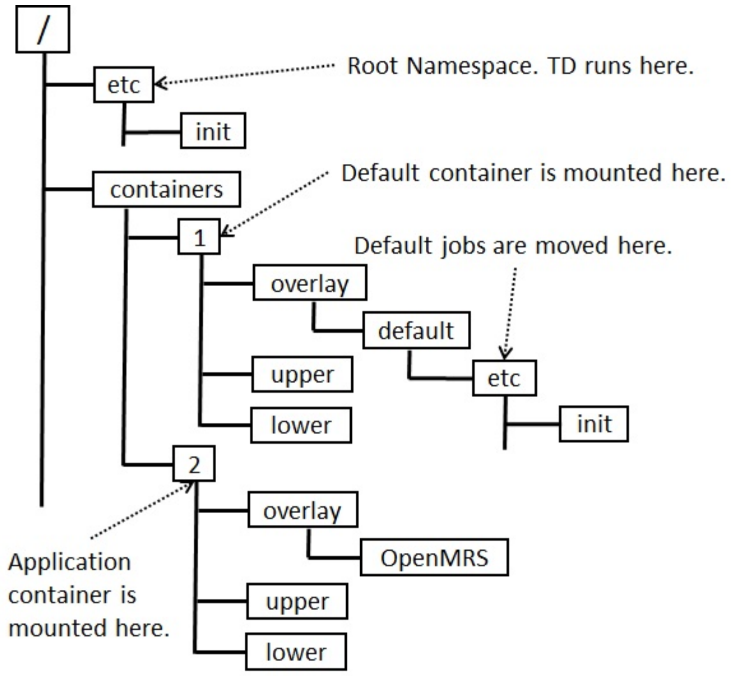
/
etc
init
Root Namespace. TD runs here.
containers
Default container is mounted here.
1
Default jobs are moved here.
overlay
default
upper
etc
lower
init
2
overlay
OpenMRS
Application container is mounted here.
upper
lower

Figure 4.4: Each container is mounted at different mount point (e.g., `/containers/1`, `/containers/2`). Overlay filesystem is used so that containers can share the necessary system files from the underline kernel. Although they are shared, they cannot be modified from inside the container because they are read only. If they are modified, the changes are kept in the `upper` directory of each container. In Linux, jobs are stored in `/etc/init`. We copy necessary jobs from `/etc/init` (root) to each container's `init` directory.

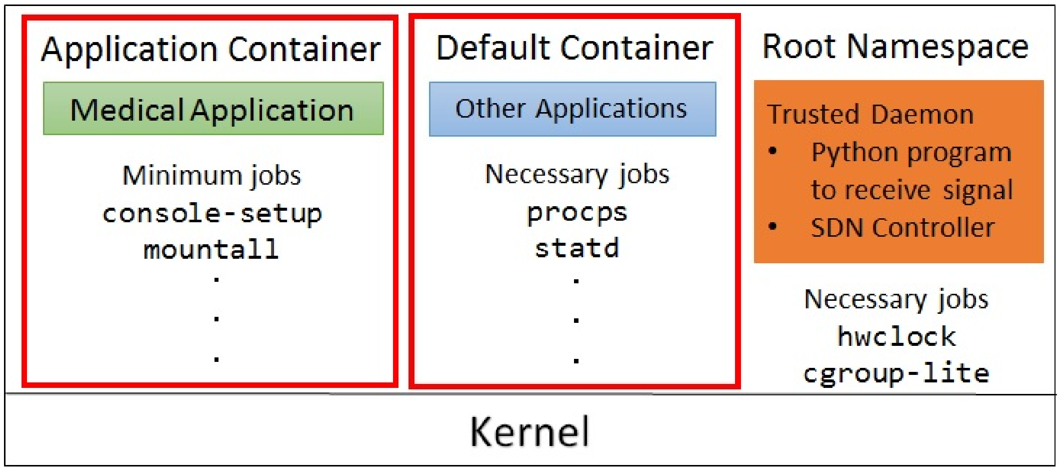

Figure 4.5: Red boxes represents containers. The application container, the default container, and the root namespace have different sets of jobs to create the appropriate environment. In addition to create/destroy, start/stop the containers, TD runs the Python program to communicate with SeaCat server and runs the SDN controller.

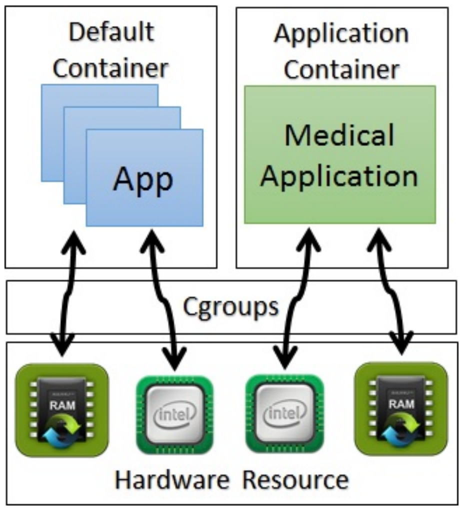

Figure 4.6: `Cgroups` allocates hardware resources to each container. We have tested that it can flexibly allocate CPU core, CPU bandwidth, and memory.

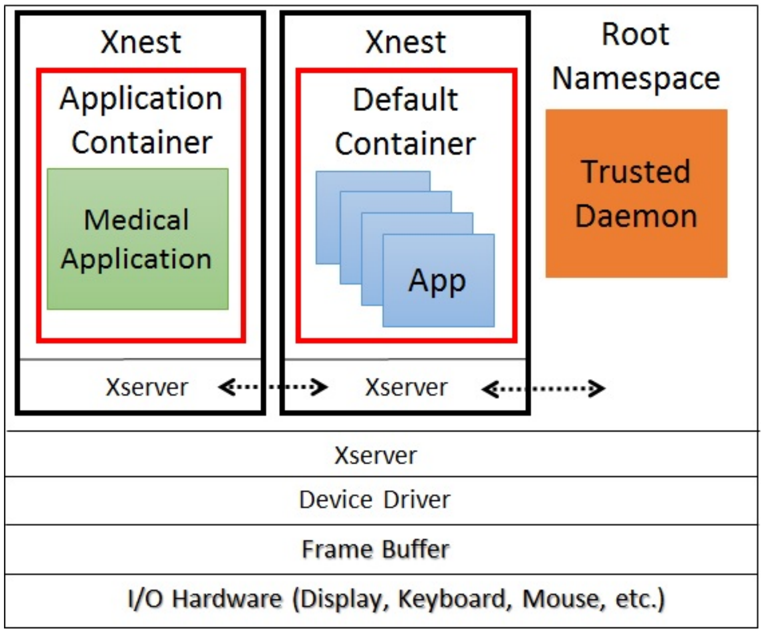

Figure 4.7: Thick black box represents the Xnest isolation of the containers. To display the image, Xserver of Xnest sends image to the Xserver of the root. Dotted arrows represent removed socket. In this diagram, the default container initially had access to the UDS of the application container and the root, which could be the attack vector.

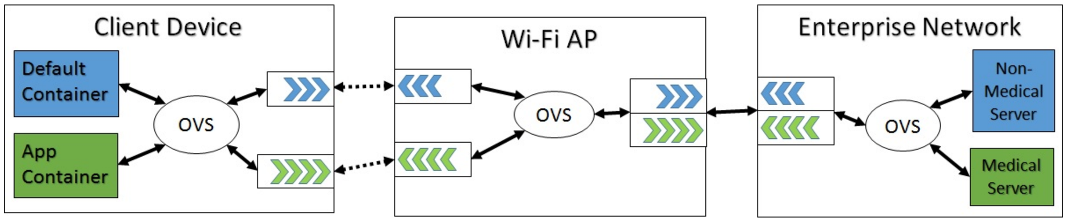

Figure 4.8: An example of using queues. Queues configured in client device, AP, and the enterprise network impose the maximum bandwidth of medical and non-medical traffic to each direction.

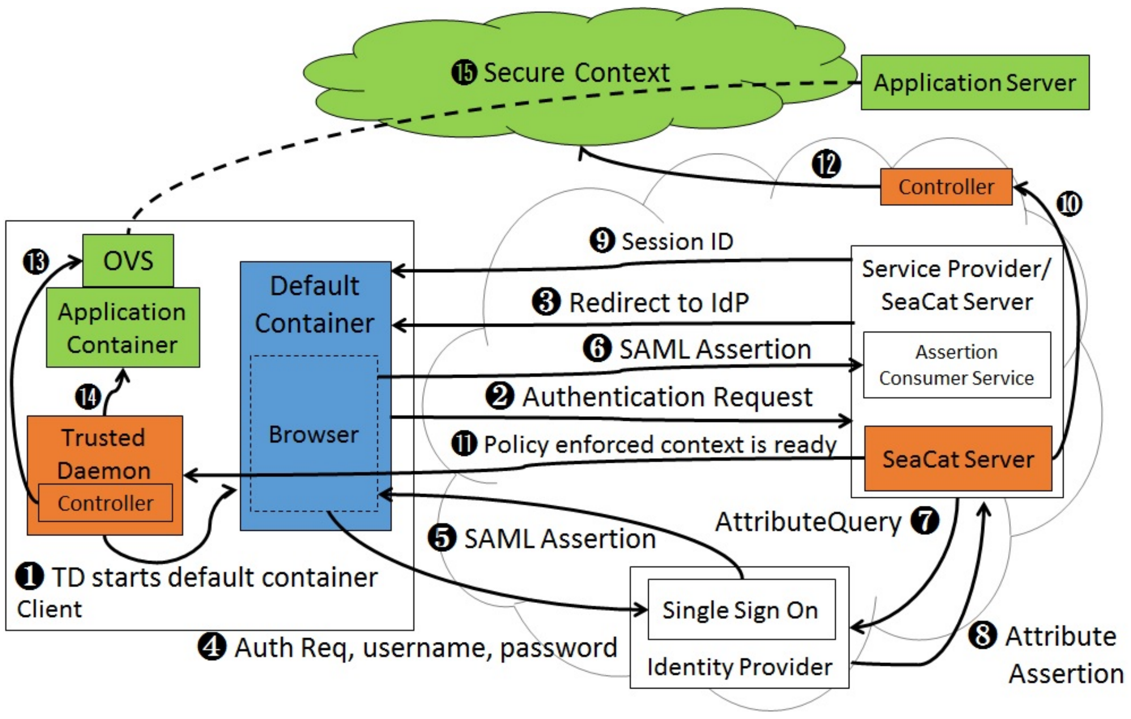

Figure 4.9: SSO steps that involve client, SP, IdP, and the controller. SP protects the access to a resource. IdP is responsible for checking the integrity of the client. Controller creates the policy enforced context upon request from the SP.

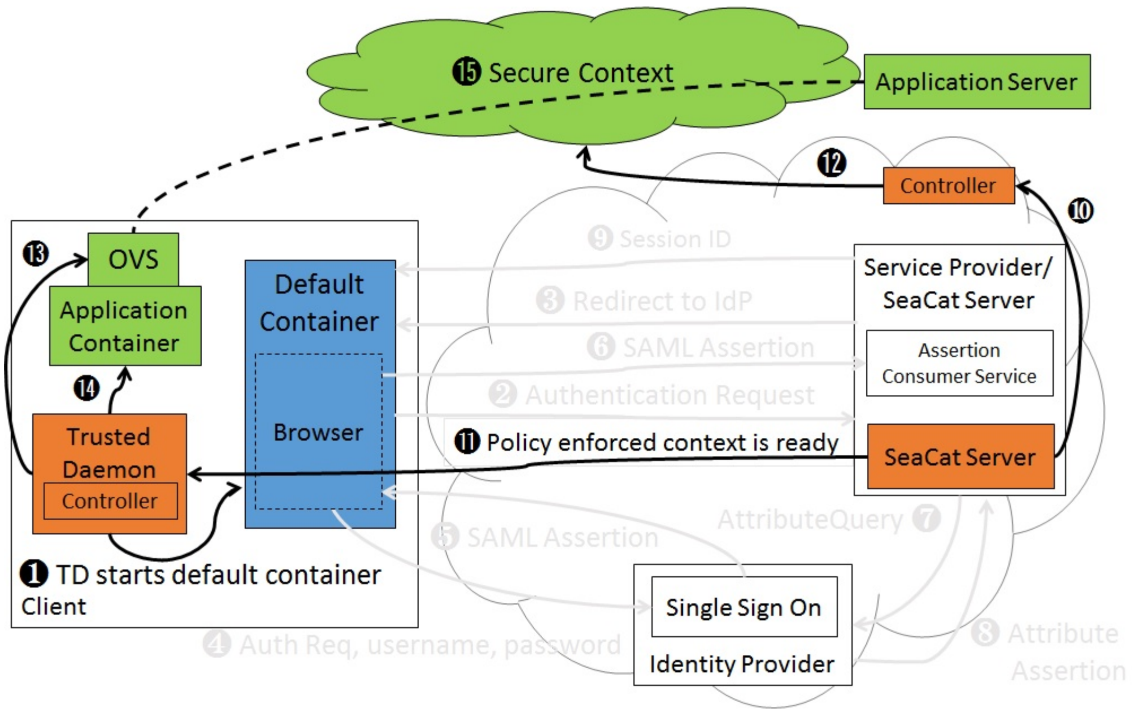

Figure 4.10: SSO steps that shows our contribution. Step 2 to 9 (grayed out) are the standard Shibboleth procedures. Instead of delivering a webpage, our SeaCat program requests the controller and the TD to create context. This shows that Shibboleth can be integrated into SeaCat. Because Shibboleth is widely used open source software, it proves SeaCat's feasibility in terms of SSO.

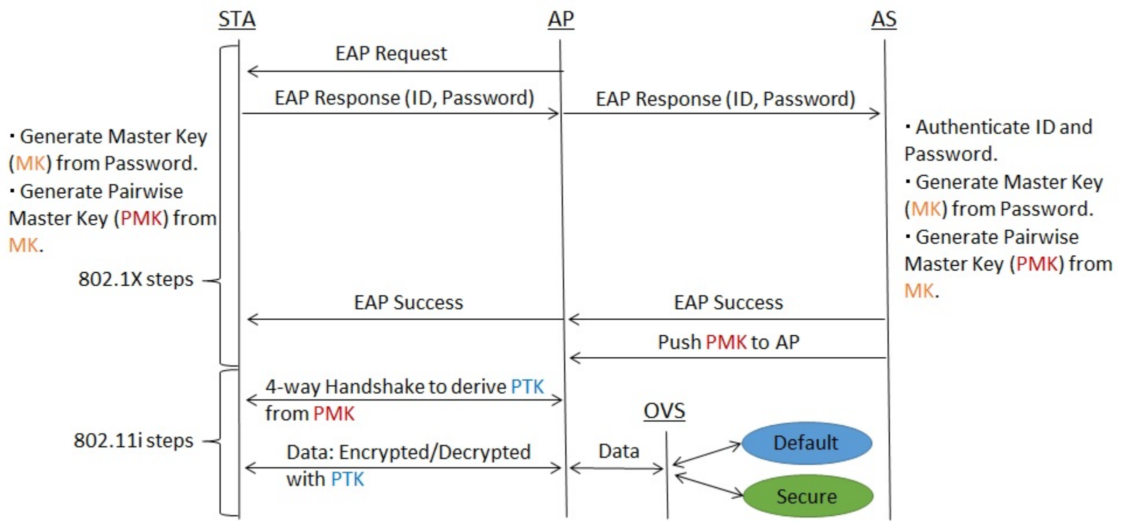
STA
AP
AS
EAP Request
EAP Response (ID, Password)
EAP Response (ID, Password)
· Generate Master Key (MK) from Password.
· Generate Pairwise Master Key (PMK) from MK.
· Authenticate ID and Password.
· Generate Master Key (MK) from Password.
· Generate Pairwise Master Key (PMK) from MK.
802.1X steps
EAP Success
EAP Success
Push PMK to AP
4-way Handshake to derive PTK from PMK
OVS
802.11i steps
Data: Encrypted/Decrypted with PTK
Data
Default
Secure

Figure 4.11: High-level steps that combine 802.1X (authentication), 802.11i (encryption), and our context.

