DEIDTECT - DISTRIBUTED ELASTIC INTRUSION DETECTION ARCHITECTURE

by

Praveen Kumar Shanmugam

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

May 2016

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Praveen Kumar Shanmugam**

has been approved by the following supervisory committee members:

Jacobus Van Der Merwe , Chair **9/15/2015**
Date Approved

Sneha Kumar Kasera , Member **9/15/2015**
Date Approved

Joseph R Breen III , Member **9/15/2015**
Date Approved

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

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

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

ABSTRACT

Current Intrusion Detection Systems (IDS) in a typical enterprise or campus network are limited by having a number of static monitoring points and static IDS resources deployed. The monitoring points are typically deployed using hardware optical taps or span ports which are directly fed into the IDS. The IDS system is a compute resource requiring dedicated-server-grade hardware, and these are statically configured when installing the network for an enterprise or campus.

We designed a framework for making a distributed elastic Intrusion Detection System (IDS) for a Software Defined Network (SDN) capable network, called *Distributed Elastic Intrusion DeTECTion* (DEIDtect). We combine the flexibility of SDN and the elastic resource usage of a cloud infrastructure with a DEIDtect orchestrating controller to achieve an elastic IDS framework. DEIDtect enables simple and more dynamic management of IDS systems. The flexibility of our approach also enables new IDS use cases and deployment strategies.

For my Mom, Dad, Sister, and Friends

CONTENTS

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost I thank my Adviser Kobus for guiding me from day one of my Master's program in the right direction. Thanks to Joe for co-advising me all the way.

Thanks to Corey, Sneha, Rob, and David for their valuable feedback and guidance in achieving the milestone.

I thank all my friends in Flux for helping me out in any technical questions towards my thesis.

And I thank all my friends who have played a big part in my grad school life.

This material is based upon work supported by the National Science Foundation under Grant No. 58501934 and 58501880.

CHAPTER 1

INTRODUCTION

Intrusion detection and prevention systems (IDS/IPS) are widely deployed as critical tools in the toolkit of security professionals. For example, among the open source IDS/IPS systems, Snort boasts millions of downloads and approximately 400,000 registered users, while it is estimated that as many as 10,000 organizations make use of Bro [1, 25]. Despite its widespread use, current IDS/IPS deployments are plagued by a number of practical concerns that limit their utility. First, the compute and network resources required to effectively run an IDS/IPS often present problematic cost versus functionality tradeoffs: Compute requirements for an IDS/IPS system vary over time, depending on the volume of traffic and the type of analysis that security personnel are performing. For example, a developing security event might require more detailed deep packet inspection, which would demand running an IDS/IPS instance configured for this purpose. In practice this results in two undesirable options: Either compute resources are deployed to accommodate anticipated peak requirements, leading to over provisioning during off-peak times; or more typically, compute resources are knowingly underprovisioned. Underprovisioning results in a loss of visibility during peak times, which, in instances like a Distributed Denial of Service (DDoS) attack, might be when intelligence is most needed.

Network requirements for IDS/IPS deployment involve a tap point in the network infrastructure and sufficient capacity from the tap point to the compute resources hosting the IDS/IPS. A network tap is typically realized using an optical splitter or, in smaller deployments, a switch span/monitoring port. Both approaches are highly inflexible: Once deployed, monitoring is constrained to the chosen tap location, which is typically deployed at the ingress/egress point of a campus or enterprise network. The implication is that intrusions that remain within the enterprise network might go undetected.

A more fundamental concern is that current IDS/IPS deployments are typically strictly local concerns. (This remains true in practice despite various earlier efforts towards distributed intrusion detection systems [14, 17, 29].) Specifically, while security professionals at different organizations readily exchange intelligence through personal communication, there is no systematic way to follow a lead to a remote location to investigate the potential source of an attack. Furthermore, it is typically not possible to utilize remote expertise or resources to investigate a local problem.

A final concern is the fact that managing an IDS/IPS system is quite complex. For example, setting up a small scale Snort/Bro instance is a well-documented activity, typically well within reach for a competent system administrator. However, performing the same activity to scale to campus or enterprise environments quickly becomes a significant engineering challenge [20]. Furthermore, systems like Bro provide more flexibility, customization, and analysis capabilities. However, it is significantly more involved to set up and requires ongoing management by domain experts.

We argue that the inflexibility associated with the compute and network resources needed for IDS/IPS is the root cause for these concerns and prevents rich cross-domain security models and investigation.

Combining the flexibility of SDN and the elastic resource usage of a cloud infrastructure with a DEIDtect orchestrating controller, we achieve an elastic IDS framework.

The initial design work of DEIDtect was presented at the ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC) 2014 and titled *"DEIDtect: Towards Distributed Elastic Intrusion Detection"* [27].

1.1 Thesis Statement

Our thesis is that Software Defined Network (SDN) and Cloud Technology will be used to create a framework which has flexible network monitoring and elastic Intrusion Detection System (IDS) deployment with cross-domain capability, enabling new IDS use cases.

1.2 Thesis Contributions

- We present the DEIDtect architecture which provides a distributed elastic framework for cross-site security functions.

- We present a detailed design of the networking component of DEIDtect which involves: (i) SDN primitives for safely tapping arbitrary traffic at arbitrary locations in an enterprise network, (ii) cloud abstractions for the precise distribution of traffic in a cloud environment, (iii) a number of interdomain SDN interactions, including security specific intersite communication.

- We present the implementation of our design and evaluate it in an emulated multiple-enterprise SDN site environment and a real Cloud environment.

- We present the tap traffic management feature of DEIDtect framework, which rate-limits tap traffic dynamically based upon the service traffic rate and, also whitelists tap traffic based upon the IDS Virtual Machine Feedback. It also scales the IDS systems in the cloud based upon the IDS Virtual Machine (VM) processor usage. This is called Adaptive Load Balancing (ALB).

- We evaluate ALB in both local and distributed deployment of DEIDtect to show how (1) the tap traffic is controlled at different service traffic rates, (2) how Whitelisting cuts down the drop rate of the tunnel traffic, and (3) the IDS will be scaled up based upon the IDS Virtual Machine (VM) processor usage.

1.3 Thesis Overview

The rest of the thesis is organized as follows. We describe the DEIDtect architecture and design in Chapter 2. Chapter 3 details the use cases of DEIDtect. Chapter 4 covers all the work related to DEIDtect. The implementation of the DEIDtect is detailed in Chapter 5, followed by the evaluation in Chapter 6. In Chapter 7 we talk about the challenges faced in building the DEIDtect framework prototype, along with the future work. Chapter 8 concludes this thesis.

CHAPTER 2

DEIDtect ARCHITECTURE AND DESIGN

2.1 Overview

In this chapter, we present the DEIDtect architecture and design methodology. The DEIDtect framework addresses the fundamental security deployment concerns by exploiting software-defined networking and cloud computing. The DEIDtect approach effectively enables decoupling of the location of the network being monitored/protected and the location of the security tools performing security functions. The DEIDtect approach also enables rapid scaling of security resources during a security event. This flexibility provides the opportunity to explore new distributed network security functionality and potentially enable better visibility and coordination among partnered organizations.

2.1.1 DEIDtect Dynamic and Comprehensive Network Tapping

DEIDtect exploits SDN functionality to allow the network and system administrators to tap at any point in the network, and feeds the tapped traffic stream to the security collectors and analyzers. Though limited to the bandwidth of the aggregate links, administrators can implement tap points rapidly at arbitrary points in the network. With a fully instrumented network, administrators can deploy taps anywhere at the access edge, distribution, core, or internet border. With a partially instrumented network, administrators can leverage SDN taps in a flexible manner and optical taps and SPAN/mirror ports at traditional key areas. DEIDtect also allows fine-grained tapping of specific traffic.

2.1.2 DEIDtect Elastic Security Compute Platform

The DEIDtect approach allows security and network administrators to leverage cloud resources to create virtual or ‘bare metal’ images of security tools. DEIDtect adds the capability to the normal cloud orchestration architecture to create the virtual image and to create a network path from the border of the cloud to the specific host with the security image. By leveraging this technique, DEIDtect can balance tapped traffic flows across multiple security images quickly in order to scale.

2.1.3 DEIDtect Distributed Network Security Functions

DEIDtect's flexible resource usage enables distributed network security functions. One possible scenario could involve a cloud vendor or a large entity, such as a university, company, or government with a large private internal cloud, offering elastic cloud-based security functions to internal or external customers. Assuming a fully DEIDtect-enabled network and cloud, i.e., DEIDtect controlled SDN in both enterprise and cloud, internal customers can be readily served by dynamically exposing selected tap points to cloud-based security tools. Serving external customers will be possible by similarly deploying DEIDtect technology in the customer network and utilizing wide-area SDN infrastructure (or at least semistatic predefined circuits) between the customer network and the cloud location.

Another scenario might involve a university or company with a number of remote sites, i.e., small campus sites, field stations or clinics. With the current set of security tools, security administrators rarely have the ability to deploy tap infrastructure with a dedicated feed into the central campus security tool suite. With a DEIDtect-enabled network and cloud, this scenario becomes feasible.

A corollary scenario might involve a university or entity that has "sister" sites or smaller campuses with a tight business, academic, research, or healthcare association. These discrete sites may wish to leverage common computational resources, security tools, or expertise. This arrangement would allow for greater visibility into emerging security concerns, thus providing a foundation for detecting more subtle attacks.

2.2 DEIDtect Architecture

DEIDtect exploits two current trends, namely, the increased use of cloud computing technologies to consolidate compute resources and the increasing deployment of software-defined networking (SDN) technology in enterprise, cloud, and wide area networks. DEIDtect uses cloud computing resources to flexibly and efficiently deal with the computing needs of security tools, like IDS and IPS, while SDN is utilized to flexibly and safely tap and distribute network traffic between monitored networks and IDS/IPS instances.

A high-level view of DEIDtect architecture is depicted in Figure 2.1. The figure shows three campus or enterprise networks, i.e., Sites A, B, and C, of which sites A and C also have their own cloud computing platforms. The different sites are interconnected via a wide-area network (WAN). For ease of exposition, we assume that all networks in question are SDN enabled, although hybrid deployments, e.g., with static or dynamic WAN circuits, would certainly be feasible. We note that the sites could be distributed locations of the same institution, such as a university, or they could be associated with different organizations that have a collaboration or business arrangement to work together on security functions.