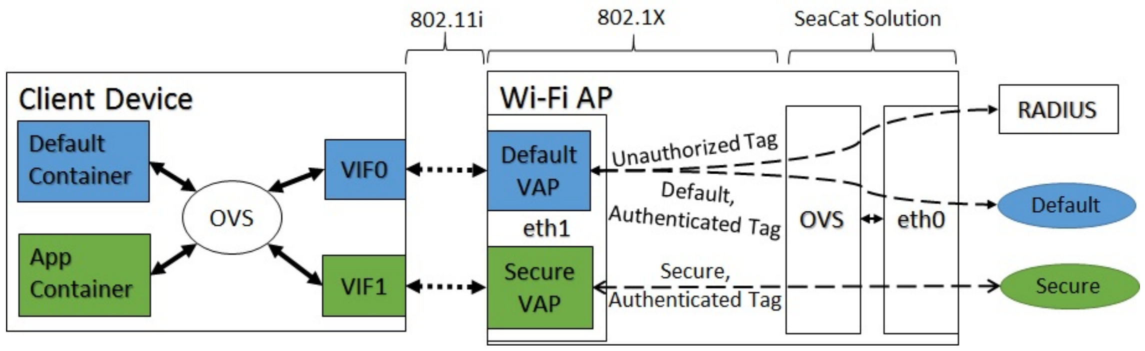

Figure 4.12: The architecture that improves the security of the Wi-Fi link. The AP uses the VLAN tag to differentiate the traffic. OVS checks the tag to send packets to the appropriate destination. We combine 802.1X and 802.11i to our solution.

CHAPTER 5

SECURITY ANALYSIS

5.1 Attacks

The client devices and enterprise network in a healthcare setting face various attacks. They include:

- **Information leakage from client's filesystem:** The default container has malware that try to steal sensitive information from the host's file system.
- **Information leakage from client's display event:** Since the display is shared among containers, when the default and the application container are running simultaneously, malware in the default container can try to steal sensitive information through the X11 window system. The user may accidentally copy and paste sensitive information from the application container to the default container.
- **DoS attack to the client from the default container:** Although malware in the default container cannot obtain secured information, they can generate large traffic to fill the client-AP link bandwidth to disrupt the medical application traffic that are sent simultaneously.
- **Hardware resource depletion on client device:** Since the hardware resources such as CPU and memory are shared among containers, processes in one container can use up all of those resources when the medical application needs them. It can be caused by a malware or some non-medical-related program running in the default container.
- **Unauthorized access to medical application server:** A malicious user attaches the host to the enterprise network trying to access the medical server to steal patient records. Alternatively, malware on the host of a legitimate user try to access the medical application server when the host is connected to the network.
- **Packet sniffing on policy-enforced context to steal sensitive information:** After the client authenticates with the SeaCat server and the end-to-end connection

is established, a malicious user connects its computer to the end-to-end connection path to obtain sensitive information sent.

- **DoS attack on switch or AP on enterprise network to disrupt the connection for medical application:** A malicious user attaches hosts to the enterprise network to conduct a DoS attack to disrupt the medical traffic. Alternatively, malware or existing devices in the network might launch a DoS attack.
- **Unauthorized access to existing secure context:** A malicious user associates the client device to the AP to connect to the default context. Assume another legitimate user has created the secure context through this AP. Then the malicious user guesses the policies used for the secure context correctly and uses the secure context without being authenticated. The same attack can be conducted from the wired switch in the network. That is, a malicious user may try to use the existing secured context by directly connecting to a switch with a wire.
- **Packet sniffing at the client-AP link:** A malicious user sniffs packets sent over Wi-Fi, especially those for secure context.

5.2 Solutions

Based on the SeaCat architecture and implementation, we propose solutions to the expected attacks described in 5.1.

We first consider a malicious user connecting to the enterprise network. The user cannot access the secured resource such as a medical application server because the network context to access the resource must be created beforehand. Prior to the authentication, every user can only access the default context. Since a secure context is created only after successful authentication with the SSO system, SeaCat prevents malicious users from creating a policy-enforced context to access secured resources.

We also consider the case when malware exists in the default container of a legitimate user. There are four types of threats in this case. First, suppose some medical application allows the user to store information (e.g., patient record) on the host. In addition, malware exists on that host trying to steal this information. In this case, SeaCat's container structure prevents malware from accessing the file system of different containers, thereby protecting the secured data to leak out of the medical application container.

Second, malware tries to obtain secured information through the client's display. We prevent this threat by running medical applications and other applications inside Xnest. Xnest is both an Xserver and a client. Since separate Xservers are used between the containers (and the root), malware in one container cannot access I/O hardware for another container. For example, a process in one container cannot log the keystrokes for another container, which is possible if Xnest is not used. In addition, Xnest prevents the user from copying and pasting between the containers. We confirmed that Xnest uses Unix Domain Socket (UDS) to communicate to the root Xserver (e.g., to display image, to obtain keyboard input). To prevent malware in the container to access another container through UDS, we remove the sockets from the overlay filesystem of each container.

Third, malware tries to disrupt the simultaneous medical application traffic by generating a large load of traffic from the default container. Our solution is to prioritize the bandwidth for the application container by using the queuing mechanism of the SDN-enabled switches in the client and in the network. Since each queue limits the maximum bandwidth, we can easily allocate enough bandwidth to the traffic for different containers. This solution also applies to the case when the malware exists in different host to attack the AP or the enterprise network.

Fourth, malware tries to deplete the CPU and memory resource for medical application. Our solution is to guarantee enough of those resources to the application container using `Cgroups`. `Cgroups` is the main tool used in LXC to group processes and assign accessible devices for each container. We allocate appropriate hardware resources (CPU core, CPU bandwidth, available memory) to each container so that the processes will be guaranteed to have them and prevent any impact to other containers.

Unauthorized access to the medical application server can be prevented with SDN. Our system separates the traffic from the default and the application container. Thus, policies on the OVS in the host and in the enterprise network limit the default container's accessibility to the default context. Our system prevents unauthorized access to the secure context from the end-point. However, a malicious user may try to access the existing secure context through the AP. This attack can be prevented by making the client to authenticate, with 802.1X, before allowing access to any of the context. Although we do not enforce such security measure on the enterprise network, it is much easier to restrict a direct physical access to switches than restring the access to the AP. Even if a malicious user can somehow access the switches, the medical application traffic is encrypted so the privacy is secured.

Packet sniffing on policy-enforced context can be prevented by encrypting the connection in the application level. The most effective way is to use existing methods such as SSL or IPsec. The Wi-Fi link is especially vulnerable against packet sniffing. The link is secured by encrypting the connection with 802.11i.

CHAPTER 6

EVALUATION

To evaluate our system, we have implemented it in a small testbed, which includes three laptops and one AP. Figure 6.1 shows the implemented system. The client laptop is the Linux-based tablet, which has two LXC containers, TD, OVS, and two VIFs. `Cgroups` is also configured for the hardware resource guarantee. We use the two VIFs (`VIF0` and `VIF1`) to separate the traffic between the default and the medical application. On the AP, it has 16 virtual Wi-Fi interfaces so it can support up to 16 different contexts with different VIFs. In our deployment, we used two VIFs (Policy VAP and Default VAP) to support the default and the medical application context.

The server laptop runs Mininet to emulate the wired SDN enterprise network. The emulated enterprise contains a medical application server, the SeaCat server, the Service Provider(SP) and Identity Provider (IdP) of SSO, and another server that emulates a server on the default network. We have emulated the enterprise network topology with the Mininet to evaluate the resource prioritization feature.

The SDN controller for the enterprise network is on the third laptop. There are two roles for this controller. One is the flow manager, which inserts flow entries to the Mininet network and to the OVS of the AP upon request from the SeaCat server. Another is the DHCP, which assigns IP address to the containers of the client laptop. We use Ryu for the controller.

6.1 Network Resource Prioritization

In the enterprise network, there may be multiple classes of traffic. The problem of network resource prioritization is to allocate appropriate amount of bandwidth to each class of traffic dynamically. Our solution is to use the queues in the SDN-enabled switches.

6.1.1 How Queues Work

Figure 6.2 shows a simple topology where queues can be useful. Suppose Class 1 Server and Class 2 Server want to send streaming data to their associated client. Since the computational resource of the switches and the link between Switch 1 and 2 are limited, it is necessary to allocate existing bandwidth to each traffic. To do so, queues are configured in the port indicated as `s1-eth1`.

Figure 6.3 shows the detail of the queue implementation. In this example, there are three queues indicated as Class 1, 2, and 3. They are implemented as Hierarchy Token Bucket [12]. The width of the arrow in the bottom of each queue indicates the maximum bandwidth allocated to the queue. The colored marble represents a packet. In this case, Class 3 queue has the highest bandwidth and has the largest number of packets waiting. The registered flow entries, which is represented as the upper trapezoid in the figure, direct packets to their corresponding queues. Note that the length of each queue is the same in our implementation as in this example. If a queue is full, the arriving packet to this queue would be dropped.

6.1.2 Solution

The simple approach is to use the Weighted Fair Queuing (WFQ) algorithm [11, 39] to realize the network bandwidth isolation. This approach allocates the fixed weight, or bandwidth ratio, to each queue and allocates the traffic to a queue. Under WFQ, suppose the weight for queue i is represented as w_i; then packets in each queue receive a fraction of service equal to $w_i/\Sigma(w_j)$ where the sum in the denominator is taken over all queues that have packets. This represents the work-conserving property, which means the scheduler immediately serves the next queue if the current queue is empty. We have come up with a unique approach that can provide similar traffic prioritization to WFQ, but can also allow the weight to fluctuate based on the throughput and the type of traffic.

Our solution is to dynamically adjust the bandwidth of each queue based on the priority of the ongoing traffic. There are two factors that determine the priority. One is the type of traffic such as medical or non-medical and the other is the current traffic load.

Figure 6.4 shows the bandwidth transition of the queues based on the type of traffic. In this case, the algorithm considers the medical traffic to have higher priority than the default traffic given the similar traffic load. At first, higher traffic load is observed in the default queue, indicated by larger number of blue marbles, compared to the medical traffic, indicated by orange marbles. When the traffic load of medical traffic starts to catch up with the default traffic, the allocated bandwidth of the queue of medical traffic is increased, indicated by the wider arrow at the bottom of the queue. At the same time, the allocated bandwidth of the default queue is decreased, which is indicated by the narrower arrow. As a result, the default traffic throughput is throttled.

Figure 6.5 shows the bandwidth transition based on the current traffic load. As the previous example, medical traffic is considered to have higher priority. At the beginning, the load of the default traffic is much lower; hence only a small portion of bandwidth is allocated to it. When the load of the default traffic starts to increase, the bandwidth is transferred from the medical traffic queue to the default traffic queue because of the policy that the lower the current traffic load, the higher the priority of that traffic becomes.

Combining these two types allows us to adjust the bandwidth of each queue in dynamic and fair manner.

6.1.3 Bandwidth Allocation Algorithm

We implemented the algorithm to realize the above solution. Note, the numbers used in the following explanations are only for the purpose of explaining this algorithm and showing its validity.

We first divide the overall bandwidth to 10 Mbps chunks. Then we allocate 10 Mbps to each queue. If the throughput of a queue reaches some upper threshold, say 8 Mbps, we simply allocate another 10 Mbps so that the bandwidth of this queue becomes 20 Mbps. On the other hand, if the throughput becomes below the lower threshold, say 8 Mbps while having 20 Mbps max, we take away the 10 Mbps from this queue and it goes back into the bandwidth pool. This pattern continues as long as there is available bandwidth left in the pool.

When all the available bandwidths are allocated, and yet some queues still want more bandwidth, we transfer the bandwidth among the queues. To decide which queue will receive or release the bandwidth, we assign a priority function to each queue. Figures 6.6, 6.7, 6.8 show the priority function of three queues. The arrow represents the current throughput. Priority is represented as the inverse function of the throughput. Intuitively, as the throughput of a queue goes up, the priority of this queue goes down. As a result, it becomes more likely that the bandwidth would be taken away from this queue. In addition, given the same range of throughput, the higher the curve of a function, the more important that function becomes. Thus, the priority function allows us to use a single line to simultaneously represent the two types of priority: current throughput and the importance of traffic. The following steps describe the transfer algorithm:

1. Divide the area under the curve in every 10 Mbps range.

2. Determine the current priority (CP) of each queue from the current throughput by finding the section area under the curve. For example, suppose the current throughput

of Queue 1 in Figure 6.6 is 29 Mbps. Then the current priority of this queue is the area under the curve in the range of 10 to 20 Mbps, which is 20. Current priority of each queue is indicated by the area between the blue and the purple line. Note that the area under the curve in the first range (between 0 to 10 Mbps) would be infinity, so we instead calculate the area from 1 to 10 Mbps.

3. If the current throughput of a queue is above the threshold, determine the prospective priority (PP), which is the adjacent section under the curve. For example, suppose the current throughput of Queue 3 in Figure 6.8 is 29 Mbps. Then the prospective priority of this queue is the area under the curve in the range of 30 to 40 Mbps, which is 43. PP of each queue is indicated by the area between the purple and the red line.

4. Sort the CPs in ascending order and the PPs in descending order. Table 6.1 shows the list of CPs and PPs calculated from the figures.

5. From the top of the CP and PP list, check if PP is larger than CP. If so, the bandwidth transfer between those two queues occurs. Check the next PP-CP pair and so on. If the transfer is rejected once (i.e., PP $<$ CP), no more transfer will occur. From the example of Table 6.1, Queue 3 receives 10 Mbps from Queue 1. The resulting state is shown in Figure 6.9 and 6.10. With the extra 10 Mbps, Queue 3 can support its 35 Mpbs throughput, while the throughput of Queue 1 is capped to 20 Mbps.

6. Wait for some interval, say 10 seconds, and go back to step 2.

Formal algorithm is shown in the Appendix.

6.1.4 Experiment Setup

The enterprise network topology we used for the experiment is shown in Figure 6.11. We implemented the queues in the ports indicated by QueuesN. Note that in the actual deployment, queues should be configured in every switch including the OVS of a host. However, the setup in Figure 6.11 is simpler and enough to show the effectiveness. Although there are only two access switches in the figure, there could be more in the actual network. Default Client represents the set of clients who use the default context. Medical 1 Client represents the set of clients who use the secure context Class 1. Class 1 represents less important medical traffic in terms of network bandwidth (e.g., EHR traffic) and Class 2 represents more important traffic (e.g., remote diagnosis). Each class is mapped to each queue in a port. We believe three is the appropriate number of classes, but the above algorithm supports arbitrary number of classes.

We implemented the above algorithm in the controller. Every 10 sec. of the polling interval, the controller measures the throughput of each queue. When the algorithm indicates that the bandwidth increase or decrease to particular queue(s) is necessary, the controller sends the commands to the switches through their REST API to change the bandwidth. We used `iperf` to generate and measure the UDP traffic for experiment 1 to 6; and generated TCP traffic for experiment 7. Experiment 1 and 3 to 7 show the behavior of the priority function algorithm. Experiment 2 compares the priority function algorithm with WFQ.

The maximum bandwidth at each port is set to 100 Mbps. The maximum bandwidth at each queue regardless of other simultaneous traffic is set to 70 Mbps. At the beginning of each experiment, 10 Mbps is allocated to each queue. The upper threshold to request 10 Mbps bandwidth increase is current max. bandwidth − 2 Mbps. The lower threshold to release 10 Mbps bandwidth is current max. bandwidth − 12 Mbps. We used the three priority functions introduced earlier (shown in Figure 6.6, 6.7, 6.8). We have conducted the experiment with Mininet.

6.1.5 Results

Experiment 1: This experiment shows the behavior of the bandwidth allocation when the Default Client and Medical 1 Client download at the same time (Figure 6.12). Figure 6.13 shows the throughput transition measured at the two clients. At first, the server generates the traffic of 5 Mbps and gradually increases it to 75 Mbps (5, 15, 25, and so on). The graph shows that the controller allocates more bandwidth as the throughput increases. The spikes are shown when the bandwidth is increased. The reason is that at the moment the bandwidth is increased, the queue is full and the incoming rate is lower than the outgoing rate. Thus, the spike appears until the number of packets in the queue decreases and becomes stable. Note that initially, it takes about 150 seconds for the 70 Mbps bandwidth to be allocated to the Default traffic. This time can be reduced by decreasing the polling interval of the controller.