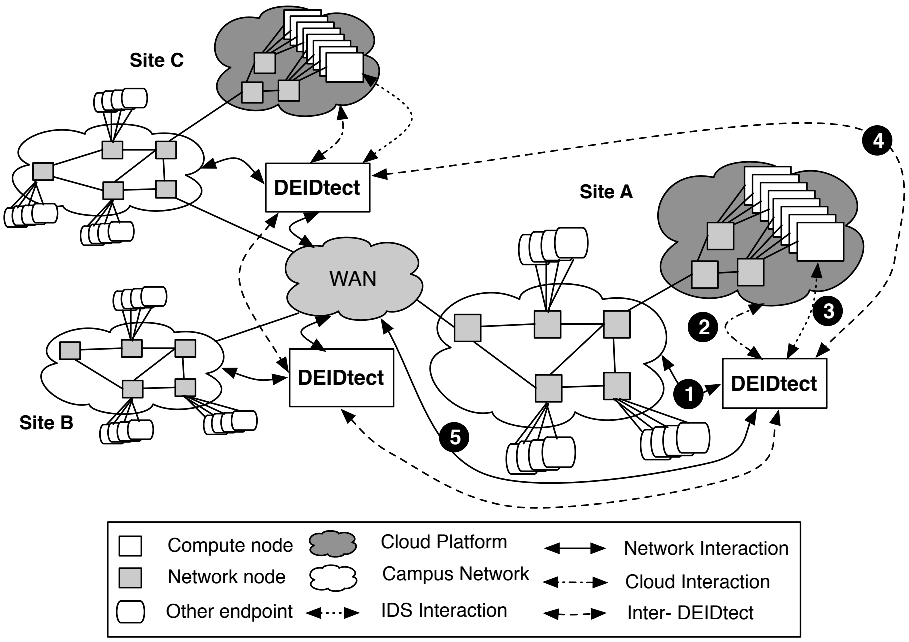


Figure 2.1: DEIDtect Architecture

Given this underlying physical infrastructure, DEIDtect architecture involves DEIDtect systems deployed at each site. As shown in Figure 2.1, the DEIDtect system at each site is involved with five types of interactions. DEIDtect interacts with: (1) The campus or enterprise network to realize network taps and to transfer tapped traffic towards the IDS/IPS systems in the cloud infrastructure. (2) The cloud computing platform to realize IDS/IPS instances and to manipulate the distribution of tapped traffic towards these instances. (3) The IDS/IPS instances in the cloud to control their intrusion detection and prevention functionality. (4) Remote DEIDtect systems to request and manipulate cloud, network, and IDS/IPS resources at remote sites. (5) The wide-area network to realize intersite connectivity.

2.2.1 DEIDtect System

A system level view of DEIDtect is depicted in Figure 2.2. At the center of the system is the DEIDtect Core Module, which interacts with and orchestrates actions across other system components. Specifically, as shown in Figure 2.1 and described earlier, the Core Module interacts with five other components in the system: (1) the Enterprise SDN Network to create tap points in the network and to deliver monitored traffic to the cloud, (2) the Cloud Computing Platform to instantiate cloud-based IDS/IPS instances and to route traffic from the network to the appropriate IDS/IPS instance, (3) the instantiated IDS/IPS instances to orchestrate intrusion detection and prevention, (4) remote DEIDtect systems to enable distributed security functions, and (5) the WAN SDN network to allow delivery of

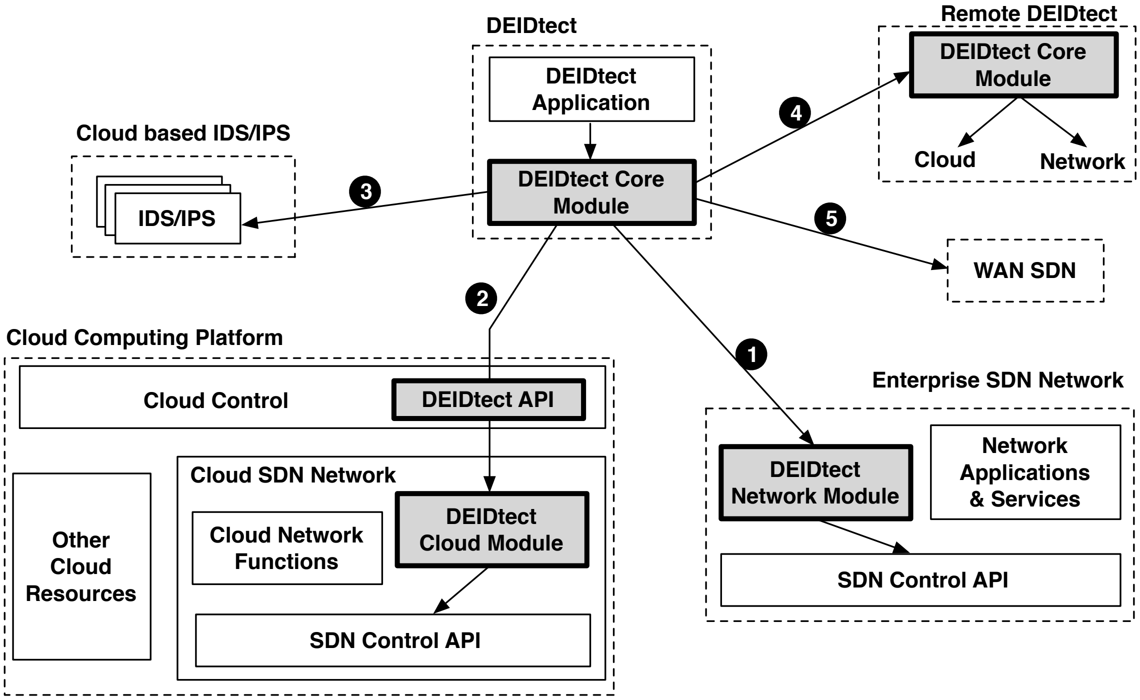


Figure 2.2: DEIDtect System

tapped network traffic between distributed locations.

Figure 2.2 also shows how DEIDtect components (shaded boxes) integrate with existing systems, specifically the cloud computing platform and the enterprise SDN network. As shown in the figure, a DEIDtect Network System module associated with the enterprise SDN network allows DEIDtect to tap the enterprise network. Similarly, the DEIDtect Cloud System module allows DEIDtect to distribute tapped traffic to appropriate IDS/IPS instances in the cloud.

2.3 DEIDtect Network System Module

We assume that the enterprise network in question is SDN enabled and specifically supports OpenFlow version 1.1 (or higher). Figure 2.3 shows a more detailed view of the submodules, which is composed in DEIDtect Network System module. It consists of (a) SafeTap module to safely tap arbitrary network traffic at arbitrary points in the network, (b) Bandwidth Monitor module to calculate the real time usage of the tap ports, (c) White Listing module to install drop rules based upon the feedback from the IDS and (d) Rate Limiter module to increase/decrease the rate of tap traffic being sent from the port based upon the current rate of service traffic at that port. This will be discussed in more detail in the coming sections.

A key DEIDtect requirement is to be able to safely tap arbitrary network traffic at arbitrary points

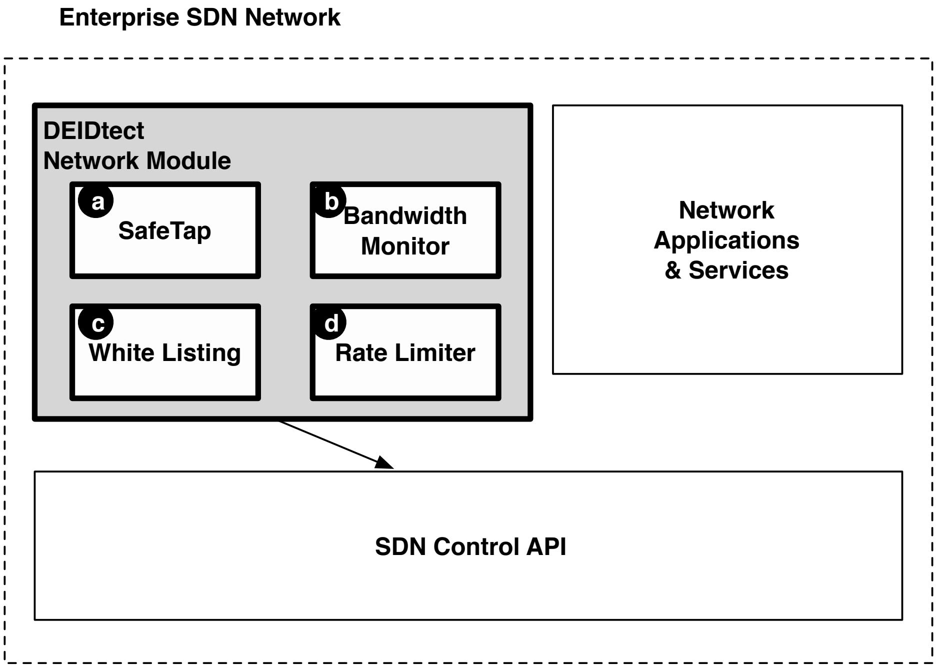
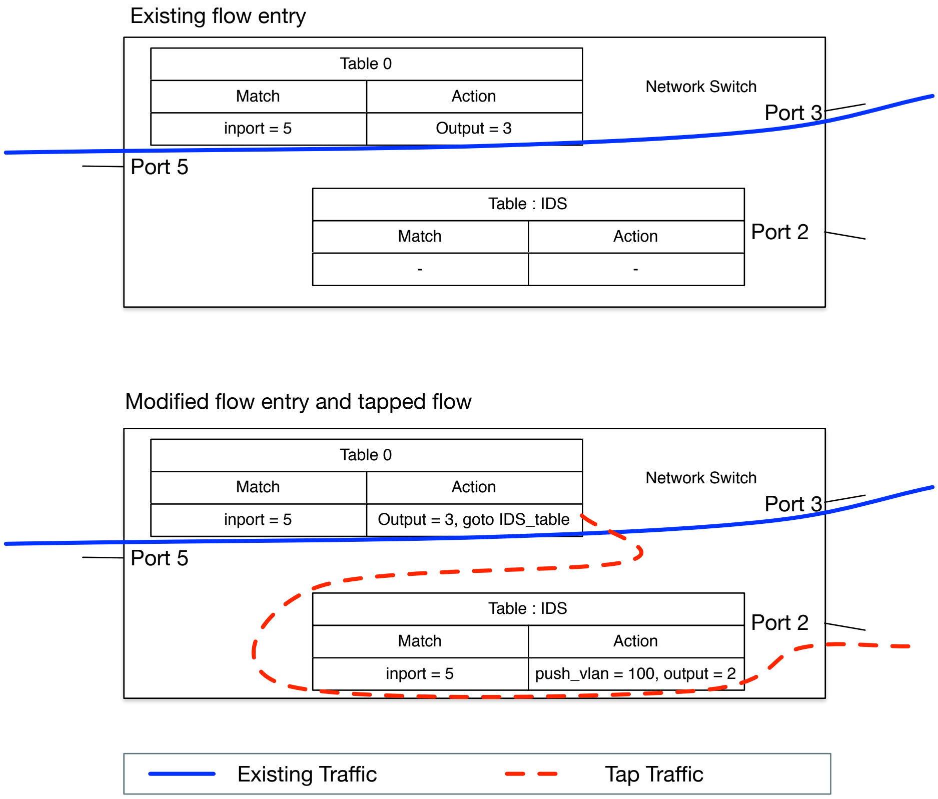


Figure 2.3: DEIDtect Enterprise Network Modules

in the network without impacting existing traffic flows. Further, tapped traffic needs to be transparently transported to the cloud platform for processing by the IDS/IPS. Transparent transportation of tapped traffic in DEIDtect is achieved by adding a tunneling tag (e.g., a VLAN tag) to tapped traffic at the tap switch, and by routing the tagged traffic to the cloud platform. DEIDtect takes advantage of the multitable functionality available in OpenFlow version 1.1 (or higher) to achieve safe tapping. Specifically, for flows to be tapped, the normal (existing) flow entry is augmented so that in addition to the existing flow actions, the flows are also routed to an IDS-specific flow table, which adds the tunneling tag and forwards the packet towards the cloud platform.

Figure 2.4 shows a single-switch tapping example. The top part of the figure shows an existing flow entry that forwards packets received on port 5 out on port 3. The bottom part of the figure shows the modified existing flow entry which continues to output packets on port 3, but also copies the packet for processing to the IDS table *goto IDS_table*. The IDS table entry, in turn, adds a VLAN tag and sends the packet out on port 2 to complete the tapping action.

Algorithm 1 shows the pseudo-code associated with flow table manipulation to realize tapping. The *egressSafeTap* function applies the safe-tap flow modification for the specified switch and out port to be monitored. The algorithm goes through all the tables in the switch and finds all the flows which have the specified out port in its action field. It then adds the *goto IDS table* instruction in the flow to create a copy of packets, and creates a flow in the *IDS table* with the same match and action instructions, along with VLAN add instruction, and it pushes the packet out via the computed



Existing flow entry

Table 0	
Match	Action
inport = 5	Output = 3

Table : IDS	
Match	Action
-	-

Modified flow entry and tapped flow

Table 0	
Match	Action
inport = 5	Output = 3, goto IDS_table

Table : IDS	
Match	Action
inport = 5	push_vlan = 100, output = 2

Figure 2.4: DEIDtect: Flow Tables Modification

path to send towards the IDS in the cloud. The flows created in the IDS table are associated with a rate-limiting meter entry created for the specified port for rate limiting the tap traffic. The intermediate switches in the computed path are installed with pass-through flows for the newly created VLAN, and packets are forwarded towards the IDS in the cloud. This basic approach can be readily extended to allow more specific traffic to be delivered to the IDS by specifying more refined match instructions in the IDS table flows.

2.3.1 Adaptive Load Balancing (ALB)

The installation of a 'tap point' is referred to as a monitoring instance. The monitoring instance is comprised of the Enterprise network flows installed in each of the switches and the Cloud instance associated with the 'tap point'. The DEIDtect safe tapping is done using the same network as the network services of the Enterprise network, following a shared network approach to send the 'tap traffic' to the IDS instance. In order to handle the rate of tap traffic being delivered, DEIDtect requires the following complementary approaches: (1) decrease/increase the traffic sent from network to the IDS (Adaptive Rate Limiting), (2) increase the number of IDS instances to handle the increased traffic load (Adaptive IDS Scaling).

Adaptive Load Balancing (ALB) is comprised of the following mechanisms. (1) ALB rate-

Algorithm 1 Tap flow table manipulation

```
procedure EgressSafeTap($tapTable, srcSwitch, srcPort, dstSwitch, flowLabel$)
    $path \leftarrow findPath(srcSwitch, dstSwitch)$
    while $node\ in\ path$ do
        if $node == srcSwitch$ then
            $flowEntries \leftarrow srcSwitch[Table = ALL]$
            for $i, 1 \rightarrow n$ do
                if $flowEntries[i].action[output] == srcPort$ then
                    $meterID \leftarrow$ create_mete(drop_band)          ▷ rate-limiting TAP flow
                    add_flow(goto_table(,tapTable)
                    add_flow(match:flowEntries[i].match,meter:meterID,
                    action:(push_label(flowLabel),output:($node.nextPort$)))
                end if
            end for
        else if $node == dstSwitch$ then
            add_flow(match:flowLabel,action:(pop_label(flowLabel),
            output:($node.nextPort$))
        else
            add_flow(match:flowLabel,output:($node.nextPort$)
        end if
    end while
end procedure
```

limiting to increase or decrease the tap traffic rate as per the service traffic rate, (2) ALB-Whitelisting creates a drop rule for a flow in the tap traffic which is classified as normal traffic by the IDS, (3) ALB-IDS scaling which creates more IDS VM to handle more traffic for analysis.

The ALB-Rate-limiting and Whitelisting are explained using Figure 2.5. As shown in Figure 2.5 (a), there are many monitoring instances installed, highlighted in different colors. Higher number of monitoring instances cause the links to get congested quickly, causing the Enterprise service disruptions labeled as ”Congestion” in the figure which is depicted as thick lines to show higher tap-traffic rate. DEIDtect ALB-Rate-limiting runs in a periodic way to adjust the tap traffic rate and ALB-Whitelisting controls the congestion of the IDS traffic from the IDS feedback. The IDS executing in the Cloud platform (1) sends feedback to DEIDtect with the Whitelisting traffic and the current load on the IDS instance. This is processed by the DEIDtect controller which (2) installs the appropriate drop rules in the enterprise network. Figure 2.5 (b) illustrates the resulting reduction in bandwidth usage of the Enterprise network after ALB takes place, which is labeled as ”Congestion Free”, depicted as thin lines to show reduced tap-traffic rate.

2.3.1.1 ALB - Adaptive Rate Limiting

The DEIDtect has the flexibility to install many *monitoring instances* in the Enterprise network. This can consume the network bandwidth at a higher rate, causing congestion. The monitoring

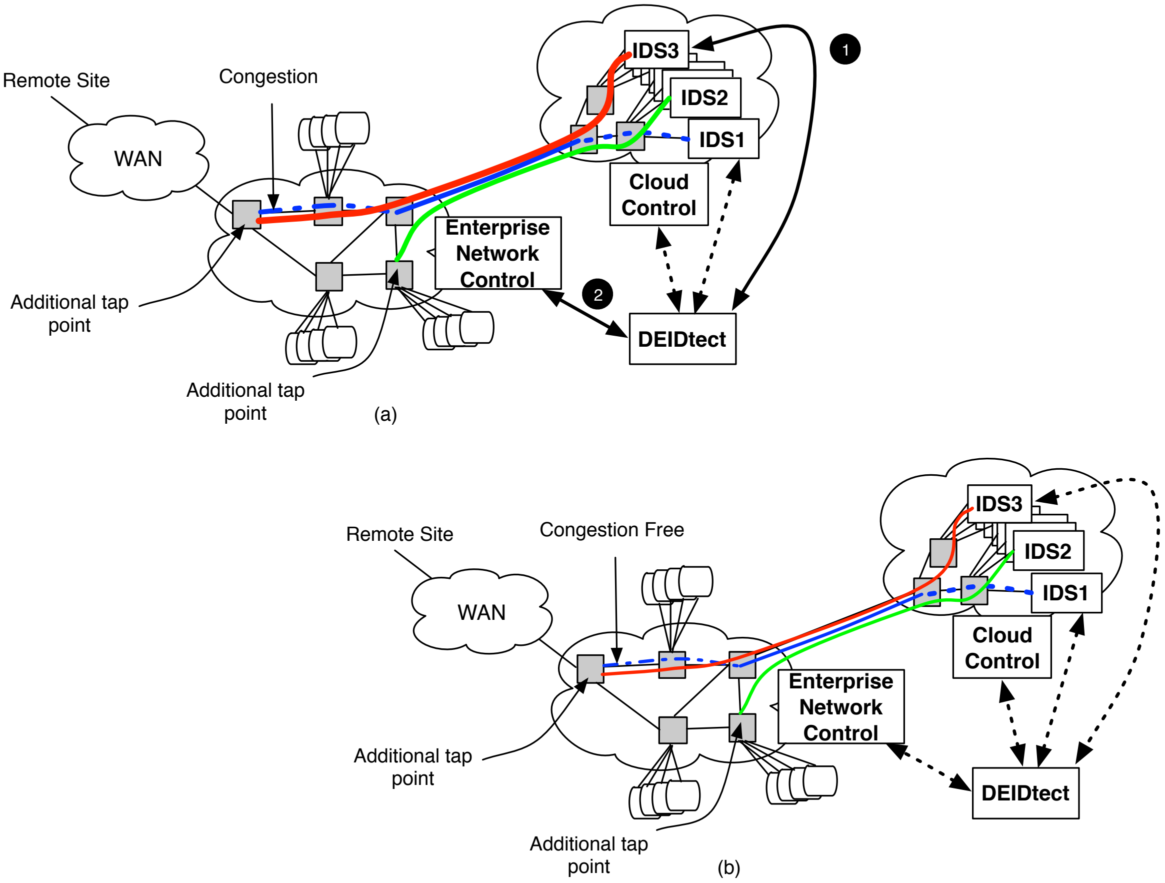


Figure 2.5: DEIDtect: Adaptive Load Balancing

instance allocates only a portion of the network bandwidth using the OpenFlow Metering feature, and makes sure that there is no impact on existing service traffic. The monitoring instance traffic rate is set as per the current traffic rate in the 'tap point', subtracting the monitoring instance metering value, and the new value is calculated based upon Algorithm 2.

2.3.1.2 Additive Increase and Multiplicative Decrease - Link Sharing

The algorithm monitors the bandwidth usage at each of the monitoring ports, and if the usage is between *threshold_min* and *threshold_max* percentage, the meter table's rate limit is reduced in a multiplicative way, and if it is less than *threshold_min*, the rate limit of the meter table is increased in an additive way. This allows the service traffic to be forwarded without impact. The additive increase and multiplicative decrease approach was inspired by the TCP congestion control Algorithm [9]. We have used this approach to have fairness in the link sharing between service and tap traffic. And also we want to be cautious about increasing the tap traffic rate, which can disrupt the service traffic. Hence we have followed the *Additive Increase and Multiplicative Decrease* model.