At about 200 sec., Class 1 starts to increase the throughput to 74 Mbps. Soon after it starts to increase, the controller transfers the bandwidth from Default queue to Class 1 queue at Queues2 port until the equilibrium is reached at 60 Mbps for Class 1 and 30 Mbps for Default. The other 10 Mbps is allocated to Class 2 queue. At about 500 sec., Default starts to decrease the throughput back to 6 Mbps. Soon after it starts to decrease, the controller transfers the unused bandwidth from Default to Class 1.

Experiment 2: The purpose of this experiment is to compare the difference between our priority function algorithm and the Weighted Fair Queuing (WFQ) algorithm. Priority function can achieve the similar effect as WFQ; that is, the weights of two classes can be set so that they are consistent regardless of the available bandwidth. In addition, priority function can also be used so that the weights would differ based on the available bandwidth.

Figure 6.14 shows the functions where the ratio of the priority of the Class 1 function ($f(x) = 1500/x$) and the Default function ($f(x) = 750/x$) is consistent by 2 to 1, regardless of the throughput. Using this set of functions achieves the similar effect as WFQ where the ratio of the weight of the Class 1 traffic and the Default traffic is always 2 to 1.

Figure 6.15 shows the functions where the ratio of the Class 1 function ($f(x) = 1500/x$) and the Default function ($f(x) = 50/log(x)$) is not consistent (i.e., as the bandwidth increases, the ratio reaches approximately 1 to 1). With this *log* function, when the available bandwidth is scarce, it prioritizes the Class 1 traffic. However, when the available bandwidth is plenty, the priority of the two classes would converge. One of the instances this feature would be useful is when the network administrator wants to enforce a policy so that when the bandwidth is scarce, they want to allow the medical traffic to have higher precedence; however, when the bandwidth is plenty, they want to allow the default traffic to have better bandwidth ratio so that the user experience will improve (e.g., researchers can have faster access to the Internet to do research). Therefore, priority functions provide the network administrator more flexibility to prioritize the traffic than using the WFQ. In general, this flexibility is useful because it can enforce more complex policies and meet more complex demands than by simply allocating fixed weights.

The setup of this experiment is shown in Figure 6.16 where the Default, Medical 1 (Class 1) Client, and Medical 2 (Class 2) Client upload at the same time. The first part of this experiment shows the case when the functions are configured so that the bandwidth allocation follows the similar pattern to WFQ (i.e., It uses the functions in Figure 6.14). Figure 6.17 shows the throughput transition measured at the three servers. The first equilibrium is reached around 200 sec to 450 sec when the Class 2 traffic, which has the highest priority, obtains 70 Mbps. During this time, the throughput ratio between Class 1 and Default is 2 to 1 (20 Mbps to 10 Mbps). The second equilibrium is reached around 500 sec to 700 sec when the Class 2 traffic is reduced to 35 Mbps, which allows the other two classes to compete for the remaining bandwidth. During this time, the throughput ratio is still 2 to 1 (40 Mbps to 20 Mbps). The third equilibrium is reached around 750 sec to 900 sec when the Class 2 traffic is reduced close to 0. During this time, the throughput ratio is still 2 to 1 (60 Mbps to 30 Mbps). Thus, priority functions can achieve the similar effect as

WFQ where the relative ratio, or the weights, of two types of traffic is consistent.

In this example, the ratio of the Class 1 and the Default was 2 to 1 for all three equilibriums. However, note that the ratio cannot be 2 to 1 at other available bandwidth values. For example, if the available bandwidth is 40 Mbps, then the ratio would be 3 to 1 as shown at time 500 sec in Figure 6.17. However, if the granularity for the bandwidth transfer unit becomes smaller (it is 10 Mbps in all of our experiments), the ratio would converge to 2 to 1, or the ratio indicated by the priority functions.

Also note that it takes about 200 sec for the Class 2 traffic to obtain 70 Mbps in this experiment. This is because we try to show the bandwidth transition; hence, the load is increased by 10 Mbps in every 30 sec. However, even if the Class 2 traffic immediately started to generate 70 Mbps load at time 0, it would take 70 sec for it to obtain the 70 Mbps since the polling interval is set to 10 sec (i.e., 10 Mbps is transferred every 10 sec). This "warm-up time" does not exist in WFQ. However, in the priority function algorithm, the warm-up time can be reduced by reducing the polling interval. Although this method reduces the warm-up time, it would increase the overhead at the controller and every switch. Thus, the tradeoff must be considered carefully.

Depending on the configurations such as the polling interval, the warm-up time can be long resulting in a relatively slow transition of bandwidth from one traffic class to another. However, this slow rate of bandwidth transfer can be beneficial. For example, in Figure 6.17, it took about two minutes to release the 40 Mbps bandwidth allocated to Class 2 (from time 400 to 500 sec) and re-allocate it to Class 1 and Default. As with the warm-up time, this transfer time can be reduced up to certain extent by reducing the polling interval. However, the slow transfer, or gradual change in the available bandwidth, is helpful for applications that use streaming. For example, sudden bandwidth drop while downloading the video streaming may starve the buffer on the client device and pause the video. Since the bandwidth allocation, or re-allocation, to each queue immediately takes effect with WFQ, such problem is more likely to occur. On the other hand, the transfer time can be arbitrarily increased with the priority function algorithm. Thus, the priority function algorithm can reduce the likelihood of problems caused by sudden change in the available bandwidth.

The second part of this experiment shows the case when the $log(x)$ is used for the denominator of the Default function instead of x (i.e., it uses the functions in Figure 6.15). It allows the priority, or the weight, of the Default traffic to converge to that of the Class 1 traffic as the available bandwidth increases. Every other factors are kept the same as the first part. Figure 6.18 shows the throughput transition measured at the three servers. The first equilibrium is reached around 200 sec to 450 sec. During this time, the throughput ratio between Class 1 and Default is 2 to 1 (20 Mbps to 10 Mbps). The second equilibrium is reached around 500 sec to 700 sec when the Class 2 traffic is reduced to 35 Mbps. During this time, the throughput ratio is still 2 to 1 (40 Mbps to 20 Mbps). The third equilibrium is reached around 750 sec to 900 sec when the Class 2 traffic is reduced close to 0. During this time, the throughput ratio is changed to 5 to 4 (50 Mbps to 40 Mbps). This result shows that the priority functions can allow the weights of the traffic to change based on the available bandwidth, which is not the case with WFQ.

Experiment 3: This experiment shows the behavior of the bandwidth allocation when the Medical 1 Client and Medical 2 Client download at the same time (Figure 6.19). Figure 6.20 shows the throughput transition measured at the two clients. The equilibrium is at 50 Mbps for Class 2 and 40 Mbps for Class 1. In this experiment, Class 2 received more bandwidth than Class 1 because its priority is defined to be higher with the functions. The gap is smaller compared to the case of Default and Class 1 because the difference of priority is smaller.

Experiment 4: This experiment shows the behavior of the bandwidth allocation when the Default, Medical 1 Client, and Medical 2 Client download at the same time (Figure 6.21). Figure 6.22 shows the throughput transition measured at the three clients. At first, Default and Medical 2 client download at over 70 Mbps. Since their traffic do not share the same port, they both get 70 Mbps. When Medical 1 client starts to download, it takes away bandwidth from Medical 2 at Queues4 port and from Default at Queues2 port. Eventually, the equilibrium is reached around 400 sec. Then at about 600 sec., Class 2 traffic starts to decrease to 4 Mbps. After Class 2 traffic is decreased, the equilibrium is reached at Queues2 port.

Note that between 400 to 600 sec., although 90 Mbps is available at Queues2 port, Default and Class 1 queues are filled up at only 40 Mbps rate. The algorithm actually allocates 50 Mbps to Class 1, but since the bottleneck is at Queues4, it can only use 40 Mbps of it. One way to mitigate this bandwidth waste is to decrease the allocating unit, say to 5 Mbps. This method helps to satisfy demand from every class in finer grained manner. It does not require any additional implementation because it only takes one parameter to be changed. However, it only reduces the wasted bandwidth; it does not completely eliminate the waste. It also increases the overhead of allocation and release. Another way is to add the monitoring algorithm to check if the allocated bandwidth is actually used. If not, the algorithm can decide to punish the class by not allocating more bandwidth to it for a certain amount of time. This method solves the problem, but the algorithm becomes more complex.

Experiment 5: This experiment shows the behavior of the bandwidth allocation when the Default, Medical 1 Client, and Medical 2 Client upload at the same time (Figure 6.16). Figure 6.23 shows the throughput transition measured at the three servers. At first, Default uploads to over 70 Mbps. At about 200 sec., Medical 1 starts to upload to over 70 Mbps. The equilibrium is reached at Queues1 and Queues2 port. Then at about 500 sec., Medical 2 starts to upload, taking away the bandwidth of Default and Class 1 at Queues2 port.

Experiment 6: This experiment simulates the DoS attack to the medical server (Figure 6.24) while the client is uploading to the same medical server. Figure 6.25 shows the throughput transition measured at the server. Initially, Class 2 client uploads to the server at over 70 Mbps. At about 300 sec., traffic is generated from the Internet to the same server at over 100 Mbps. However, the uploading rate from the attacker is limited to 20 Mbps and it does not affect the throughput of Class 2 (with chosen priority functions).

Experiment 7: The setup of this experiment is the same as Experiment 1 (Figure 6.12). We generated TCP traffic from the two clients instead of UDP. Note that unlike the previous experiments, the TCP algorithm changes the load generated by the clients. Figure 6.26 shows the throughput transition measured at the server. The two clients started to generate the traffic simultaneously and continued for 700 sec. Before 200 sec, Default traffic obtained higher bandwidth, but the bandwidth was transferred to Class 1 traffic after 200 sec because of its higher priority. At about 300 sec, the same equilibrium as Experiment 1 was reached; Default obtained 30 Mbps and Class 1 obtained 60 Mbps. This result shows that our transfer algorithm also works with TCP.

6.1.6 Contributions

The main advantage of this algorithm is the flexibility. The network administrator can adjust the functions to precisely represent the relative priority of traffic. Because flow entries are used to allocate traffic to the queues, it can be done in a simple and dynamic manner even when there are many queues. The network administrator can also adjust the bandwidth section range and polling interval to come up with the most appropriate configuration to realize the policies. Finally, unlike with WFQ, the priority functions allow the weights to fluctuate based on the available bandwidth.

Another advantage is that given some available bandwidth, configuring the equilibrium bandwidth between two classes is as easy as configuring the WFQ. Figure 6.27 and 6.28 show the two priority functions used in experiment 1. In this experiment, we showed that the equilibrium bandwidth is 60 Mbps for Class 1 traffic and 30 Mbps for Default traffic.

We can find the priority functions that achieve this equilibrium with the following method. First, find the area under the curve adjacent to the maximum bandwidth of the two types of traffic. They are indicated as a and b in Figure 6.27; indicated as A and B in Figure 6.28. Use an arbitrary function, say $50/x$, for the first queue. Then, adjust the other function so that $b < A$ and $B < a$. In this case, $100/x$ satisfies these conditions.

6.2 Hardware Resource Guarantee

The LXC containers of SeaCat guarantee hardware resources to the applications running inside. LXC uses `Cgroups` so that CPU, memory, block I/O, and other hardware resources are allocated to each container. CPU and memory are two of the most important hardware resources for any application. Our evaluation shows that it is possible to allocate these resources to our containers.

6.2.1 Experiment Setup

We used Emulab testbed [46] to take advantage of the architecture with many cores and large memory. The machine we used has the Intel CPU with eight cores and about 12 GB of total memory. We have configured our default and application container on this machine.

To generate load from the default container, we used `stress`, which is the tool used in Linux to stress test a system mainly regarding the CPU and the memory. We used it to spawn worker threads that conduct computational intensive calculations (calling `sqrt()`). We also used it to allocate and keep certain amount of memory. To measure the effect from `stress`, we used another tool called `sysbench`, which is the tool used in Linux to measure the system performance. We used it to measure the CPU performance by measuring the time it took to conduct prime number calculations.

6.2.2 CPU Core Allocation

Allocating separate CPU core to each container is an effective way to guarantee the CPU resource. Figure 6.29 and 6.30 show the experiment setup. Figure 6.29 shows the case when two cores are allocated to the default container and the other two are allocated to the application container. Figure 6.30 shows the case when four cores are shared by the two containers.

We spawned 10 threads that repeatedly executes CPU intensive calculation (calling `sqrt()`) using `stress` from the default container. At the same time, we ran `sysbench` to measure the time it takes to conduct a fixed amount of calculation in the application container. In this case, `sysbench` calculated prime numbers up to 20,000 with 10 threads.

We spawned 10 threads in both `stress` and `sysbench` to keep all cores busy. We have conducted this experiment by allocating different/same cores to each container.

Figure 6.31 shows the results. Blue bars with horizontal stripes show the average time it took for `sysbench` to calculate the prime numbers for 10 times when no other container is running. This works as the base case. Orange bars with tilted grid show the average time when the cores are not shared as in Figure 6.29 and the `stress` threads are running. Red solid bars show the average time when cores are shared and the `stress` threads are running. Thin black bars show the 95th percentile. We tested using 1 to 4 cores. The results show that the calculation time at least doubled when the cores are shared in all number of cores. It also shows limited impact from other container when separate cores are allocated. This means using the separate cores for different containers is an effective way to guarantee the CPU resource.

6.2.3 CPU Bandwidth Allocation

Mobile devices, especially smartphones, usually do not have many cores. If the device has only one core, allocating separate core to each container is not possible. CPU bandwidth allocation provides finer-grained control over the CPU resource even there are only a few cores in the machine. To do so, `Cgroups` assigns a relative share of CPU bandwidth to each container. It can set the priority to a group of processes, in our case containers, so that the scheduler favors the higher priority processes and they receive more CPU time.

Figure 6.32 shows the results. Blue bars with horizontal stripes indicate the average prime number calculation time (up to 40,000) in the application container when there is no `stress` in the default container. As in the core allocation experiment, the average was taken over 10 calculations. CPU bandwidth is uniformly allocated, which means the bandwidth ratio between the default and the application container is 1:1. This works as the base case. Orange bars with tilted grid show the time when the `stress` runs in the default container while maintaining the 1:1 ratio. Red solid bars show the time while the `stress` is running and when the ratio is changed to 1:2, which means twice more bandwidth is allocated to the application container. We tested these experiments in the case of 1, 2, 4, and 8 cores.

The results show that the bandwidth allocation is effective for every scenario. Especially in 1 core case when the core allocation is not possible, it showed about 24% decrease in time. The strength of this method is that since the ratio can be changed to any value, it is possible to allocate the CPU resource with much finer-grained manner than the core allocation method.

6.2.4 Memory Allocation

Figure 6.33 shows the experiment setup to run the memory allocation experiment. `Cgroups` can set the memory-use limit by each container. We used `stress` in default container which tries to obtain 2GB of memory using `malloc()` calls. It also keeps the obtained block in memory by re-dirtying so that the OS does not remove it. We measured the available memory in the application container with `/proc/meminfo`, which stores various memory-related status.

Figure 6.34 shows the results. The blue bar with horizontal stripes indicated by NS, NL shows the available memory in the application container when there is no `stress` in the default container. This is used as the base case. The orange bar with tilted grid indicated by S, NL shows the available memory when `stress` obtains 2GB of memory. Since no limit is imposed for the default container, the available memory for the application container is reduced by approximately 2GB. The three red solid bars show the available memory when the `Cgroups` imposes the memory limit to the default container. The limits are indicated by the value on the x-axis. They show that `Cgroups` can accurately impose memory limit to a container.

These results, combined with the CPU resource guarantee, show that LXC can guarantee two of the most critical hardware resources to the desired container.

6.3 Evaluation of Namespace Isolation

A namespace is an abstraction of the system resources controlled by the Linux kernel. There are six namespaces; IPC, network, mount, PID, user, and UTS. Processes that are in different namespaces do not share the resources; hence, it appears to those processes that they own the isolated instance of each resource. The changes made to the resources from one namespace do not appear to the processes in another namespace. LXC uses all of the six namespaces to create the isolated containers. We will evaluate these namespaces to show that containers appropriately create isolated environment for the purpose of our solution.