Algorithm 2 Adaptive Traffic Reduction - Rate Limiting

```
procedure RATELIMIT($switch, meterID, bandwidth, usedBW, percentageInc$)
    $maxRateLimit \leftarrow bandwidth/2$                                ▷ 50% of the bandwidth
    $rateInc \leftarrow bandwidth * percentageInc$
    $rateLimit \leftarrow$ OF.getMeterConfig(switch, meterID)      ▷ Openflow query to get meter drop limit
    if $threshold\_min \leq usedBW \geq threshold\_max$ then
        $newLimit \leftarrow rateLimit/2$                               ▷ Multiplicative Decrease
    else
        if $rateLimit = maxRateLimit$ then
            return                                                        ▷ Do nothing
        end if
        $newLimit \leftarrow rateLimit + rateInc$                        ▷ Additive Increase
        if $newLimit \geq maxRateLimit$ then
            $newLimit \leftarrow maxRateLimit$
        end if
    end if
    OF.meterMod(switch, meterID, newLimit)
end procedure
```

2.3.1.3 ALB - Whitelisting

This metering table rate-limiting feature drops the packet as per the switch implementation and hence we do not have any control over how the packet is dropped. But DEIDtect can reduce tap traffic volume in a smarter way. The 'tap traffic' for each monitoring instance is comprised of both normal and abnormal traffic. Since we have the IDS, it is possible to identify the normal traffic. The normal traffic identification is supported by most of the IDSes, and this is used by DEIDtect to accomplish the way the traffic is dropped in the Enterprise network for the monitoring instance. The IDS is configured to send drop request to DEIDtect to drop selected normal traffic at the 'tap point' which ensures that the packets which go past the rate-limiter are part of the abnormal traffic rather than the normal/whitelist traffic. This is referred to as traffic Whitelisting. The pseudo-code to facilitate the traffic throttling via Whitelisting is given in Algorithm 3.

The algorithm installs a drop rule with the traffic type in the safe tapping IDS table, as in Figure 2.4, which results in unwanted traffic being dropped before the metering could be applied. Dropping packets in this manner at the tap point ensures that only nonwhite-listed traffic is being sent to the IDS. These Whitelisting drop rules are set with a timeout to resume normal operation again, and

Algorithm 3 Adaptive Traffic Reduction - Whitelisting

```
procedure WHITELISTTRAFFIC($switch, whitelist\_type$)
    InstallDropRule(switch,table=tapTable,
    match=(whitelist_type), timeout = safeTapTimeout) ▷ configurable timeout
end procedure
```

if the IDS picks up the whitelist traffic again, the drop rules are installed. This timeout is to make sure that the drop rule does not cause the traffic type to forever remain whitelisted.

2.4 DEIDtect Cloud System Module

We assume that the enterprise network in question is SDN enabled. The DEIDtect Cloud System module has the following functionality: (1) to create an IDS instance for a monitoring instance, (2) to deliver the traffic from the cloud gateway to the associated IDS instance, and (3) to increase/decrease IDS instances associated with monitoring instance. Figure 2.6 shows the module level details of DEIDtect Cloud System: (a) DEIDtect Cloud Controller module to create the IDS instance, (b) DEIDtect Cloud Network module to create tunnel path for the tap traffic, and (c) DEIDtect ALB IDS scaling module to scale up/down the number of IDS instances based upon the VM processor usage.

DEIDtect Cloud System module creates an IDS VM with Whitelisting and cpu usage reporting configuration using the DEIDtect Cloud Controller module. DEIDtect Cloud Network module sets up the tap traffic delivery by finding the topology of the cloud network and finding the shortest path between the cloud gateway and the IDS instance, and installing the required flows. The flow installation procedure checks for existing flows and splits the traffic at the lowest possible subtree. This also takes into consideration that the Virtual Machine's (VM) existing traffic must not be disrupted. DEIDtect ALB-IDS scaling monitors the IDS usage and scales up/down the number of IDS instances to handle the traffic for the associated monitoring instance. The Whitelisting request from the IDS are

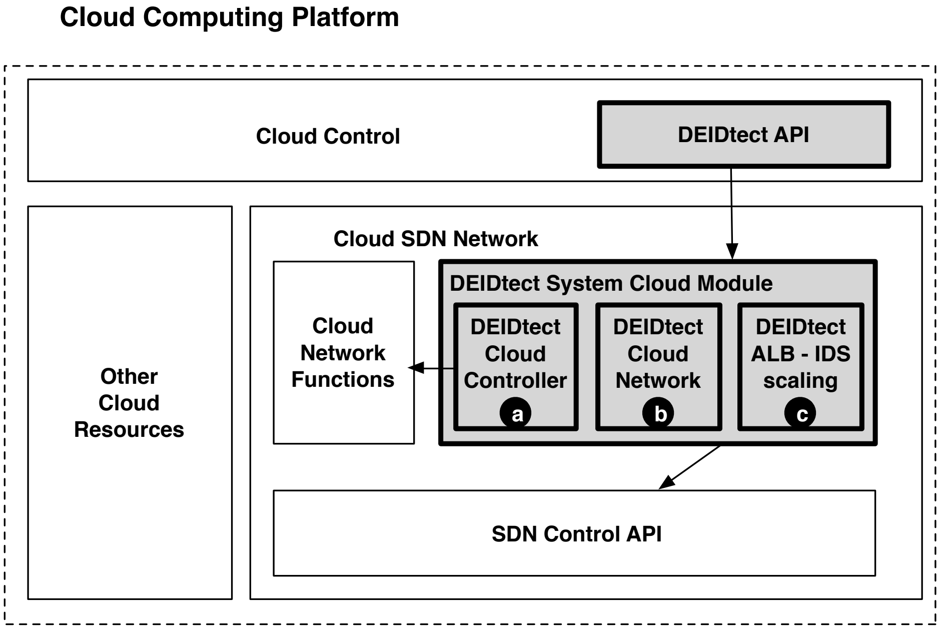


Figure 2.6: DEIDtect Cloud System Modules

sent to the DEIDtect Core module and that delegates it to the associated DEIDtect Network module in local/remote site based upon the monitoring instance information.

As described in Section 2.2, the DEIDtect Cloud System module forms part of and extends the functionality of an SDN-capable cloud computing platform. We abstracted DEIDtect Cloud Network System module functionality into a higher level API that a cloud control architecture would expose to allow DEIDtect to orchestrate cloud network functionality:

1. *createTrafficRoute(tunnel_id, traffic_type, VM_instance)* tells the cloud controller to create an isolated tunnel flow between the cloud gateway switch and the VM instance. Traffic tagged with tunnel_id in the cloud gateway, with the particular traffic_type delivered to the VM_instance.

2. *removeTrafficRoute(tunnel_id, VM_instance)* removes all the flow entries which were installed for *createTrafficRoute* API for the particular VM_instance identified by the tunnel_id.

3. *removeAllRoutes(VM_instance)* removes all the flow entries associated with that VM_instance. In other words, removes all flows installed for *createTrafficRoute* API calls, which are associated with VM_instance.

The above given higher level DEIDtect cloud API maps to the lower level API of the Cloud System module, which is part of the cloud SDN network as shown in Figure 2.2. The lower level API exposes the following functions:

1. *tapTunnelEntry(src_switch_id, dst_switch_id, dst_port, vlan_id)*: This creates a tunnel from the source switch to the destination switch with the given vlan_id. The dst_port specifies the port which is connected to the VM with respect to the cloud environment, and at the destination switch the vlan_id tag is removed and the traffic is delivered as it is seen by the actual destination.

2. *tapTunnelDelEntry(src_switch_id, dst_switch_id, dst_port, vlan_id)*: This removes the tunnel created by *tapTunnelEntry*.

3. *splitTunnelEntry(src_switch_id, dst_switch_id, dst_port, vlan_id, traffic_type)*: This splits the traffic tagged with vlan_id from the source switch and pushes the specified type of traffic_type to the destination switch via the dst_port.

4. *splitTunnelDelEntry(src_switch_id, dst_switch_id, dst_port, vlan_id, traffic_type)*: This removes the changes done by *splitTunnelEntry*.

2.4.1 ALB - Adaptive IDS Scaling

The traffic rate-limiting and Whitelisting are two of the actions taken by DEIDtect's ALB. Another action is the AutoScaling of IDS instances. This is used in scenarios where the bandwidth availability in the Enterprise Network is sufficient for a monitoring instance but the IDS instance associated with it gets overloaded by processing all the tapped traffic. The monitoring instance requires more computing resources to handle the traffic. DEIDtect have the capability to provision such needs of adding more worker nodes to the IDS cluster architecture as a part of this work. The pseudo-code to facilitate the auto scale is given in Algorithm 4.

The algorithm increases the number of worker nodes along with the network topology for a cluster IDS setup if the VM processor utilization is higher than *cpu_threshold*.

Algorithm 4 ALB - Dynamic Auto Scaling

```
procedure AutoScaling($TAPID, currentUtilization$)
    if $currentUtilization \geq cpu\_threshold$ then
        $nodes \leftarrow getCloudInstance(TAPID).nodeCount()$
        $workerInfo \leftarrow updateCloudInstance(TAPID, workers = nodes + 1)$
        updateMonitorInstanceInfo(workInfo)
    end if
end procedure
```