The easiest way to check the namespace isolation between the root and a container is to list the symbolic links in `/proc/self/ns` directory in verbose format. The following terminal output shows the list of symlinks in root namespace.

```
kano@appct:~$ ls -l /proc/self/ns
total 0K
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 net -> net:[4026531980]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 pid -> pid:[4026531836]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 user -> user:[4026531837]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 uts -> uts:[4026531838]
```

The following shows the symlinks in one of our containers.

```
kano@default:~$ ls -l /proc/self/ns
total 0
lrwxrwxrwx 1 kano cs6480 0 Mar 28 09:25 ipc -> ipc:[4026532312]
lrwxrwxrwx 1 kano cs6480 0 Mar 28 09:25 mnt -> mnt:[4026532310]
lrwxrwxrwx 1 kano cs6480 0 Mar 28 09:25 net -> net:[4026532315]
lrwxrwxrwx 1 kano cs6480 0 Mar 28 09:25 pid -> pid:[4026532313]
lrwxrwxrwx 1 kano cs6480 0 Mar 28 09:25 user -> user:[4026531837]
lrwxrwxrwx 1 kano cs6480 0 Mar 28 09:25 uts -> uts:[4026532311]
```

The ten digit numbers towards the right are the inode numbers. An inode number is the reference to an inode. An inode is the table that stores the metadata and the pointer to the content of a particular file. The kernel ensures that if the two processes are in different namespace, their inode numbers in this directory would be different. In this case, the inode number for every namespace except user namespace is different between the root and the container. This means all the namespaces except for the user namespace are properly isolated. We will describe the solution to isolate the user namespace in the following subsection.

In the remainder of this section, we will describe and evaluate each namespace isolation to prove the effectiveness of the container isolation.

6.3.1 IPC Namespace

IPC namespace isolates the System V inter-process communication (IPC) mechanisms, namely, message queues, semaphore sets, and shared memory segments. We performed the following experiments to show that IPC through all three mechanism is isolated between containers.

To evaluate the message queue isolation, we wrote the sender-receiver program in C, which communicates the string messages typed by the user and sends them through a message queue. In this program, both the sender and the receiver uses the `msgget()` method. It uses a key (an integer) to create a message queue. The return value represents the Queue ID. When two processes in the same namespace use the same key to create a message queue, they can use this queue to exchange messages. When they are in different namespace (i.e., different containers), two different queues with different Queue IDs are created even though the same key is used. As a result, the processes cannot communicate if they live in different namespace.

To evaluate the semaphore sets isolation, we wrote a C program that is similar to the one described above. One process creates the semaphore with `semget()` method with a key. Another process accesses this semaphore with the same key to exchange values. A similar result was observed as in the experiment of message queue. When the processes are in a different namespace, they cannot communicate using the semaphore even if they use the same key.

To evaluate the shared memory isolation, we wrote a C program that is similar to the one described above. One process allocates the shared memory with `shmget()` method with a key. Another process calls `shmat()` method with the same key to attach to the same memory to communicate. Similar results were observed as in the experiment of message queue. When the processes are in different namespace, they cannot communicate using the shared memory even if they use the same key.

6.3.2 Network Namespace

Network namespace allows each container to have its own virtual network interface. Those interfaces can be configured independently (e.g., IP address, port number, routing table) from interfaces in other namespaces. Processes in the same container see the same network interfaces, but cannot see the interface of other containers. Because of this isolation, we were able to assign different IP address to each container. We used OVS to map each container's virtual interface to the corresponding Wi-Fi virtual interface.

6.3.3 Mount Namespace

Mount namespace isolates file system mount points seen by the processes. Processes in different mount namespace see a different file system hierarchy. Thus, the main purpose of using the mount namespace is to separate the file systems. Our containers are mounted to different mount point (e.g., `/containers/1/`, `/containers/2/`) using the mount namespace feature. In addition, we combined it with overlay file system to efficiently share the necessary kernel files in copy-on-write manner.

6.3.4 PID Namespace

PID namespace is used so that processes in the container can use the PID range independently from other containers or the root. For example, the `init` of a container and the `init` of the root can both have PID 1 because their PID namespaces are different. Note that every process in a container, including `init`, can be seen from the root namespace. For example, the default container's `init` can be seen from the root namespace running as PID 7726. This is because the PID namespace uses the nested hierarchical structure where a process can only see other processes running in the same PID namespace and the namespaces nested below its PID namespace.

6.3.5 User Namespace

A user ID (UID) is a positive integer assigned to each user in Unix OS. Having different user namespace means assigning nonoverlapping blocks of UIDs to each namespace. In the root namespace, 2^{32} different UIDs are available, but only a small portion of it is allocated to each user namespace. For example, suppose five people share one machine. Each person is allocated to their own user namespace, which typically has the UID range of 65,536. For example, person A has the range 100,000 to 165,536, person B has the range of 165,537 to 231,073, and so on. Within these ranges, each person can create more users.

Until recently (2014), LXC has not supported the user namespace isolation. In such a case, the UID block allocated to the container would be the same as the one in the root namespace. This means that if the process in the container somehow get access to any host resource through `proc`, `sys`, or some system calls, the process may be able to escape the container [17]. We learned that the entire 32-bit range of UIDs are allocated to our containers.

One solution is to use the unprivileged container. It allows the container creator to assign any range of UIDs to the container. Thus, the creator simply needs to allocate unused range in the root to the new container. That way, the processes in the container are kept in its own user namespace. We have installed the unprivileged container to our Linux testbed and allocated unused UID range to it. The following terminal output shows the list of symlinks in the root namespace.

```
kano@appct:~$ ls -l /proc/self/ns
total 0K
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 net -> net:[4026531980]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 pid -> pid:[4026531836]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 user -> user:[4026531837]
lrwxrwxrwx 1 kano cs6480 0 Mar 20 12:12 uts -> uts:[4026531838]
```

The following terminal output shows the list of symlinks in the unprivileged container.

```
root@ap1:~$ ls -l /proc/self/ns
total 0
lrwxrwxrwx 1 root root 0 Mar 20 18:12 ipc -> ipc:[4026532598]
lrwxrwxrwx 1 root root 0 Mar 20 18:12 mnt -> mnt:[4026532584]
lrwxrwxrwx 1 root root 0 Mar 20 18:12 net -> net:[4026532619]
lrwxrwxrwx 1 root root 0 Mar 20 18:12 pid -> pid:[4026532605]
lrwxrwxrwx 1 root root 0 Mar 20 18:12 user -> user:[4026532577]
lrwxrwxrwx 1 root root 0 Mar 20 18:12 uts -> uts:[4026532591]
```

All the inode numbers, including for user namespace, are different between the root namespace and the unprivileged container.

The unprivileged container used in this evaluation is installed from the template provided by the LXC team. Thus, the user namespace isolation feature must be understood and integrated into our custom container. We leave this effort as future work.

6.3.6 UTS Namespace

UTS namespace is used to isolate the hostname identifier. LXC uses it so that each container has its own hostname and NIS domain name (although we do not use NIS domain). Various initialization and configuration scripts use the hostname. With SeaCat, Trusted Daemon can assign an arbitrary host name to each container.

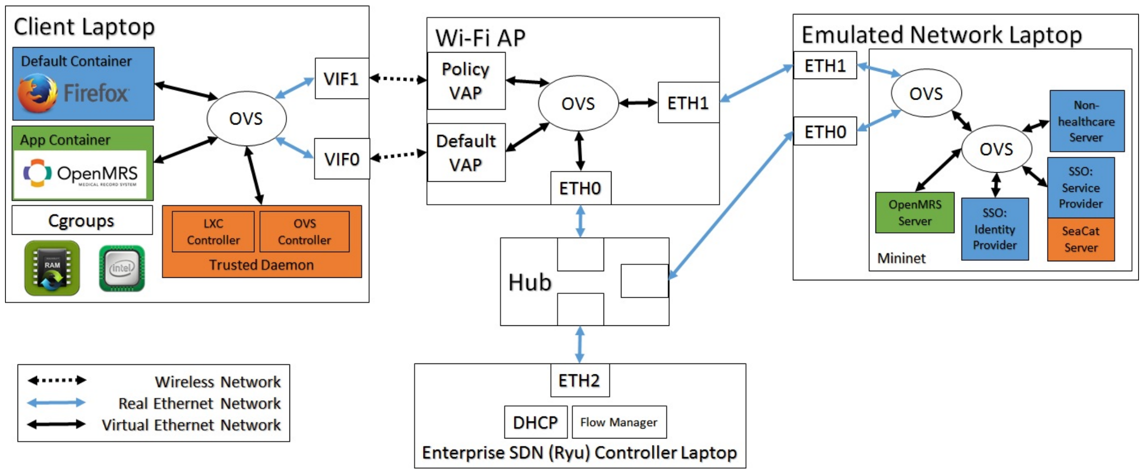

Figure 6.1: Implemented instance of SeaCat. Client laptop has two containers and the TD. It is connected to the SDN-enabled AP through virtual wireless interfaces. The emulated network laptop runs Mininet to emulate the enterprise network. The controller laptop runs the SDN controller for the enterprise network.

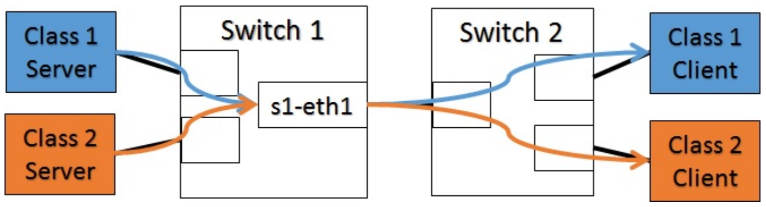

Figure 6.2: Topology with two servers and two clients. Queues are configured in `s1-eth1`.

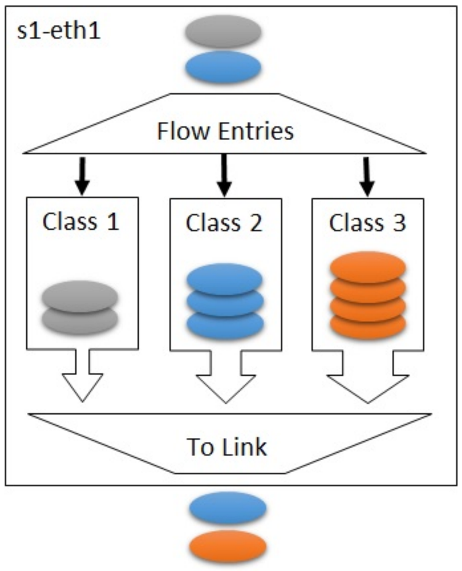

Figure 6.3: Three queues in `s1-eth1` are indicated as Class 1, 2, and 3. Width of the arrow in the bottom of the queue indicates the allocated bandwidth. A colored marble represents a packet. Flow entries direct incoming packets to appropriate queue. Packets are sent out at different rates.

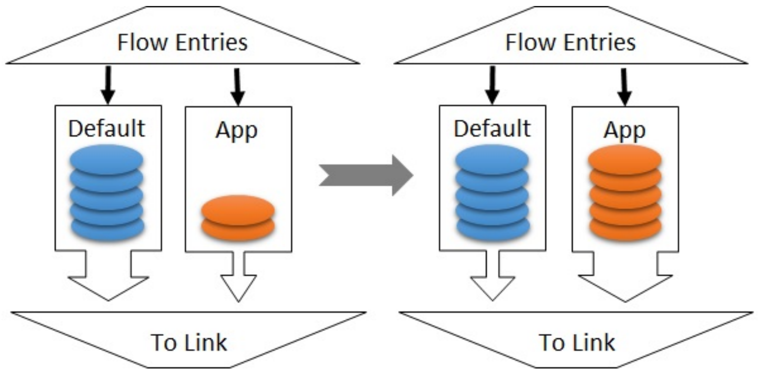

Figure 6.4: Assume that the algorithm favors the application traffic. Initially, larger traffic is generated from the default hosts; hence, larger number of marbles (packets) are going through the default queue. When the application traffic load increases, the allocated bandwidth is transferred to it, indicated by narrower arrow for the Default queue and wider arrow for the App queue. If both queues have large traffic load, larger bandwidth is allocated to more important traffic.

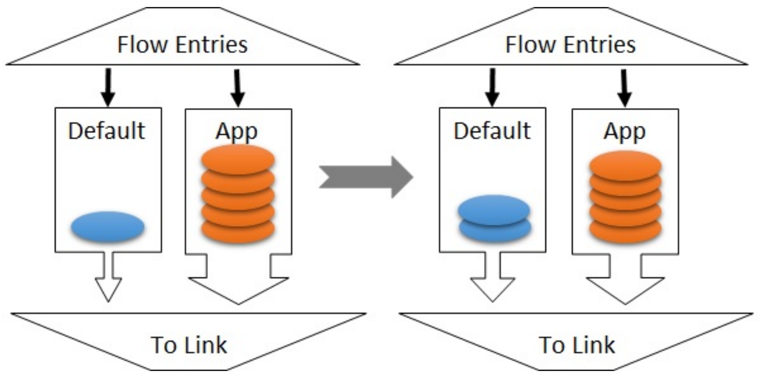

Figure 6.5: Initially, larger bandwidth is allocated to the App queue. When the default traffic load starts to catch up, bandwidth is transferred from the App to the Default queue. Although the application traffic is considered more important, the algorithm favors the queue with lower traffic load and likely to transfer available bandwidth to it if it needs more.

Table 6.1: Summary of CP and PP values calculated from the sample functions.

Queue ID	CP		Queue ID	PP
1	20		3	43
2	41		1	14
3	61			

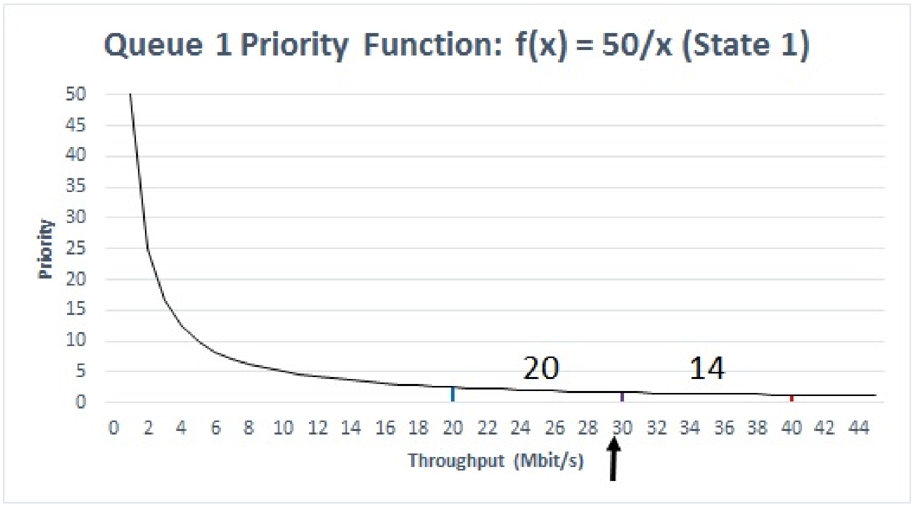

Figure 6.6: Sample priority function for Queue 1 at State 1 (before bandwidth transfer). Numbers indicate the area under the curve, which are the current priority and the prospective priority. Arrow indicates the current throughput.

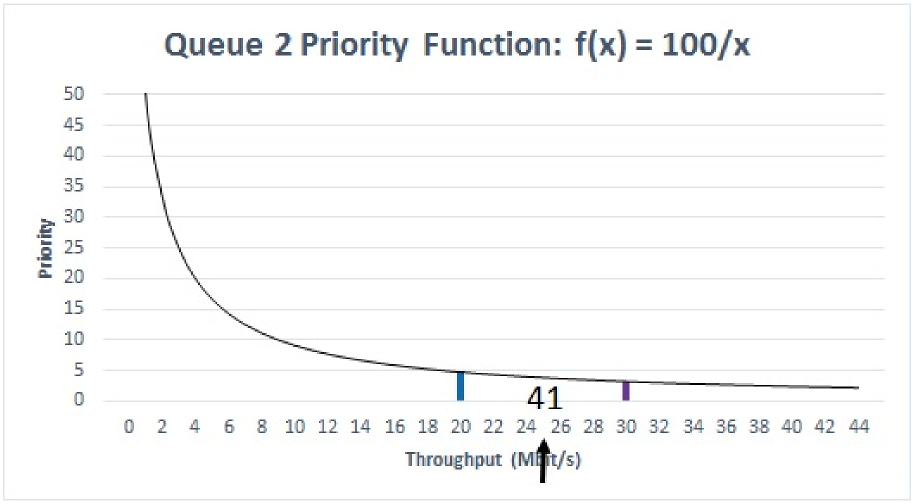

Figure 6.7: Sample priority function for Queue 2.