CHAPTER 3

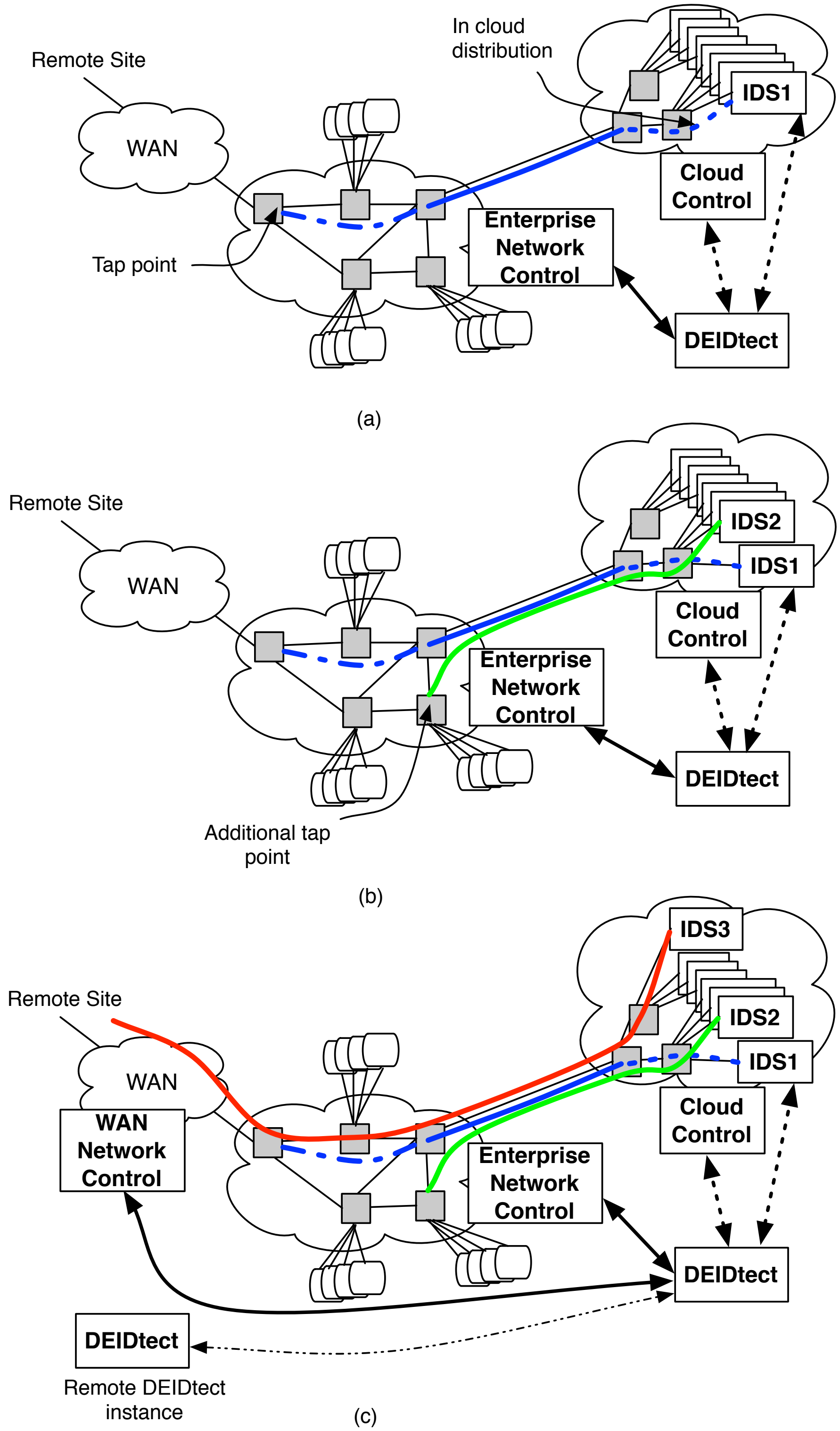
DEIDtect USE CASES

Different scenarios enabled by DEIDtect are depicted in Figure 3.1. Figure 3.1 (a) shows the default case that mimics current common practice. As shown in the figure, an IDS instance is assumed to be operational in the (general purpose) cloud environment. This IDS is fed by a single tap point at the network ingress/egress. A key difference between DEIDtect and conventional deployments is depicted in Figure 3.1 (b), where a security professional, or the system by itself, determines the need to realize another tap point inside the campus network and create another IDS instance (IDS2) in the cloud platform to monitor this new tap point. Finally, Figure 3.1 (c) depicts an intersite scenario whereby another IDS instance (IDS3) is realized in the cloud, and in this case traffic from a network tap at a remote site is being monitored by the new IDS instance.

The setup in the last scenario accommodates several different use cases. For example, the security administrator of a remote site may wish to have traffic from its network be analyzed by a more sophisticated setup elsewhere. E.g., site B in Figure 2.1, which does not have its own cloud infrastructure, might routinely outsource the security functions of its network to sites A or C. Or site C might run its own Snort instance, but might have a need to perform more detailed analysis using a Bro instance administered at site A. Alternatively, the security administrator of site A in Figure 2.1 might want to investigate an attack originating from site B, and since site B does not have a cloud platform to allow dynamic instantiation of IDS instances, the remote tap traffic is relayed back to site A.

Note that the scenarios illustrated in Figure 3.1 and discussed here are example configurations. A key strength of DEIDtect is its flexible use and manipulation of distributed resources related to security which enables many alternative scenarios.

From an SDN perspective the DEIDtect architecture involves several types of inter-SDN domain interactions. First, within each site the cloud platform and campus network represent two separate SDN domains. As shown in Figure 3.1 (a), the campus network is directed by an enterprise network controller to realize the functionality and policies associated with such an environment. The cloud network, on the other hand, is controlled by a cloud control architecture to realize cloud-specific functionality. The inter-SDN requirement here involves DEIDtect coordinating with both the network



(a)

(b)

(c)

Figure 3.1: DEIDtect Network Functionality

control and cloud control entities, i.e., across two different domains, to realize distributed elastic detection. Specifically, this includes creating tapping resources in the campus network, creating distribution resources in the cloud network, and finally, orchestrating the interconnection of these resources between the two domains. This is depicted in Figure 3.1 (a), with the solid line interconnecting the tap resources in the campus network with the distribution resources in the cloud platform, respectively depicted as different types of dotted lines.

A similar set of inter-SDN domain interactions are involved with the intersite DEIDtect functionality depicted in Figure 3.1 (c). First, the DEIDtect systems in each site need to interact to realize the required functionality, e.g., setting up a (possibly remote) network tap or instantiating a local or remote IDS instance. Following these application-specific interactions, DEIDtect again needs to orchestrate the connection of these sets of resources to realize end-to-end functionality. In this case, however, the orchestration would typically involve interaction with a network controller responsible for interconnecting the distributed sites across the WAN.

CHAPTER 4

RELATED WORK

DEIDtect combines cloud computing and software-defined networking across a variety of domains to realize a distributed network security framework. We touch upon the most relevant related work below.

4.1 SDN in Cloud Networking

The use of OpenFlow to develop a networking infrastructure for the cloud which can support millions of IP and MAC addresses by virtualizing layer 2 network has been proposed [22] . The Cloud Broker work [16] uses OpenFlow to connect multiple data centers via flow-based networking. Support for SDN in the popular OpenStack cloud has also been developed [8]. The Inter-Cloud Network Gateway [24] provides network control and configuration capabilities over a network of distributed cloud resources. CloudWatcher [28] uses OpenFlow to provides monitoring services for large and dynamic cloud networks. In contrast to these works, DEIDtect exposes an SDN cloud abstraction to allow control of the delivery of network traffic to specific virtual machine instances in the cloud.

4.2 SDN in Security

SDN has also been used in the context of enterprise security functionality. Microsoft's Demon [12] does the traffic monitoring using OpenFlow but requires hardware to be installed, whereas the goal of DEIDtect is to avoid the installation of special hardware for monitoring. OpenFlow capabilities have been used for the distribution of traffic load from routers into multiple IDS instances [20]. This approach uses SDN in localized fashion with a static set of IDS resources, lacking DEIDtect's network-wide tapping and elastic compute capabilities. Perhaps most related to DEIDtect's enterprise and cloud interworking, the use of SDN to enable communication between enterprise and cloud platforms and to enable intercloud workflow is suggested in [7]. DEIDtect realizes a framework to enable a security-related workflow that spans across distributed cloud and enterprise instances. To enable the wide-area part of our architecture, DEIDtect also assumes the use of interdomain "stitching" protocols, either using SDN technology [18] or more conventional dynamic circuit establishment [13].

DEIDtect also follows in the footsteps of a variety of distributed security efforts over a long period of time. Dshield [5] is part of the SANS' Internet Storm Center program, allowing firewall users to share intrusion detection information so as to analyze and make it publicly available. Snapp et al. [11] demonstrated a prototype of Distributed IDS (DIDS) that combines distributed monitoring and data reduction with centralized data analysis (through the DIDS Director). A Distributed Intrusion Prevention System (DIPS) has been proposed by Sproull et al. [29]. Our work is complementary to these approaches, focusing on the flexible use of network and cloud resources across different domains to realize security functions in a distributed setting.

4.3 Scalability of Network Security Tools

The scalability of security tools has been addressed by a number of earlier works. For example, IDS clusters to improve the scalability of intrusion detection have been proposed [14, 30]. In [14] a load balancing system is proposed which splits responsibilities of a node to others, replicates traffic to NIDS clusters and aggregate results to split expensive processing at the NIDS. The NIDS cluster work [30] realizes highly scalable intrusion detection by running individual IDS instances in a cluster, exchanging low-level information among instances. With a complete DEIDtect realization we expect to use similar approaches for the DEIDtect cloud-based IDS instances.

4.4 IDS in Cloud

IDS/IPS in the cloud has been proposed to provide security for cloud tenants [21, 23]. A cooperative IDS network is proposed in [21] to prevent DDOS attacks on the cloud. The integration of an IDS into a cloud environment was demonstrated using Eucalyptus [23]. The focus of these existing approaches is on providing cloud security. In contrast, DEIDtect uses the elastic properties of cloud for the security of enterprise networks.

4.5 SDN - Adaptive Load Balancing

SciPass [6] proposes a reactive approach of installing white-listing flows sent towards Firewall from IDS response. DEIDtect's ALB uses the similar approach to realize the throttling feature on a broader scale rather than at a single point of the network. It is done in a distributed environment across domains, which gives much more control on the traffic management towards the IDS.

CHAPTER 5

DEIDtect IMPLEMENTATION

5.1 DEIDtect Core

The DEIDtect Core is the heart of DEIDtect framework and is responsible for orchestration of the DEIDtect Enterprise Controller and DEIDtect Cloud Controller, which dispatches the necessary actions for adding or removing monitoring instances. It listens for any remote requests to allocate resources in the local cloud domain, and issues remote request.

5.1.1 DEIDtect Core - Local Tap Work Flow

Figure 5.1 shows the workflow for a local monitoring instance add request handling in DEIDtect. It also shows all the submodules in each of the domain systems and all the interactions that are done for a monitoring instance creation.

For a Local Site, the “User Tap Request” to DEIDtect Core initiates a “Create VM” request to the DEIDtect Cloud System via the DEIDtect Cloud API module. DEIDtect Cloud API requests the DEIDtect Cloud Helper to create the IDS VM with all the necessary configurations for the Whitelist and CPU usage reporting to the DEIDtect Core framework. It also sets up the tunnel for the tap

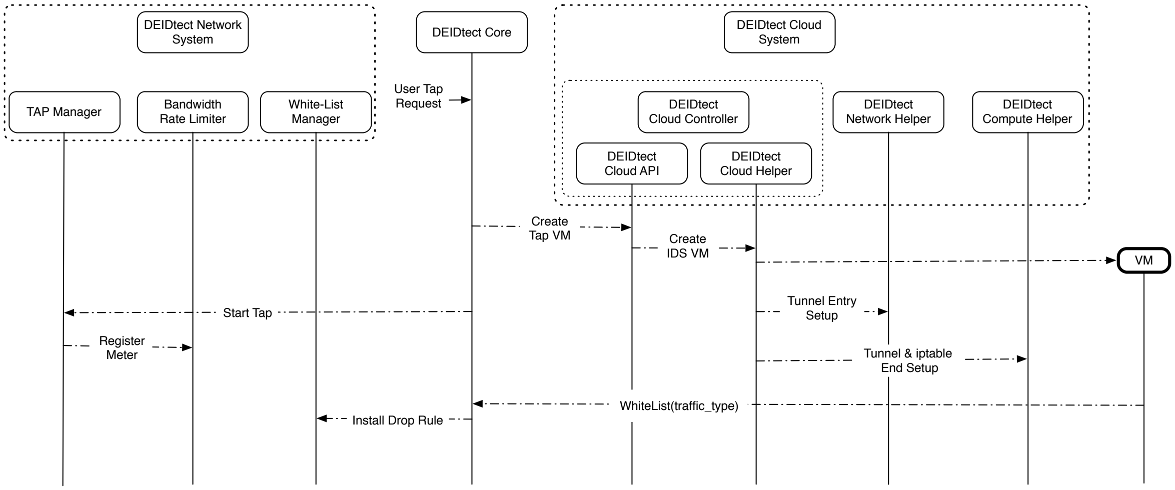


Figure 5.1: DEIDtect Local Tap Request - Work Flow

traffic by sending requests to the DEIDtect Network Helper and DEIDtect Compute Helper. After the IDS VM and the tunnel are set up in the Cloud, DEIDtect Core initiates a “Start Tap” request to the DEIDtect Network System’s Tap Manager submodule. The Tap Manager submodule starts the Bandwidth Rate Limited request to manage the tap traffic. This completes a single “User Tap Request.” The IDS VM reports the traffic to whitelist to the DEIDtect Core whenever it detects a normal traffic flow, which initiates an “Install Drop Rule” to the Whitelist Manager submodule in the corresponding DEIDtect Network System.