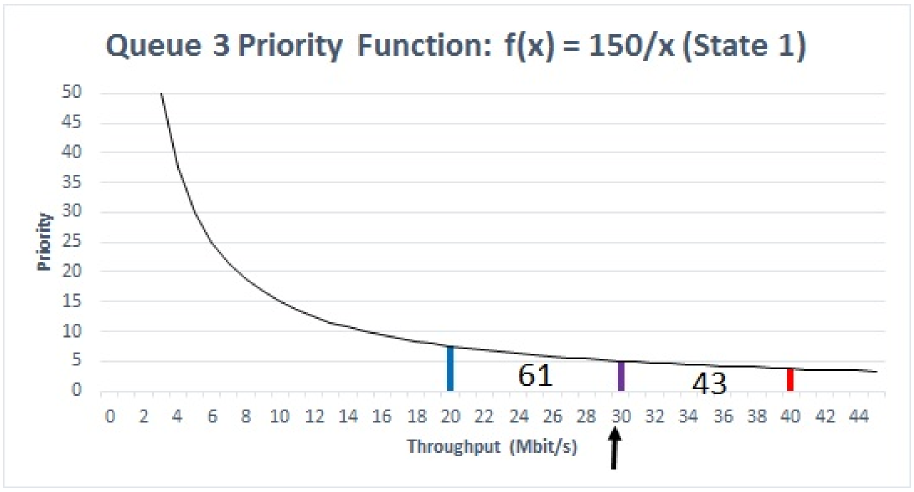

Figure 6.8: Sample priority function for Queue 3 at State 1 (before bandwidth transfer).

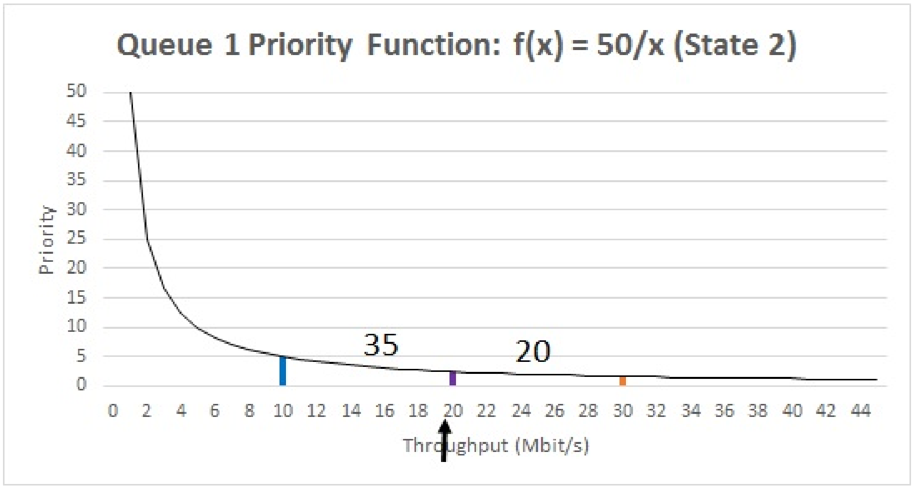

Figure 6.9: Sample priority function for Queue 1 at State 2 (after bandwidth transfer).

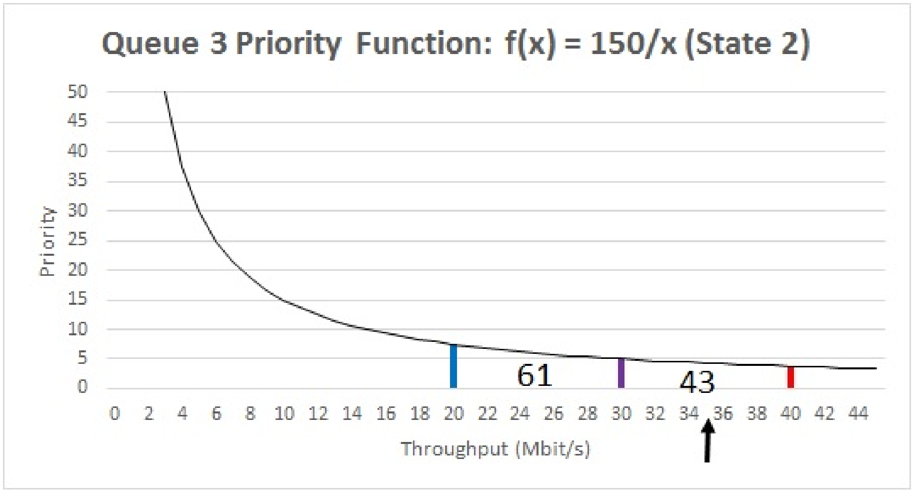

Figure 6.10: Sample priority function for Queue 3 at State 2 (after bandwidth transfer).

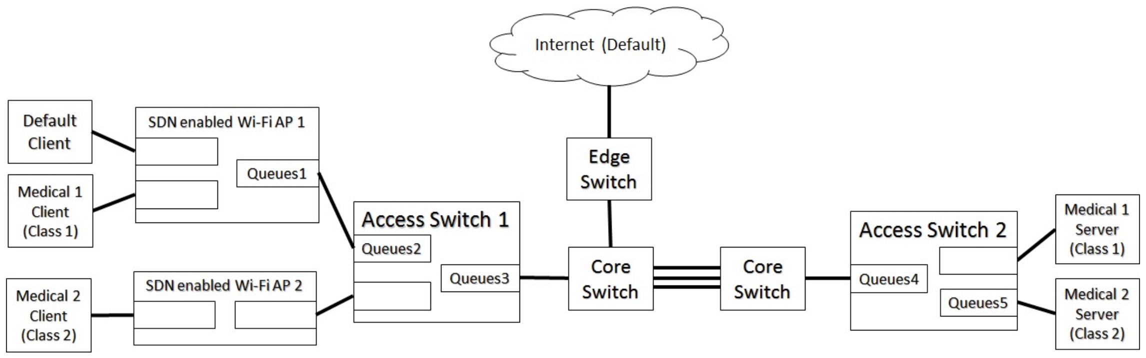

Figure 6.11: Enterprise topology created with Mininet for the experiment. Access switches are connected to the core switches. Edge switch works as the gateway to the Internet. Each client and server represent a set of hosts. Queues are configured in access switch and Wi-Fi APs.

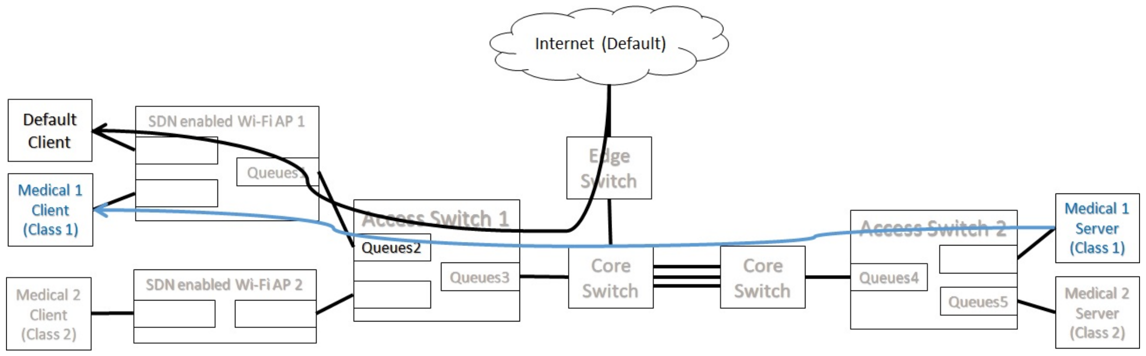

Figure 6.12: Scenario for Experiment 1. Default Client and Medical 1 Client download at the same time. Default downloads from the server in the Internet while Medical 1 downloads from Medical 1 server in the network. Queues are configured in Queues2 port.

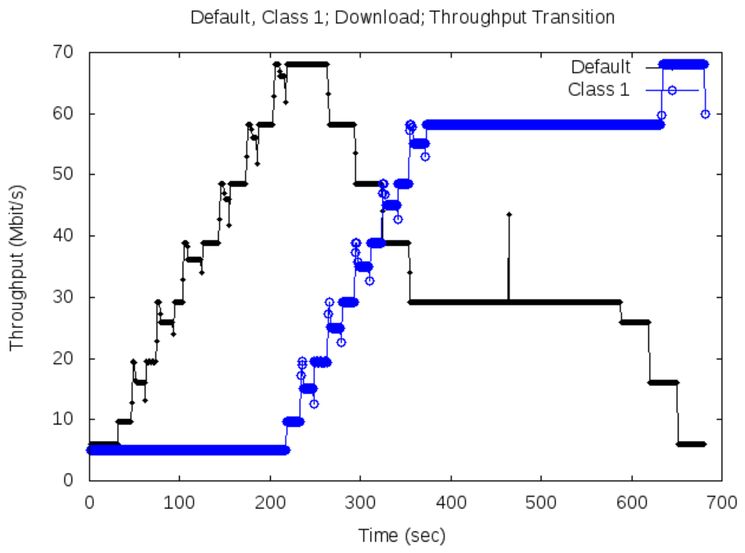

Figure 6.13: Throughput transition for Experiment 1. It shows the equilibrium between Class 1 and Default.

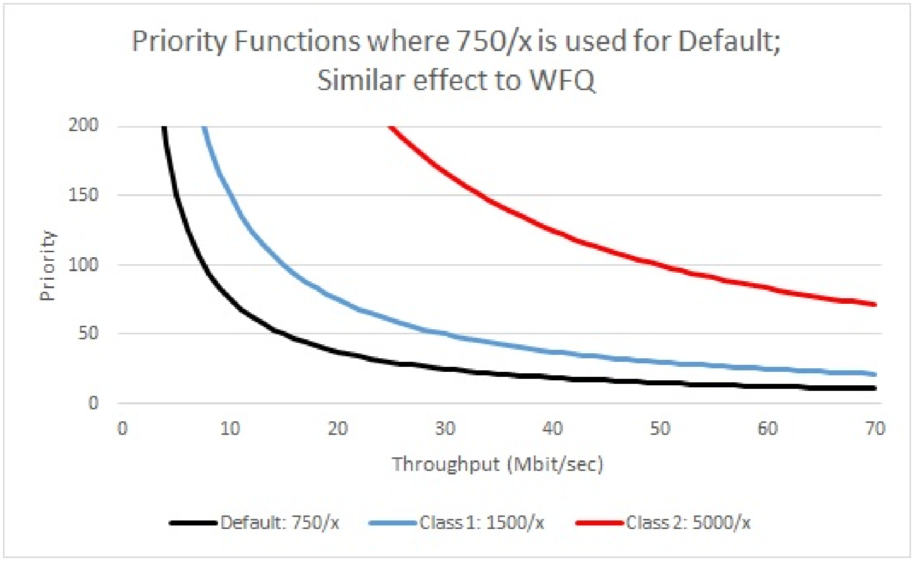

Figure 6.14: Priority functions for Default, Class 1, and Class 2 traffic. The ratio of the priority between the Medical 1 ($1500/x$) and the Default ($750/x$) is 2 to 1.

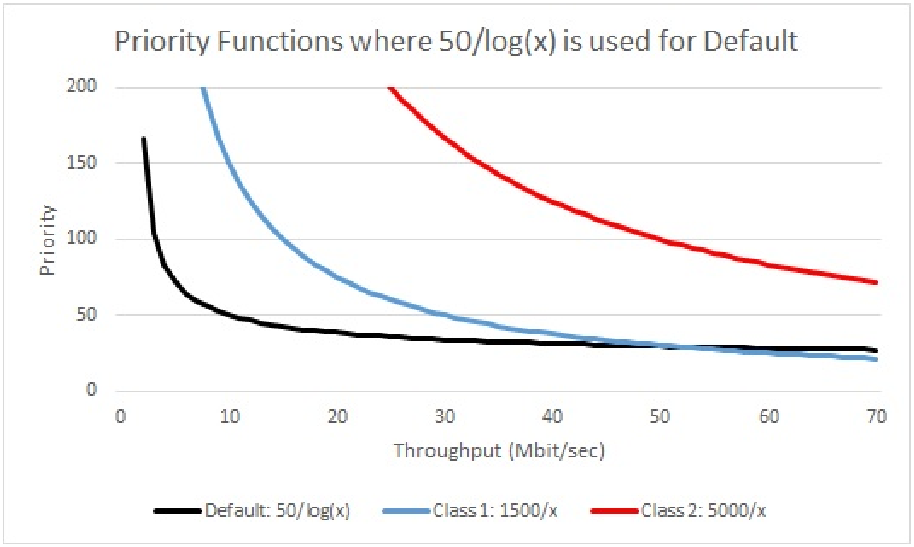

Figure 6.15: Priority functions for Default, Class 1, and Class 2 traffic. The ratio of the priority between the Class 1 ($1500/x$) and the Default ($50/log(x)$) varies based on the available bandwidth.

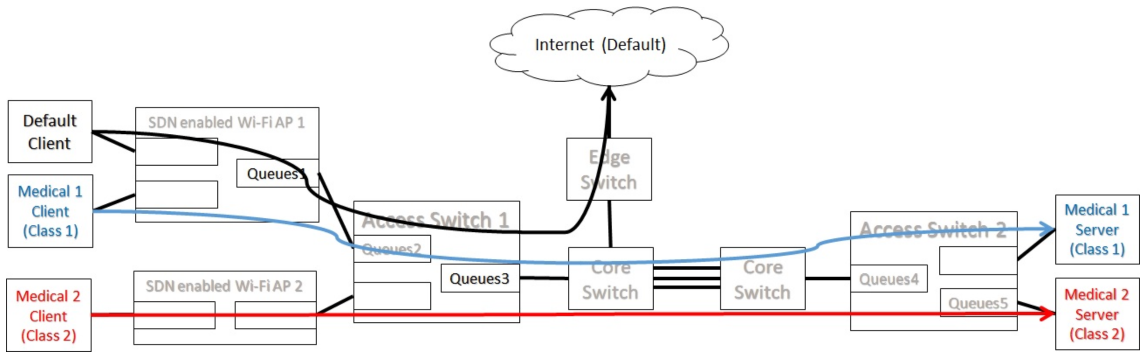

Figure 6.16: Scenario for Experiment 2 and 5. Default, Class 1, and Class 2 upload at the same time. Queues are configured in Queues1 and Queues3 port.

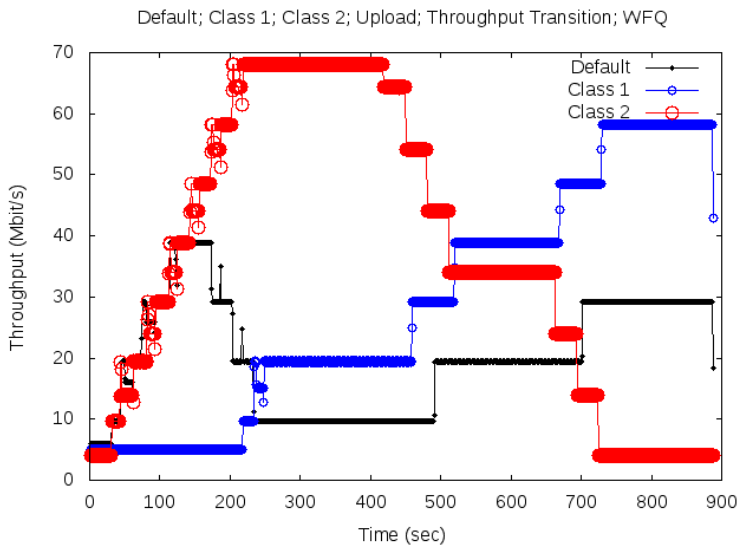

Figure 6.17: Throughput transition for Experiment 2, which achieves the similar result as WFQ during the equilibrium, or the time when there is no bandwidth re-allocation or transfer. The Class 1 traffic receives twice more bandwidth than the Default traffic at all of the three equilibrium.