5.1.2 DEIDtect Core - Remote Tap Work Flow

Figure 5.2 shows the workflow for a remote monitoring instance add request handling in DEIDtect.

In case of a Remote Site, the “User Tap Request” to a DEIDtect Core (remote) initiates a “Remote Request” to the DEIDtect Core (local) for Cloud Resource allocation. Then the IDS VM creation follows the same work flow as the “Local User Tap Request.” The DEIDtect Core (local) initiates a “Create VM” request to DEIDtect Cloud System via DEIDtect Cloud API module. DEIDtect Cloud API issues a request to the DEIDtect Cloud Helper. DEIDtect Cloud Helper creates the IDS VM with all the necessary configurations for the Whitelist and CPU usage reporting to the DEIDtect framework. It also sets up the tunnel for the tap traffic by sending a request to the DEIDtect Network Helper and DEIDtect Compute Helper. After the IDS VM and the tunnel are set up in the Cloud, DEIDtect Core (remote) initiates a “WAN tunnel request” to the WAN Controller to setup the tunnel between the Remote Site gateway node and the Local Site Cloud Network Node, and a “Start Tap” request to the DEIDtect Network System’s Tap Manager submodule. The Tap Manager submodule starts the Bandwidth Rate Limited request to manage the tap traffic, which completes a single remote “User Tap Request.” The IDS VM reports the traffic to whitelist to the DEIDtect Core (local). Whenever this is detected, this is relayed to the DEIDtect Core (remote). Which in turn initiates an “Install Drop Rule” to the Whitelist Manager submodule in the corresponding DEIDtect Network System.

The event graph in Figures 5.1 and 5.2 shows the events that are taking place in each of the modules of DEIDtect framework to service a single user request for a local and remote site monitoring instance creation/deletion. The traffic evaluation graph is shown in Sections 6.3 and 6.4.

5.2 DEIDtect Network System

The DEIDtect Network System has been built as a ryu application module. The assumptions and required configuration are that the ryu controller has the basic simple switch application enabled to apply the initial flows between the hosts in the enterprise network. The DEIDtect Network System

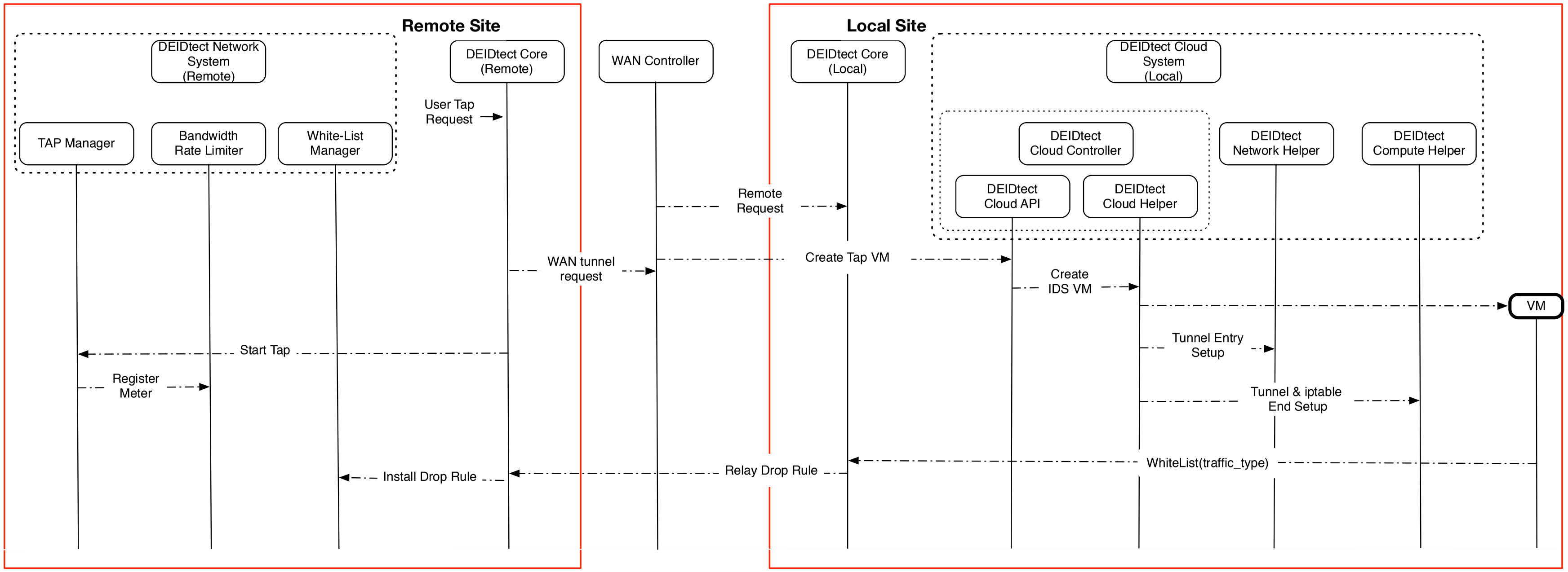


Figure 5.2: DEIDtect Remote Tap Request - Work Flow

requires initial configurations of the hosts connected to the switches and the local/remote gateway switch to which the tap traffic is to be tunneled. This is used to find out the tunnel path from the switch where the tap is to be setup to the local/remote gateway switch to which the traffic is to be delivered. DEIDtect Network System module takes the topology information from a configuration file along with the connected host, cloud gateway, or remote DEIDtect information. It can be extended to get the network topology by running the topology discovery application which comes along with ryu as a native application. This is just to keep things simple in terms of the implementation and does not change any of the claims made for DEIDtect framework.

The DEIDtect Network System is written as a set of ryu application modules. Figure 5.3, shows the DEIDtect Network System in a sample Enterprise environment, with its core components shown in grey boxes (1) Tap Manager - Creates duplicate traffic of specified flows with VLAN tag. (2) Adaptive Rate Limiter - for rate limiting the tap traffic. (3) Whitelisting Module - Installs drop rules of whitelist traffic from IDS VM (feedback loop). (4) Bandwidth Monitor - Calculates the bandwidth usage of the tap port for the Adaptive Rate Limiter module.

5.2.1 ryu-Tap Manager

The ryu Tap-Manager application is the core module responsible for creating the safe tap modification of the flows as per the request. The implementation requires initial flows to be present to create safe tap flows. The tap flow can be installed to monitor an entire port traffic or a specific traffic type on the specified port of the switch. The module communicates with the DEIDtect Core System module via the REST interface in public IP space. For each monitoring instance an associated **Drop Meter** (OpenFlow metering) is created for rate-limiting the 'tap traffic' alone. The meter table serves as the way to rate-limit the amount of traffic being sent to the IDS VM in the cloud. But this rate-limiting

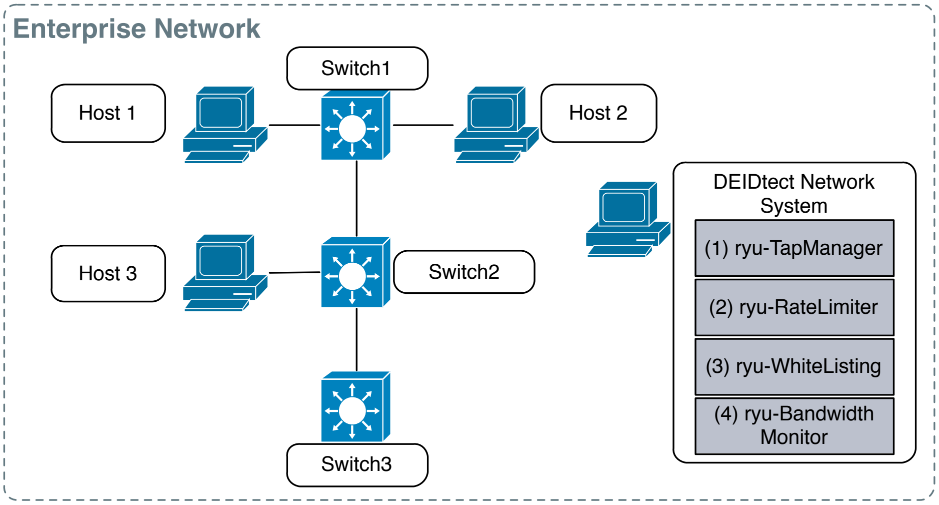


Figure 5.3: DEIDtect Network System - Enterprise Domain

is purely switch-dependent, and there is always high chance of losing useful traffic being sent to the IDS. This will be addressed by the Whitelisting module in the section below. The management of the **Drop Meter** is done by the Adaptive Rate Limiter module, explained in the upcoming section.

5.2.2 ryu-Adaptive Rate Limiter

The ryu Adaptive Rate Limiter application comes into play when there is an active monitoring instance. It runs periodically to check whether the service traffic usage is higher or lower than the tap traffic usage. Based upon the usage, it either increases the tap traffic rate in case of low service traffic usage or decreases the tap traffic rate in case of high service traffic usage. The traffic dropped by the metering is purely switch-based and one does not have any control over the kind of traffic that is dropped. This is a key factor, as the current IDSes requires all the data possible for proper detection of intrusions. Hence we have the Whitelisting module as part of DEIDtect Network System module, which tries to push as much useful 'tap traffic' for IDS analysis.

5.2.3 ryu-Whitelisting

The DEIDtect feedback loop from the IDS VM is used to accomplish the Whitelisting of traffic towards the IDS. This ensures that the traffic being rate-limited by the meter table is useful for the IDS to detect anomalies. The ryu Whitelisting application has a REST interface which accepts the type of flow to be dropped for the specified tap port. The feedback loop greatly helps to achieve greater accuracy of IDS detection in the restricted resource environment of DEIDtect. The Whitelisting requests are sent by the DEIDtect Core System module as described in the design of the framework, as the IDS VM cannot directly contact the associated DEIDtect Network System module because of the different domains they reside in. The Whitelist request from IDS is sent to the DEIDtect Core and relayed to the DEIDtect Network System if it is a local system or Remote DEIDtect Core instance of the associated monitoring instance.

5.2.4 ryu-Bandwidth Monitor

The ryu Bandwidth Monitor application is used to poll for switch port statistics to calculate traffic being sent out on the switch ports. This is used to calculate the amount of service traffic that is being sent to that port, and the rest of the unused bandwidth can be used to send the tap traffic. Since this information is only required when there is an active tap port, the bandwidth monitoring is done only when there is an active monitoring instance.