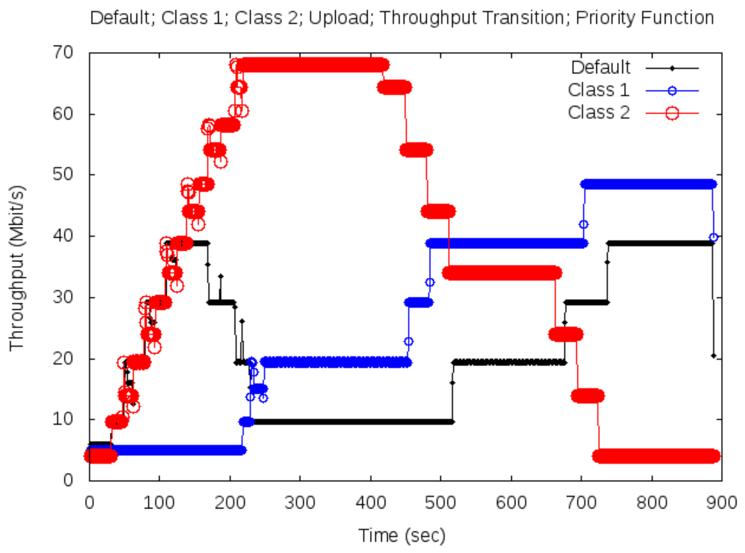

Figure 6.18: Throughput transition for Experiment 2, which achieves the variable ratio between the Class 1 and the Default. During the first two equilibrium period, the ratio is 2 to 1. However, during the third equilibrium period, the ratio is 5 to 4.

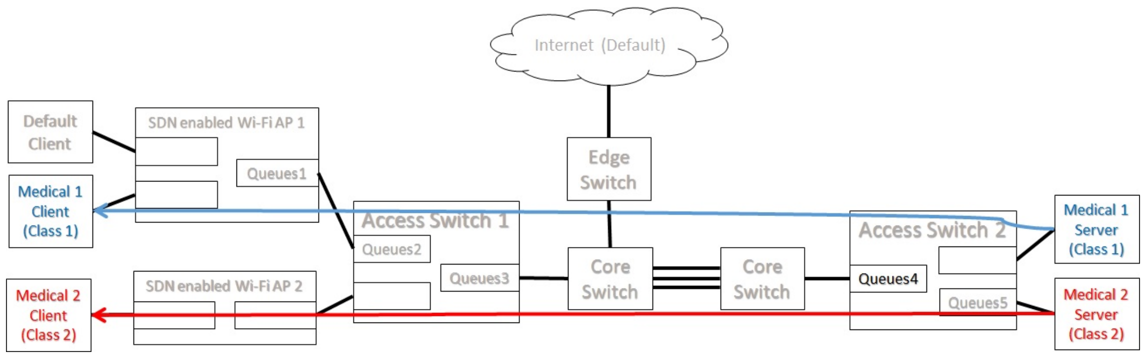

Figure 6.19: Scenario for Experiment 3. Class 1 and 2 download at the same time. Queues are configured in Queues4 port.

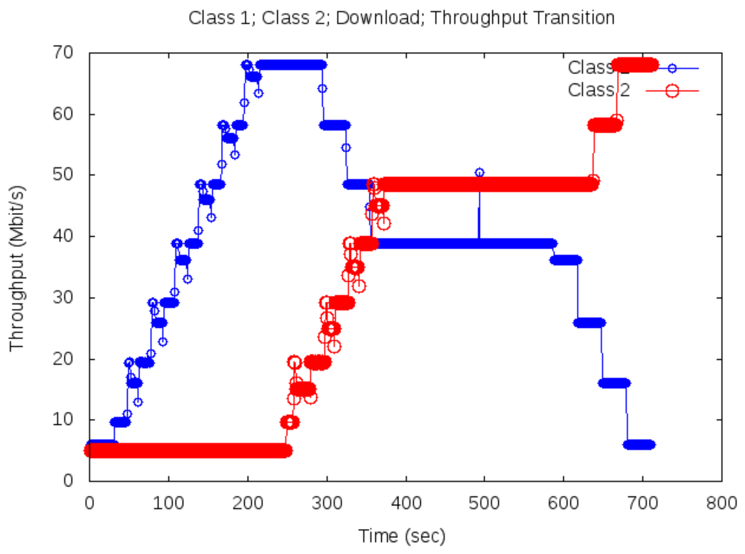

Figure 6.20: Throughput transition for Experiment 3. It shows the equilibrium between Class 1 and Class 2.

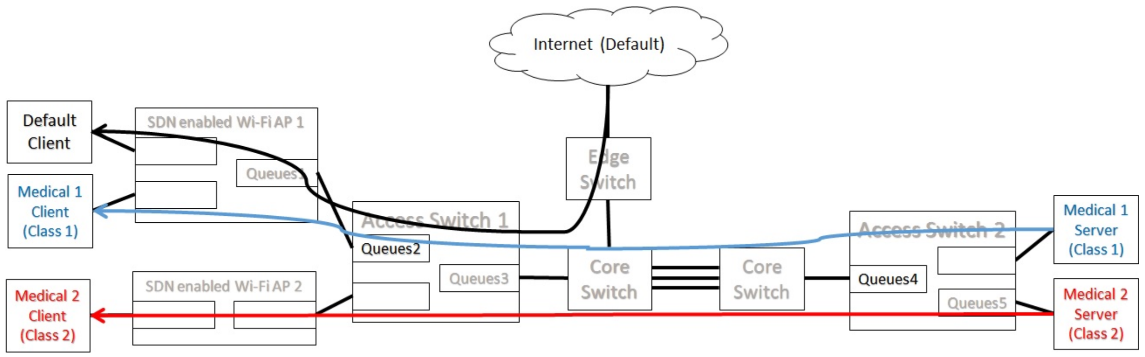

Figure 6.21: Scenario for Experiment 4. Default, Class 1, and Class 2 download at the same time. Default and Class 1 traffic are managed at Queues2 port. Class 1 and Class 2 traffic are managed at Queues4 port. Different queue configurations are applied to the two ports.

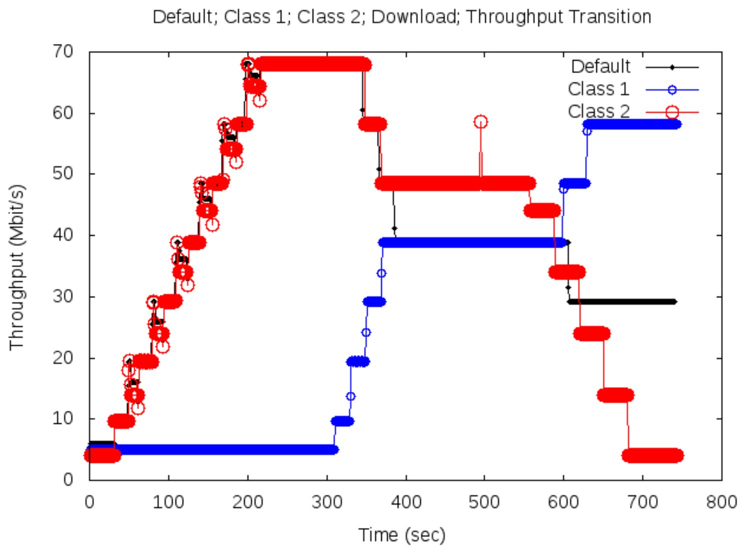

Figure 6.22: Throughput transition for Experiment 4. It shows how Class 1 affects both Default and Class 2 throughput. Default and Class 1 reach their equilibrium when Class 2's throughput goes down.

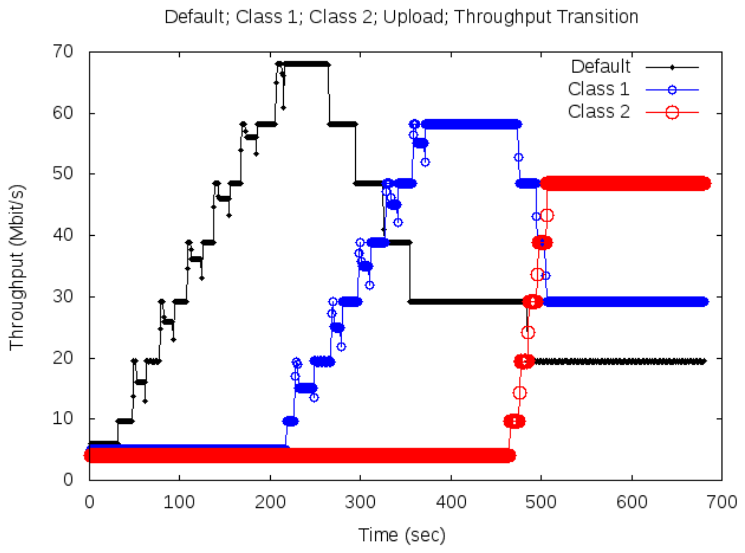

Figure 6.23: Throughput transition for Experiment 5. It shows the transition when three classes share the same port.

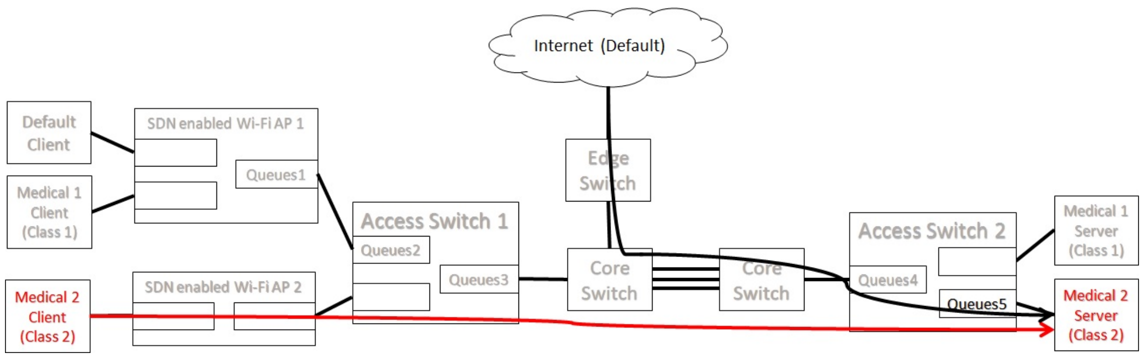

Figure 6.24: Scenario for Experiment 6. DoS attack traffic and Medical 2 traffic from the valid user arrive to Medical 2 server at the same time. Queues in Queues5 port allocate enough bandwidth to Medical 2 traffic.

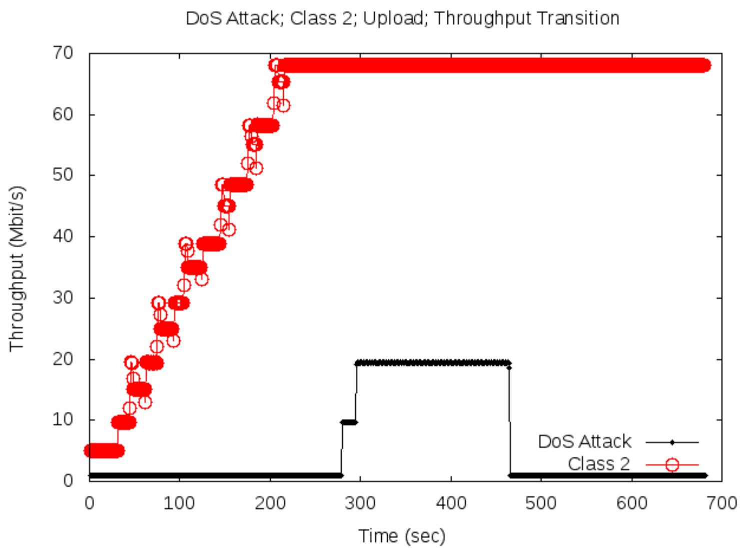

Figure 6.25: Throughput transition for Experiment 6. It shows how queues can be used to prevent DoS attack.

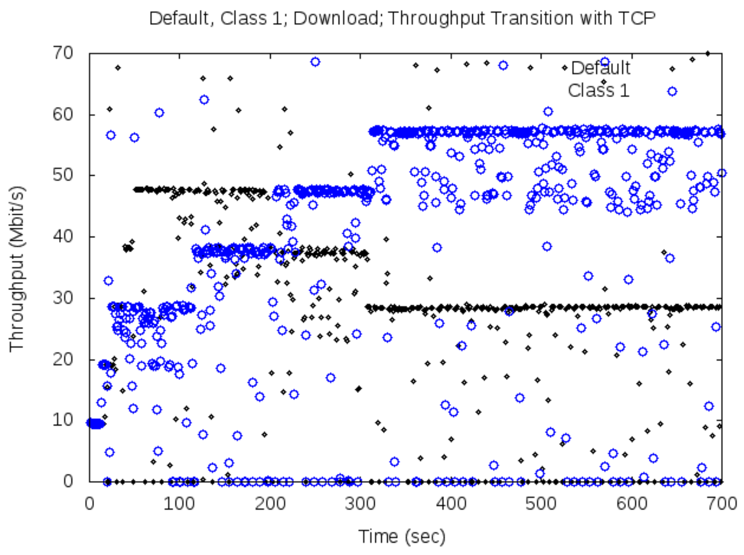

Figure 6.26: Throughput transition for Experiment 7, which uses TCP. It shows similar result as in Experiment 1. There is more fluctuation than UDP, but the equilibrium is reached on average.

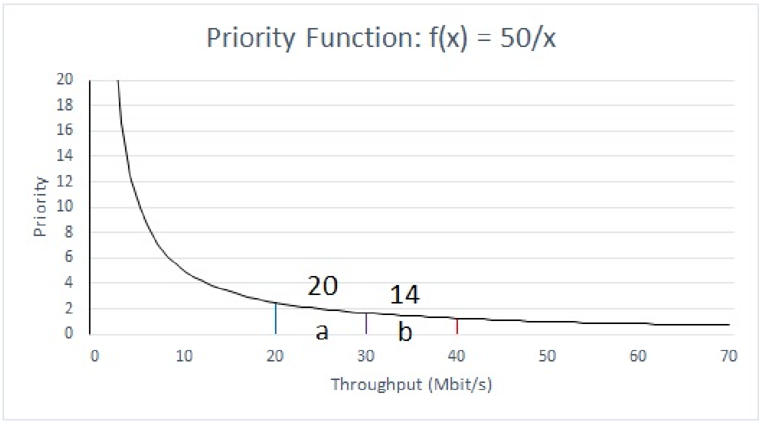

Figure 6.27: Priority function for $50/x$. Use the section area under the curve around the maximum bandwidth to find the function that achieves the desired equilibrium.

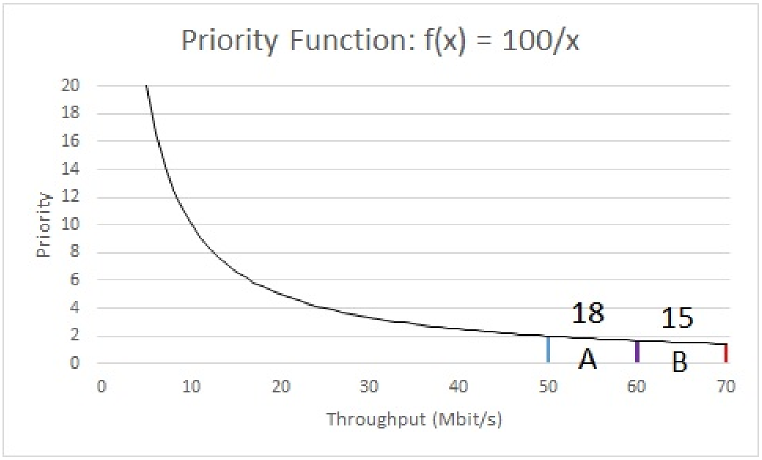

Figure 6.28: Priority function for $100/x$.

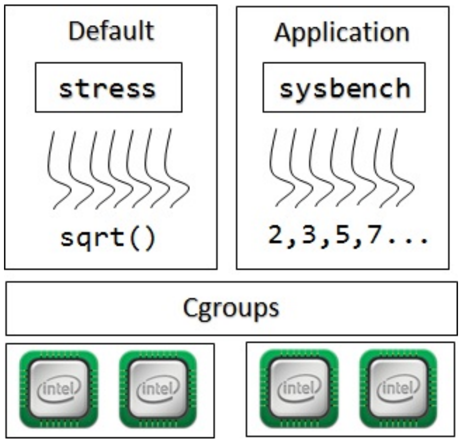

Figure 6.29: Separate cores are allocated to separate containers. `Cgroups` handles the allocation. `stress` spawns threads in the default container to repeatedly call `sqrt()` to exhaust the CPU resource. At the same time, `sysbench` in the application container measures the speed to calculate the prime numbers.