5.3 DEIDtect Cloud System

The cloud environment being used for DEIDtect is OpenStack [26]. The deployment information is shown in Figure 5.4.

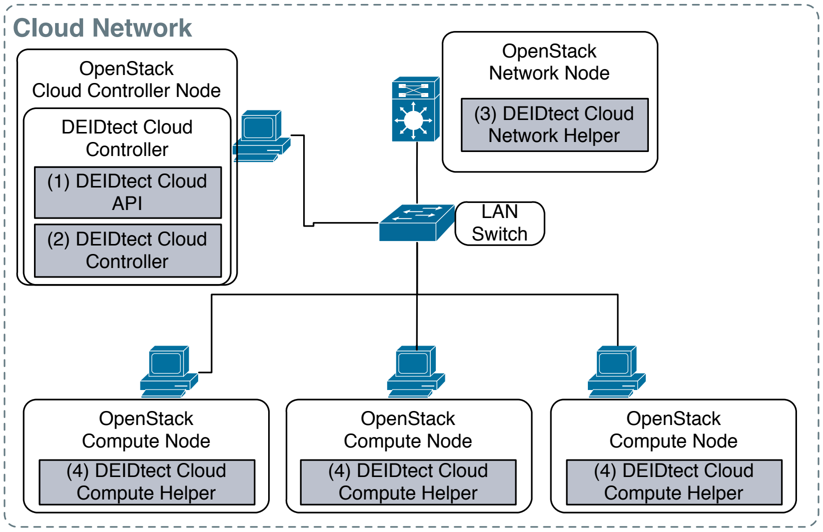


Figure 5.4: DEIDtect Cloud System - OpenStack

The OpenStack is deployed using the Open-VSwitch plugin to have the SDN environment in cloud and uses Generic Routing Encapsulation (GRE) network isolation type. The OpenStack cloud deployment for DEIDtect Cloud System is shown in Figure 5.4, for which the primary function is to create/delete IDS VMs in the cloud and create a tunnel from the cloud gateway to the specific VM created for the tap port. The way it accomplishes this is described in the sections below. The communication between these modules happen on the OpenStack management network is shown in Figure 5.5 for keeping the access within the local domain. Only the DEIDtect Cloud Controller can be accessed by the DEIDtect Core module.

The OpenStack deployment consists of a Cloud Controller Node, a Network Node, and a set of Compute Nodes, as shown in Figure 5.4. The figure shows only the datapath connection of the network, but the deployment also has another LAN for the management network, which is not shown.

The DEIDtect Cloud System consists of the DEIDtect Cloud Controller comprising (1) the DEIDtect Cloud API layer to enable external access to the DEIDtect Cloud System APIs (2) the DEIDtect Cloud Controller Helper to orchestrate the underlying modules to accomplish the tunnel creation/deletion and IDS VM instance creation/deletion, (3) the DEIDtect Cloud Network Helper to add/remove OpenFlow rules for tunnel creation/deletion in the OpenStack Network Node and (4) the DEIDtect Cloud Compute Helper to add/remove OpenFlow rules for tunnel creation/deletion in the OpenStack Compute Node along with iptable rule installations.

5.3.1 DEIDtect Cloud Controller

This is a python module running on the OpenStack Cloud Controller Node which comprises two sub modules **DEIDtect Cloud API layer** and **DEIDtect Cloud Controller Helper**, as show

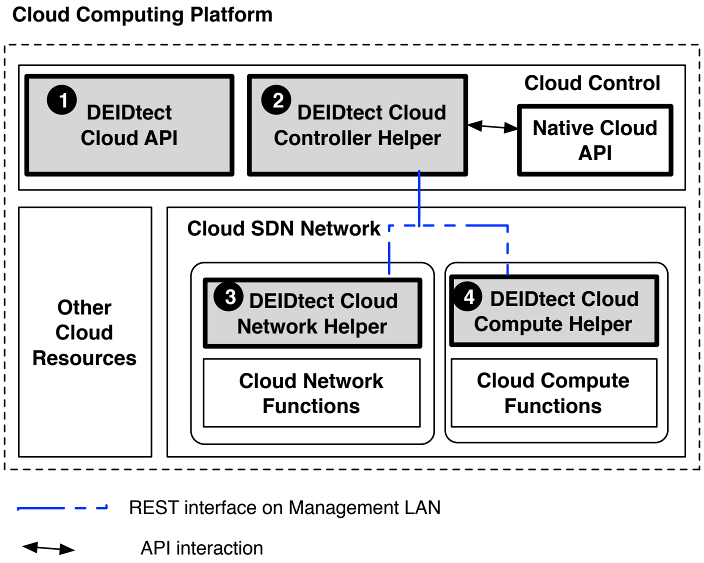


Figure 5.5: DEIDtect Cloud System - Communication Path

in Figure 5.4 blocks labeled (1)&(2). The DEIDtect Cloud API layer, in Figure 5.4 (1) is a python REST API layer for inter module communication. The DEIDtect Cloud Controller module uses the OpenStack API to manage the instance creation, security group rule management for the IDS, and interface configurations for it. The DEIDtect Cloud Controller module shown in Figure 5.4 (2) creates a tunnel path from the *OpenStack Network Node* to the corresponding *OpenStack Compute Node* where the VM is hosted. The OpenStack API provides rich capabilities to find the virtual (not physical) network path through which the tunnel needs to be created. This is used by DEIDtect Cloud Controller to install OpenFlow rules to create a tunnel path for tap traffic and modify iptables at the compute node to allow all traffic to VM. The iptables custom rule is to allow tap traffic to be delivered to the interface as the destination IP is not the IDS VM's IP. The DEIDtect Cloud Controller module (2) communicates with DEIDtect Cloud Network module (3) and DEIDtect Cloud Compute module (4) to orchestrate the IDS and tunneling.

The IDS VM created has two different virtual network interfaces attached to it, one for VM access via internet using the floating IP associated with the interface, and the second one for the 'tap traffic' to be delivered to the VM. These two interfaces are part of different virtual networks created in OpenStack.

The image used by the DEIDtect Cloud Controller to create the VM is preconfigured with BRO IDS listening on second ethernet interface (eth1). The DEIDtect Cloud Controller also passes on the cloud-init script to configure the BRO IDS in standalone/cluster mode depending upon the user request. The cloud-init script is dynamically created as per the request, and it also has the DEIDtect Network System module IP and the switch Data Path ID (DPID) for the Whitelisting module in the

VM to send the Whitelisting request to.

The communication between these modules happens over the OpenStack Management network, making it accessible only within the closed network.

5.3.2 DEIDtect Network Helper

This is a python module, as shown in Figure 5.4 in the block labeled (3), which runs on the OpenStack Network Node and has a REST interface listening on the management network of OpenStack. The OpenStack Network Node has three ovs-bridges **br-ex, br-int, br-tun** connected via veth pair bridges called patch-ports as shown in Figure 5.6.

To create the cloud network tunnel for the tap traffic the flows are installed in all the three bridges which are tagged with the VLAN 'X' seen in **br-ex**, and forwarded to **br-int** and onto **br-tun**. The **br-tun** has all the GRE port end points connected to all OpenStack Compute Nodes. The GRE out port is found using the destination IP of the destination OpenStack Compute Node. The VLAN tag is removed and a tunnel ID 'Y' is set to send it via the GRE port. The VLAN is later used in the OpenStack Compute Node to again recover the traffic after it is obtained from the GRE tunnel port based on the tunnel ID. So each 'tap traffic' requires two distinct IDs, the tunnel ID and VLAN ID as per the OpenStack's current design for proper traffic delivery to the VM in the DEIDtect framework. The OpenFlow rules installed for creating the tunnel to deliver the tap traffic to the Compute Node at each of the ovs-bridges in the Network Node are shown in Figure 5.7. The deletion of the tunnel is the removal of all the associated flows in each of the bridges.