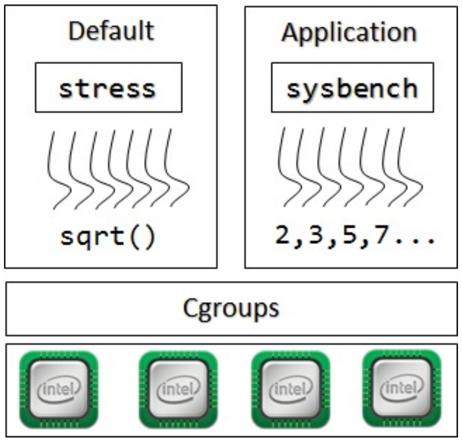

Figure 6.30: Containers share the same core. Same experiment as in the case when the cores are not shared.

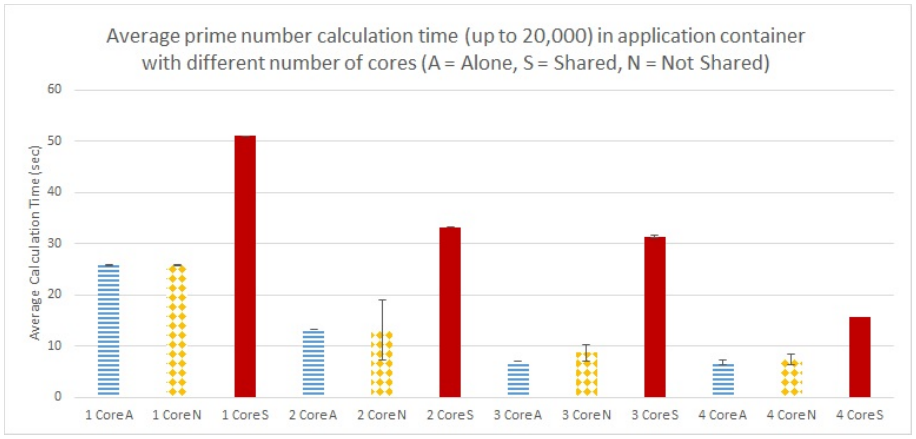

Figure 6.31: Experiment results of CPU core allocation. Blue bars with horizontal stripes show the calculation time without another container. Orange bars with tilted grid show the calculation time when separate cores are allocated. Red solid bars show the time when cores are shared. (e.g., 2 Core N means separate two cores, with total of four cores, are allocated to each container. 2 Core S means two containers share the same two cores.) Sharing the cores with other container significantly impacts the calculation time.

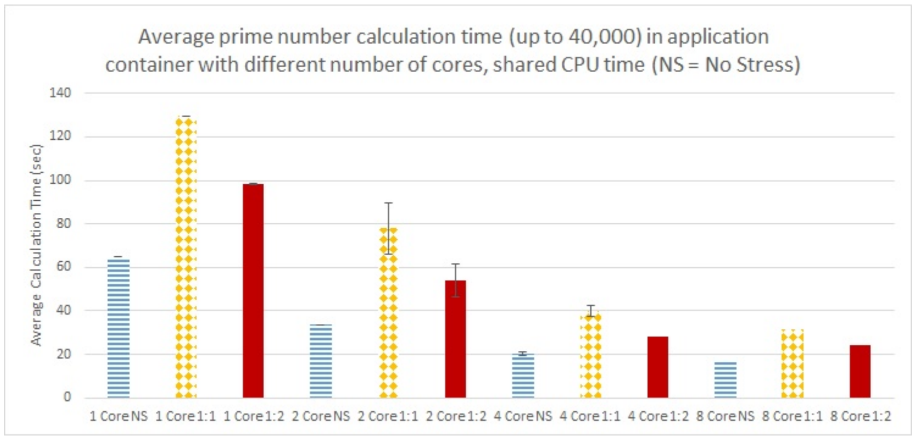

Figure 6.32: Experiment results of CPU bandwidth allocation. Blue bars with horizontal stripes show the calculation time without another container. Orange bars with tilted grid show the calculation time when the two containers are allocated the same share of bandwidth while `stress` is running on the default. Red solid bars show the time when twice more bandwidth is allocated to the application container. Allocating larger CPU bandwidth significantly reduced the calculation time.

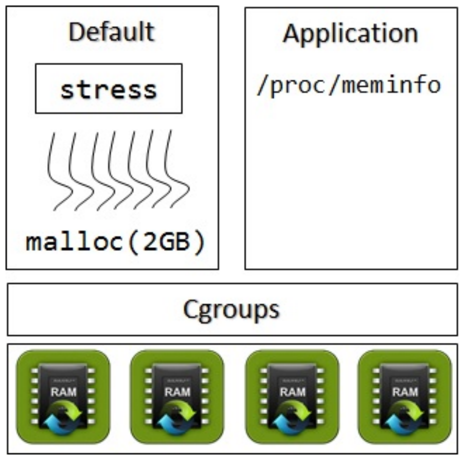

Figure 6.33: Memory resource is shared between the containers. In the default container, `stress` spawns threads to call `malloc()` that tries to obtain 2GB of memory. While it keeps the memory, `/proc/meminfo` was read from the application container.

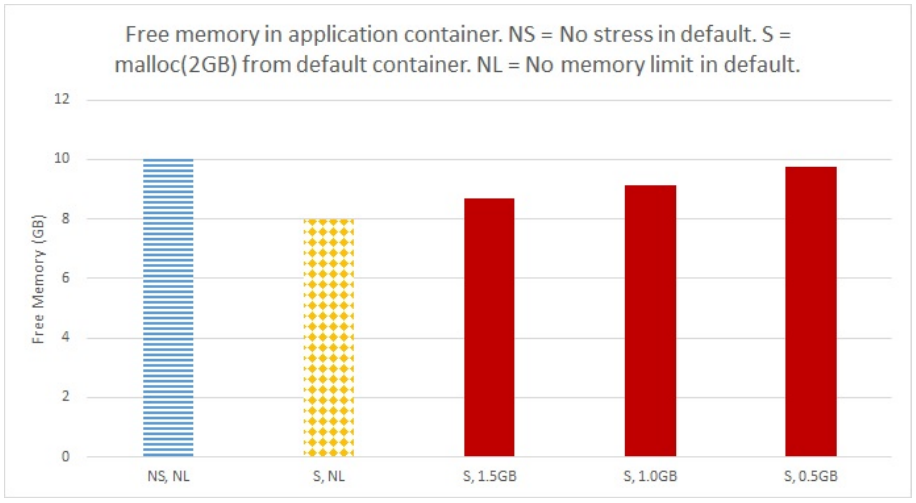

Figure 6.34: Experiment result of memory allocation. Blue bar with horizontal stripes shows the default free memory in the application container. Orange bar with tilted grid shows the free memory when the `stress` in the default container tries to obtain 2GB. Red solid bars show the free memory when `Cgroups` imposes the maximum bandwidth. For example, the leftmost red bar shows the case when `Cgroups` imposed 1.5GB limit on the default container. In this case, `stress` can only obtain up to 1.5GB memory.

CHAPTER 7

FUTURE WORK

7.1 Trusted Platform Module (TPM)

The most significant security issue with LXC is that containers share the same kernel. Although the risk is low, if the kernel is compromised, the malware in the default container can access the application container or the root namespace. TPM [18] technology can be used to validate the integrity of the kernel, which significantly mitigate this issue. TPM also allows the SeaCat server to ensure the integrity of the application that is about to be used through a remote attestation. Since the client device is a personal device, the device could be tampered in various ways. TPM reduces the risk of allowing a tampered device to access the secured network by allowing the SeaCat server to notice if the device is tampered in any way.

TPM is a chip installed on the motherboard; hence, this approach can only be used with the client device that supports it. The chip stores the private key that is not directly accessible from other hardware. It uses this private key to check the integrity of BIOS, boot loader, kernel, and the files specified by the user.

Figure 7.1 shows the preliminary design of how TPM technology can be used in SeaCat to assure the kernel and medical application. First, the client sends the signed hash of BIOS, kernel, Application A(medical application), TD, and the application container in addition to the username and password. The list of medical applications and files that make up the TD, container, kernel, and BIOS PCR (Platform Configuration Register) are stored in the SeaCat server. The server compares the hashes to ensure that none of these components are compromised. Finally, the server signals the controller to create the policy enforced context.

Figure 7.2 describes how this procedure works in more detail. TrouSerS [43] is the open source software stack that complies with TSS (TCG Software Stack) 1.2. It is used to send commands to the TPM. TrouSerS has the interface for userspace commands and the API for C programs. The userspace commands are limited for configurations (e.g., set ownership), encryption, decryption, and a few others. C programs are used to send various commands.

The TPM is used for two tasks. First is to sign the hash of BIOS, the files for kernel, TD, application container, and the application. To sign an arbitrary file, a C program must be written to do the following:

1. Generate the signing key.
2. Load the private portion of the key to the TPM.
3. Using the TrouSerS API, request TPM to hash the data.
4. Sign the digest with the loaded private key.
5. Save the signature to a file, which will be sent to the SeaCat server along with the public key.

To minimize the risk of leaking medical data from the client device, the medical application should store all the data to the server. However, if the application requires data to be stored in the client device, encrypting them when the container is not in use would reduce the risk. To encrypt data, sealing feature of the TrouSerS must be used. Sealing takes the following steps:

1. TrouSerS obtains the current value of PCR.
2. TPM seals the data with obtained PCR.

Note that sealing is the type of encryption that involves the PCR (state of the machine). Unsealing requires the password associated with the key used for the sealing (SRK). In addition, the current PCR value must match with the one calculated from the PCR used for the sealing. These procedures are handled with TrouSerS.

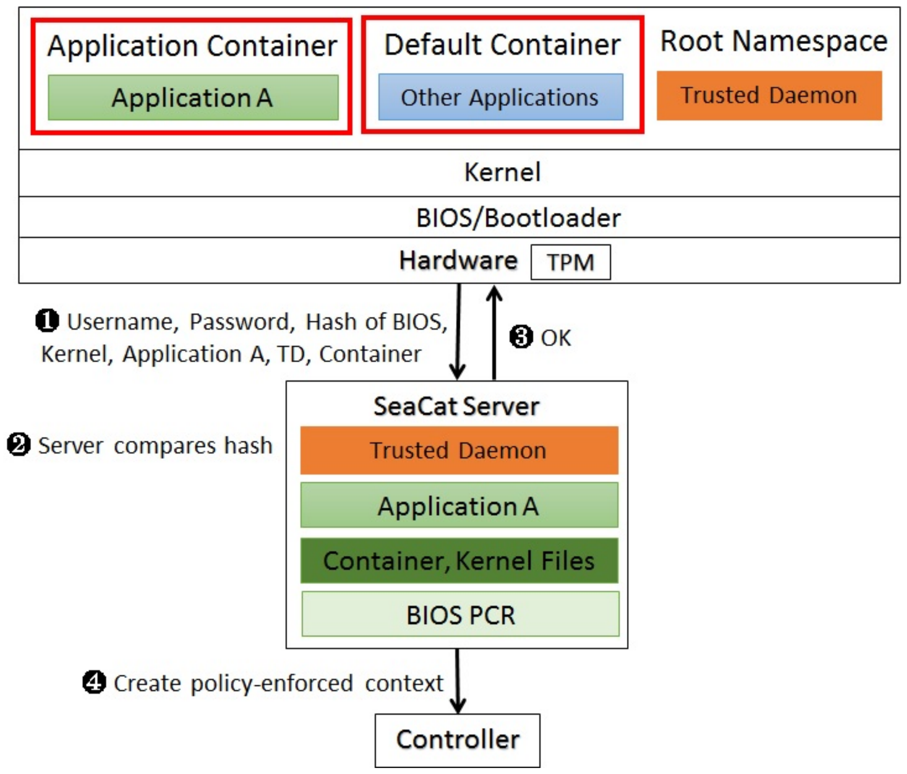

Figure 7.1: TPM checks the integrity (i.e., calculates the hash and sign) of BIOS, kernel, TD, application container, and the application. Client sends the signature to the SeaCat Server to authenticate. SeaCat server stores the files so it can check the signature. When the signature is validated, it signals the client to start the application. Finally, SeaCat server sends the signal to create the policy enforced context.

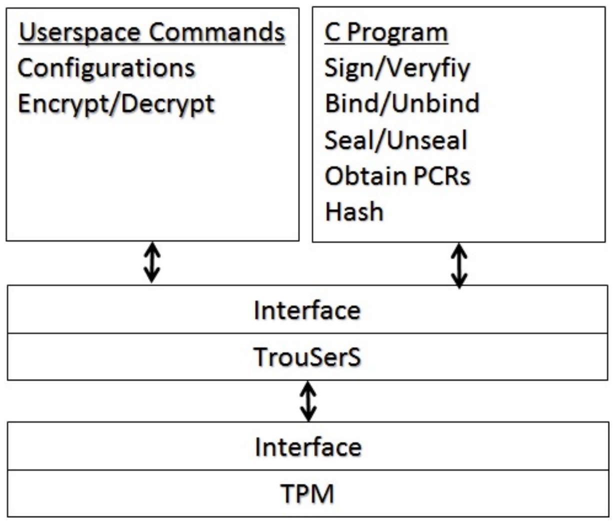

Figure 7.2: TrouSerS is the software stack that communicates with the TPM. It provides the interface, which can be used by tpm-tools commands and C programs. `tpm-tools` commands are used to configure the TPM and C programs are used to sign and encrypt.

CHAPTER 8

RELATED WORK

8.1 End-Point Isolation

There are multiple ways to achieve the end-point isolation. Docker [33] uses LXC for consistent development and deployment of the containers for specific purposes. Its main advantage is the container portability. A container can be saved as an image so that it can be easily shared with different hosts. Containers are lighter-weight than virtual machines because it does not have a guest OS. Docker runs on top of host OS and the containers share the kernel with other containers.

Docker uses AuFS (Advanced Multi-Layered Unification Filesystem), which mounts all the containers to the same mount point. Our system is more secure because containers are mounted at different mount point. This architecture prevents the containers from interacting with each other through the filesystem.

Cells [2] achieves end-point isolation in a mobile device. It can create multiple virtual phones (VPs) to isolate the environment. They use device namespaces to isolate the devices of the phone (e.g., framebuffer, GPU, power, virtual NIC, binder, sensor) for each VP.

With Cells, users can control which VP to start and stop. They can also control when to start and stop it. Our system does not merely achieves the isolation. It is more secure because the user must be authenticated with the external server before using the secured containers. In addition, SeaCat can provide finer grained security with less effort from the user. Unlike Cells, the user can simply start the application and SeaCat creates the custom environment and network context for a specific application.

Qubes OS [40] is the virtualization system that runs multiple VMs on the Xen hypervisor. Each VM represents a domain such as work or personal. Users create and start new domains from the trusted domain, *dom0*. Each domain is represented as a separate window on the desktop. Although users can simultaneously view the windows, the OS prevents malware on one domain to collect display event information from other domains. Quebes OS achieves this by isolating the Xserver for running domains.

Qubes OS can achieve similar isolation properties to SeaCat. However, our system uses containers so it is lighter-weight. Unlike Qubes OS, our system does not require full OS stack for each container.

Cloud Terminal [31] provides isolated environment to run secure application. One of the limitations is that since the applications run on the cloud, only the applications that do not require large network resource (e.g., online banking, email) can be used. Thus, their use case is more limited. In addition, with Cloud Terminal the application provider needs to migrate the application to the Cloud Terminals cloud. Our system is more practical because we do not require any changes to the application servers.

8.2 End-Point SDN

There are some published works of using OVS in a mobile device. One of them showed that it is possible to use OVS to dynamically control the network interface to use (e.g., Wi-Fi, 3G, WiMAX) [47]. OVS can be used to manage handover too [10].

meSDN (Mobile Extension of SDN) [27] uses SDN in similar way with our work. They implement the OVS in the mobile device to control the uplink traffic. A global controller located somewhere in the network negotiates with the local controller in the mobile device about the uplink policy.

meSDN architecture is similar to ours. However, our global controller not only enforces a policy to the client, but also enforces policies in the network; hospital LAN. Our local controller (i.e., Trusted Daemon) is more involved because it enforces policies to the client OVS and also manages containers.

8.3 Resource Allocation

The use of queues in switches have been studied for more than a decade. David Clark argued that it would be ideal to be able to control the available bandwidth for each user [9]. Weighted Fair Queuing (WFQ) has been shown to be effective [15]. WFQ has been used in the Wi-Fi link too [3]. CHOKe [38], RED-PD [29], Rainbow Fair Queuing [6], and Dynamic Token Buckt [26] all try to use queues to divide the flow at a switch and allocate appropriate bandwidth to each queue.

After OpenFlow has been invented, many studies have been conducted to use the OpenFlow to satisfy the QoS. The work that tries to maximize the QoE of video streaming [16] took the similar approach as our work. They use the function similar to our priority function to efficiently allocate the bandwidth to different types of streaming traffic. However, our approach is more generic because our priority function can be used for any type of traffic.

Falloc [20] uses OpenFlow to dynamically allocate bandwidth in a cloud with the concept of Nash Bargaining Solution (NBS). In NBS, the users make the sub-optimal decisions to maximize the group's total gain. It is different from our concept because our algorithm allows each user to act selfish to reach the bandwidth equilibrium. Other work such as PolicyCop [4], QoSFlow [25], and OpenFlow Virtualization Scheme [42] use queuing in OVS to satisfy QoS.

All the previous works described above prioritizes network resource only for part of the connection. Our work not only prioritizes the network resource, but also the hardware resource for the client device. We combine these features to construct the end-to-end connection so that resources are prioritized for the entire path between the client and the server.

8.4 Remote Attestation

Remote attestation is used in many platforms such as cloud and mobile devices. One of the novel approaches is to authenticate the application. The authors of Remote Attestation on Program Execution [19] developed the prototype to check the secured programs state whenever the program executes and uses relevant executable or data objects. The idea is that as the program continues to execute, its state will change so attesting only at the start of the execution is not enough. The authors of Semantic Remote Atestation [21] tried to apply the language-based security to the application. The language-based security involves parsing the code to determine whether the code will run as expected.

Both of these approaches would add security to SeaCat. Our work is different from these because our goal with remote attestation is to secure the entire client device with TPM rather than just the application. Even the behavior of the application is verified, a compromised kernel may leak the secured data from the file system, if the application stores data on the device. Thus, it is more effective to verify every layer of the system with TPM.

CHAPTER 9

CONCLUSION

SeaCat establishes end-to-end isolation and resource prioritization for connections for a medical application using end-point isolation and SDN. The solutions SeaCat provide are not only unique in the field of healthcare security, but also strengthen the effects of existing solutions. The main contribution of SeaCat is its ability to provide isolation and resource prioritization at the same time with the end-to-end context. Contexts are created by traffic forwarding rules, traffic prioritization, end-point isolation with containers, 802.1X, 802.11i, and VAP that secure the Wi-Fi link. Various features have been implemented and evaluated over the course of the project.

SeaCat uses LXC to isolate a medical application from others to secure its data and avoid interference. The advantage of LXC is that creating, deleting, starting, and stopping them are fast. We have selected the jobs to run in each container to customize the environment for the applications. LXC also provides hardware resource guarantees for each container. In addition to the containers, a client device has TD, which manages the containers and the SDN configurations of the client device. We solved the shared display problem on the client device by running the containers inside their own Xnest. SeaCat uses virtual interface on the AP and the client device to support multiple contexts in the air. It also uses the bandwidth allocation algorithm with queues to allocate network resource dynamically and efficiently. The link between the client and the AP can be secured by integrating 802.1X and 802.11i into our system. Finally, it allows the client to authenticate with the SeaCat server through Shibboleth SSO system.

With these features, SeaCat protects medical applications and their data from various attacks. LXC prevents secured data to be stolen from the client device. It uses Xnest to prevent information leakage from client's display event. It also prevents malware or other non-medical applications to deplete the hardware resources. SDN not only prioritizes the network resource, but also prevents DoS attack to medical application traffic. Shibboleth SSO prevents unauthorized access while allowing the network administrator to create context in the network.

One of the weaknesses of LXC is that containers share the same kernel. We believe that by using the TPM enabled client device, SeaCat server can perform a remote attestation to check its integrity to make sure that the device with compromised kernel is not using the medical application and its secured context.

APPENDIX

PRIORITY FUNCTION ALGORITHM

The following is the bandwidth allocation algorithm used in Chapter 6. It consists of two components; initialization code and event handler method.

```
$priorities[1][0] = 115$    ▷ Define priorities (area under the curve) with 2D array. First index represents a Queue ID and second index represents a range.
$priorities[1][1] = 35$
$priorities[2][0] = 230$
$priorities[2][1] = 69$    ▷ Add more if necessary.
for $i = 1$ to $QUEUECOUNT$ do    ▷ Initialization loop.
    $maxBW[i] = 10$    ▷ $maxBW$ contains the maximum bandwidth for queue $i$. At the beginning, max bandwidth for every queue is 10Mbps.
end for
$bwToRangeMap.add(10, 0)$    ▷ Mapping between max bandwidth and range ID in the priorities array.
$bwToRangeMap.add(20, 1)$
$bwToRangeMap.add(30, 2)$    ▷ Add more if necessary.
$availableBW = 100 - 10 * QUEUECOUNT$    ▷ Total available bandwidth for the port in Mbps. Every queue tries to get the share from this value.

function QueueStatsReplyHandler    ▷ This method is called every time a switch returns the queue statistics.
    for $i = 1$ to $QUEUECOUNT$ do
        $pPriorities[i] = 0$    ▷ Initialize PP of all queue to 0. In default, no queue wants more bandwidth.
        $newMaxBW[i] = maxBW[i]$    ▷ Save the current max. bandwidth to the temporary variable. Update $maxBW$ at the very end.
    end for
```

```
for each $q$ in $Queues$ do
    Calculate throughput of $q$.
    $rangeID = bwToRangeMap[maxBW[q.id]]$    ▷ Obtain current range ID of this queue.
    if $throughput < maxBW[q.id] - 12$ then    ▷ Current bandwidth is too much for this queue. It's okay to reduce it.
        $newMaxBW[q.id] = maxBW[q.id] - 10$
        $availableBW+ = 10$
        $cPriorities[q.id] = priorities[q.id][rangeID - 1]$    ▷ Use the reduced range ID to find the new current priority.
    else if $throughput > maxBW[q.id] - 2$ then ▷ This queue wants more bandwith.
        $cPriorities[q.id] = priorities[q.id][rangeID]$
        if $rangeID < MAXRANGE$ then
            $pPriorities[q.id] = priorities[q.id][rangeID + 1]$    ▷ Update PP from 0 to the next priority level of CP.
        else    ▷ Current bandwidth is just enough for this queue.
            $cPriorities[q.id] = priorities[q.id][rangeID]$
        end if
    end if
end for

Create the sorted array of Queue ID based on PP in descending order.
Create the sorted array of Queue ID based on CP in ascending order.

for $i = 0$ to $QUEUECOUNT - 1$ do    ▷ Allocate available bandwidth to those who want.
    if $availableBW <= 0$ then    ▷ If there is no more bandwidth in the free pool, move onto transfer section.
        Break
    end if
    $requestingQID = sortedPPIDs[i]$
    $associatedPP = pPriorities[requestingQID]$
    if $associatedPP > 0$ then
```

```
            $newMaxBW[requestingQID] = maxBW[requestingQID] + 10$
            $pPriorities[requestingQID] = 0$    ▷ Keeping the record that the increase request from $requestingQID$ has been fulfilled.
            $availableBW -= 10$
        end if
    end for

    for $i = 0$ to $QUEUECOUNT - 1$ do    ▷ Transfer bandwidth from the queue with low CP to the queue with high PP.
        $requestingQID = sortedPPIDs[i]$
        $requesterPP = pPriorities[requestingQID]$
        $targetQID = sortedCPIDs[i]$
        $targetCP = cPriorities[targetQID]$
        if $requesterPP > 0$ and $requesterPP > targetCP$ then ▷ Highest PP > Lowest CP so transfer must occur.
            $newMaxBW[requestingQID] = maxBW[requestingQID] + 10$
            $newMaxBW[targetQID] = maxBW[targetQID] - 10$
        end if
    end for

    for $i = 1$ to $QUEUECOUNT$ do
        if $maxBW[i]$ != $newMaxBW[i]$ then
            Set eqch queue's new max. bandwidth using the controller's API.
        end if
        $maxBW[i] = newMaxBW[i]$    ▷ Finally, update the $maxBW$ for the next check iteration.
    end for
end function
```

REFERENCES

[1] *Long-term verifiability of the electronic healthcare records authenticity*, International Journal of Medical Informatics, 76 (2007), pp. 442 – 448.

[2] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, *Cells: A virtual mobile smartphone architecture*, in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, New York, NY, USA, 2011, ACM, pp. 173–187.

[3] A. Banchs and X. Perez, *Distributed weighted fair queuing in 802.11 wireless lan*, in Communications, 2002. ICC 2002. IEEE International Conference on, vol. 5, 2002, pp. 3121–3127.

[4] M. Bari, S. Chowdhury, R. Ahmed, and R. Boutaba, *Policycop: An autonomic qos policy enforcement framework for software defined networks*, in Future Networks and Services (SDN4FNS), 2013 IEEE SDN for, Nov 2013, pp. 1–7.

[5] B. Blobel, *Authorisation and access control for electronic health record systems*, International Journal of Medical Informatics, 73 (2004), pp. 251 – 257. Realizing Security into the Electronic Health Record.

[6] Z. Cao, Z. Wang, and E. Zegura, *Rainbow fair queueing: fair bandwidth sharing without per-flow state*, in INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 2, 2000, pp. 922–931.

[7] T. Cgroups, *Cgroups - archwiki.* https://wiki.archlinux.org/index.php/Cgroups, 2015.

[8] J. Choe and S. K. Yoo, *Web-based secure access from multiple patient repositories*, International Journal of Medical Informatics, 77 (2008), pp. 242 – 248.

[9] D. Clark and W. Fang, *Explicit allocation of best-effort packet delivery service*, Networking, IEEE/ACM Transactions on, 6 (1998), pp. 362–373.

[10] P. Dely, A. Kassler, L. Chow, N. Bambos, N. Bayer, H. Einsiedler, C. Peylo, D. Mellado, and M. Sanchez, *A software-defined networking approach for handover management with real-time video in wlans*, Journal of Modern Transportation, 21 (2013), pp. 58–65.

[11] A. Demers, S. Keshav, and S. Shenker, *Analysis and simulation of a fair queueing algorithm*, in Symposium Proceedings on Communications Architectures &Amp; Protocols, SIGCOMM '89, New York, NY, USA, 1989, ACM, pp. 1–12.

[12] M. Devera, *Hierarchy token bucket home.* http://luxik.cdi.cz/~devik/qos/htb, 2013.

[13] J. L. Fernndez-Alemn, I. C. Seor, P. ngel Oliver Lozoya, and A. Toval, *Security and privacy in electronic health records: A systematic literature review*, Journal of Biomedical Informatics, 46 (2013), pp. 541 – 562.

[14] A. Ferreira, R. Cruz-Correia, L. Antunes, P. Farinha, E. Oliveira-Palhares, D. Chadwick, and A. Costa-Pereira, *How to break access control in a controlled manner*, in Computer-Based Medical Systems, 2006. CBMS 2006. 19th IEEE International Symposium on, 2006, pp. 847–854.

[15] J.-P. Georges, T. Divoux, and E. Rondeau, *Strict priority versus weighted fair queueing in switched ethernet networks for time critical applications*, in Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, April 2005, pp. 141–141.

[16] P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race, *Towards network-wide qoe fairness using openflow-assisted adaptive video streaming*, in Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking, FhMN '13, New York, NY, USA, 2013, ACM, pp. 15–20.

[17] S. Graber, *Lxc 1.0: Unprivileged containers [7/10]*. https://www.stgraber.org/2014/01/17/lxc-1-0-unprivileged-containers/, 2014.

[18] T. C. Group, *Trusted computing group - developers - trusted platform module*. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[19] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, *Remote attestation on program execution*, in Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, STC '08, New York, NY, USA, 2008, ACM, pp. 11–20.

[20] J. Guo, F. Liu, H. Tang, Y. Lian, H. Jin, and J. C. Lui, *Falloc: Fair network bandwidth allocation in iaas datacenters via a bargaining game approach.*, in ICNP, 2013, pp. 1–10.

[21] V. Haldar, D. Chandra, and M. Franz, *Semantic remote attestation: a virtual machine directed approach to trusted computing*, in USENIX Virtual Machine Research and Technology Symposium, vol. 2004, 2004.

[22] C. Humer and J. Finkle, *Your medical record is worth more to hackers than your credit card*. http://www.reuters.com/article/2014/09/24/us-cybersecurity-hospitals-idUSKCN0HJ21I20140924, September 2014.

[23] IEEE, *802.11i-2004 - 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: Amendment 6: Medium access control (mac) security enhancements*. http://standards.ieee.org/findstds/standard/802.11i-2004.html, 2004.

[24] ——, *802.1x-2010 - ieee standard for local and metropolitan area networks–port-based network access control*. http://standards.ieee.org/findstds/standard/802.1X-2010.html, 2010.

[25] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelem, *Control of multiple packet schedulers for improving qos on openflow/sdn networking*, in Software Defined Networks (EWSDN), 2013 Second European Workshop on, Oct 2013, pp. 81–86.

[26] J. Kidambi, D. Ghosal, and B. Mukherjee, in Journal of High Speed Networks.

[27] J. Lee, M. Uddin, J. Tourrilhes, S. Sen, S. Banerjee, M. Arndt, K.-H. Kim, and T. Nadeem, *mesdn: Mobile extension of sdn*, (2014).

[28] LXC, *Lxc-linux containers: Userspace tools for the linux kernel containers.* https://linuxcontainers.org/.

[29] R. Mahajan, S. Floyd, and D. Wetherall, *Controlling high-bandwidth flows at the congested router*, in Network Protocols, 2001. Ninth International Conference on, Nov 2001, pp. 192–201.

[30] S. Marshall, *It consumerization: A case study of byod in a healthcare setting*, Technology Innovation Management Review, 4 (2014), pp. 14–18.

[31] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, *Cloud terminal: Secure access to sensitive applications from untrusted systems.*, in USENIX Annual Technical Conference, 2012, pp. 165–182.

[32] D. Matic, *Xnest(1) manual page.* http://www.xfree86.org/4.2.0/Xnest.1.html.

[33] D. Merkel, *Docker: Lightweight linux containers for consistent development and deployment*, Linux J., 2014 (2014).

[34] J. E. Moyer, *Managing mobile devices in hospitals: A literature review of byod policies and usage*, Journal of Hospital Librarianship, 13 (2013), pp. 197–208.

[35] OpenMRS, *Openmrs.* http://openmrs.org/.

[36] T. OpenNetworkingFoundation, *Openflow - open networking foundation.* https://www.opennetworking.org/sdn-resources/openflow, 2015.

[37] T. OpenvSwitch, *Open vswitch.* http://openvswitch.org/, 2015.

[38] R. Pan, B. Prabhakar, and K. Psounis, *Choke - a stateless active queue management scheme for approximating fair bandwidth allocation*, in INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 2, 2000, pp. 942–951.

[39] A. K. Parekh and R. G. Gallager, *A generalized processor sharing approach to flow control in integrated services networks: The single-node case*, IEEE/ACM Trans. Netw., 1 (1993), pp. 344–357.

[40] QubesOS, *Qubesos.* https://wiki.qubes-os.org/.

[41] Shibboleth, *Shibboleth consortium.* https://shibboleth.net/about/.

[42] P. Skoldstrom and W. John, *Implementation and evaluation of a carrier-grade openflow virtualization scheme*, in Software Defined Networks (EWSDN), 2013 Second European Workshop on, Oct 2013, pp. 75–80.

[43] T. TrouSerS, *Trousers - the open-source tcg software stack.* http://trousers.sourceforge.net/.

[44] T. Upstart, *upstart - event based init daemon.* http://upstart.ubuntu.com, 2010.

[45] USDHHS, *Health information privacy.* http://www.hhs.gov/ocr/privacy/index.html.

[46] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, *An integrated experimental environment for distributed systems and networks*, Boston, MA, Dec. 2002, pp. 255–270.

[47] K.-K. Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar, *Making use of all the networks around us: A case study in android*, in Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design, CellNet '12, New York, NY, USA, 2012, ACM, pp. 19–24.