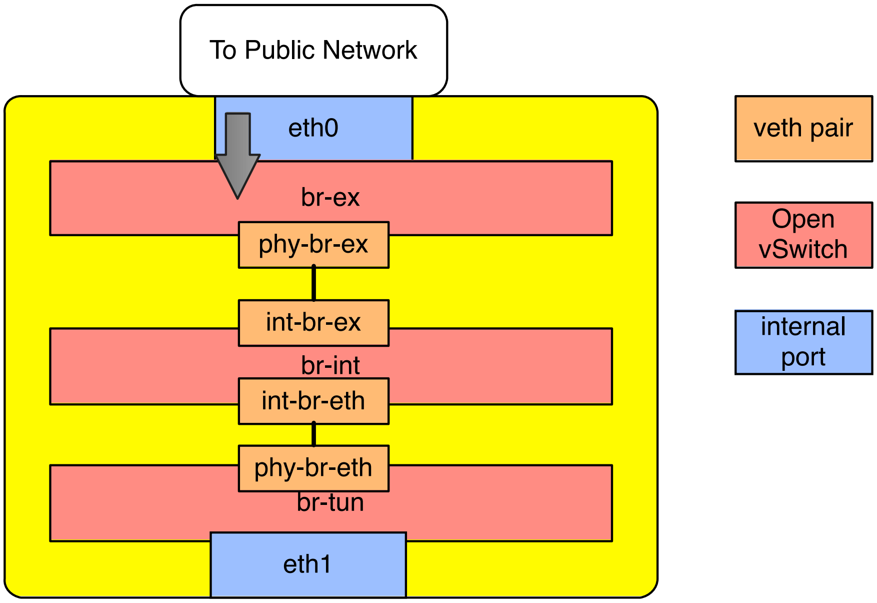


Figure 5.6: OpenStack Network Node - OpenVSwitch Bridges

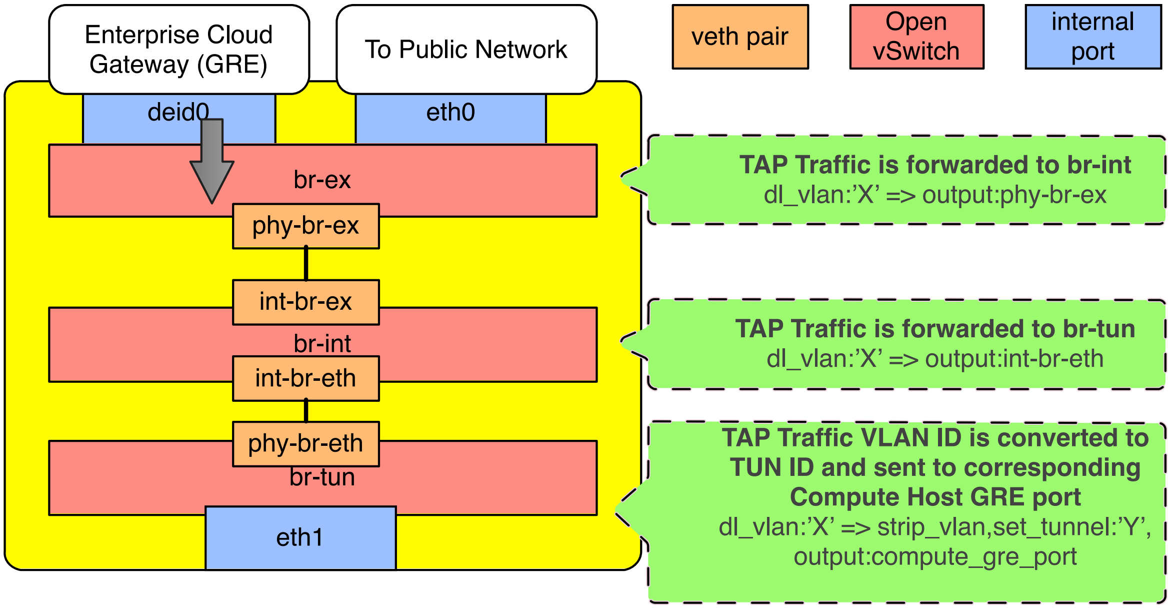

Enterprise Cloud Gateway (GRE)
To Public Network
veth pair
Open vSwitch
internal port
deid0
eth0
br-ex
phy-br-ex
int-br-ex
br-int
int-br-eth
phy-br-eth
br-tun
eth1
TAP Traffic is forwarded to br-int
dl_vlan:'X' => output:phy-br-ex
TAP Traffic is forwarded to br-tun
dl_vlan:'X' => output:int-br-eth
TAP Traffic VLAN ID is converted to TUN ID and sent to corresponding Compute Host GRE port
dl_vlan:'X' => strip_vlan,set_tunnel:'Y', output:compute_gre_port

Figure 5.7: OpenStack Network Node - OpenVSwitch Under The Hood

5.3.3 DEIDtect Compute Helper

This is again a python module as in the Figure 5.4 block labeled (4), which runs on each of the OpenStack Compute Node with a REST interface on the management network of OpenStack. The OpenStack Compute Node has two ovs-bridges **br-tun, br-int** connected via veth pair bridges called patch-ports, as shown in Figure 5.8.

As per the request from the DEIDtect Cloud Controller, the openflow rules are pushed to the **br-tun, and br-int**. The **br-tun** consists of all the GRE port end points between OpenStack Compute Node and OpenStack Network Node. The tunnel rules are installed in **br-tun** to forward the traffic of the specified tunnel ID 'Y' and add the VLAN 'X' back to it and send it to the **br-int**. The **br-int** then pops the VLAN tag and forwards it to the port on which the IDS VM is attached (eth1). The OpenFlow rules installed for the tunnel traffic delivery to the VM at each of the ovs-bridges in the Compute Node are shown in Figure 5.9. The deletion of the tunnel is the removal of all the associated flows in each of the bridges.

The module also installs iptable rules associated with the interface. Though the controller has created the necessary rules to allow all traffic to the IDS from any sources, the traffic has to be destined to the IDS. The traffic with a different destination address will not be allowed to get delivered to the VM. Hence the DEIDtect Compute Helper installs necessary iptables. This ensures that the interface is put in promiscuous mode and all the tunnel traffic is sent to the IDS without the firewall drops. Also, the VM's iptables are to be configured to allow all ingress traffic in the monitoring interface and deny all egress traffic. Cloud-init can be used during initial boot up of VM.

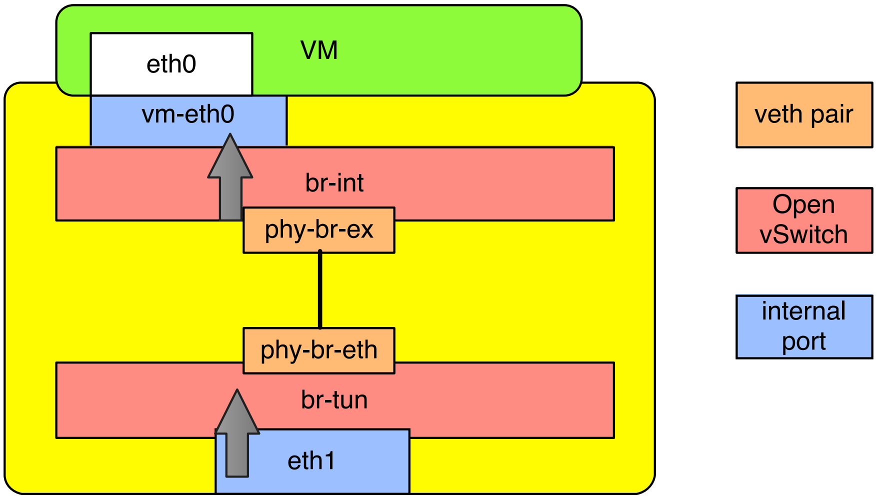


Figure 5.8: OpenStack Compute Node - OpenVSwitch Bridges

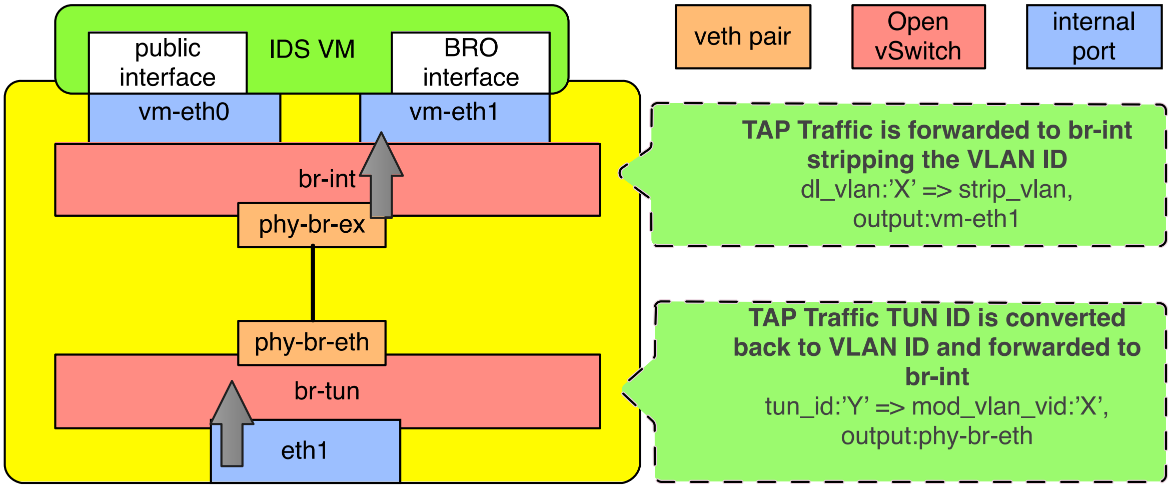

public interface
IDS VM
BRO interface
vm-eth0
vm-eth1
br-int
phy-br-ex
phy-br-eth
br-tun
eth1
veth pair
Open vSwitch
internal port
TAP Traffic is forwarded to br-int stripping the VLAN ID
dl_vlan:'X' => strip_vlan, output:vm-eth1
TAP Traffic TUN ID is converted back to VLAN ID and forwarded to br-int
tun_id:'Y' => mod_vlan_vid:'X', output:phy-br-eth

Figure 5.9: OpenStack Compute Node - OpenVSwitch Under The Hood

CHAPTER 6

DEIDtect EVALUATION

6.1 Questions answered by this evaluation

These are the questions we answer through this evaluation.

- How easy is it to setup a monitoring instance for a local site and a remote site?
- Does the IDS VM detect the anomalies using the tunnel traffic?
- How is bandwidth of tap traffic being managed at different service traffic rates?
- How does the Whitelisting feature of the framework act based upon Bro script and whitelist target traffic?
- How does IDS scaling happen at different IDS VM usage?
- What is the impact of traffic rate limiting on IDS detection?

6.2 Experimental Setup

The testbed depicted in Figure 6.1 has been deployed in Emulab [15]. This figure shows the assumed physical topology. The logical view of the testbed is shown in Figure 6.2. The local and remote enterprise networks are emulated using *Mininet* [19] with CPqD [10] software switches. The testbed comprises (a) Remote Enterprise Network, (b) External node which acts as Wide Area Network (WAN), (c) Local Enterprise Network, and (d) OpenStack Cloud Environment.

The hardware switches supporting OpenFlow 1.3 protocol which were available for this work did not support multitable and meter table, which are primary requirements in an enterprise network environment for DEIDtect deployment. Hence we used the software switch which had the required features for the enterprise network. The software switch was not used in Emulab nodes, as the experiments in the testbed were isolated using VLANs, but DEIDtect also requires VLANs for its isolation. Hence we used Emulab nodes to emulate an enterprise network using Mininet. Mininet has the capability to attach physical interface bridges to the emulated network switches, which is how the enterprise network is connected to the physical gateway interface.

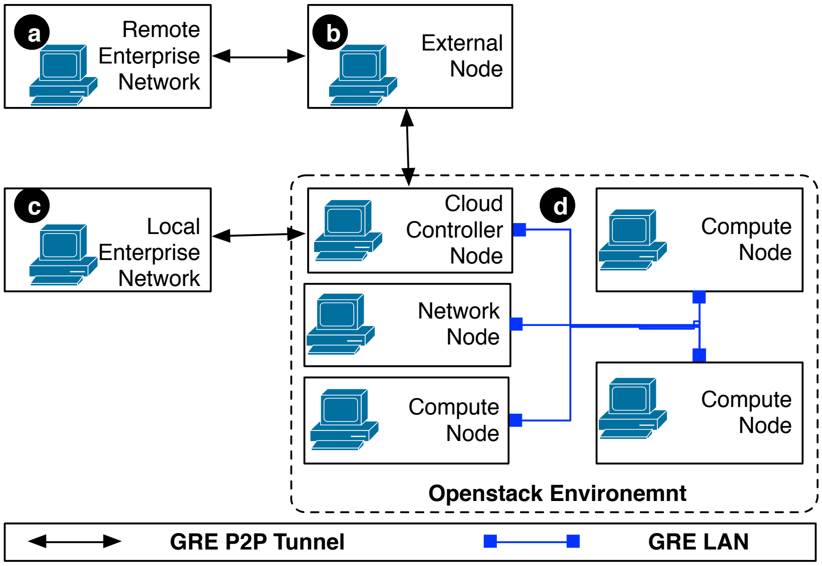


Figure 6.1: EMULAB Physical (Datapath) Testbed Topology

The cloud software used for the deployment is OpenStack [26] deployed in Emulab. This is open source software which integrates OpenFlow-based software-defined networking (SDN), enabling automation and provisioning of network services. OpenStack deployment for DEIDtect uses the Open vSwitch plug-in and GRE tunneling network options for isolation. The reason to use GRE tunneling instead of VxLAN is that the deployment is on top of Emulab. Since Open vSwitch is the only option available in the OpenStack to enable SDN for network management, DEIDtect implementation could not have metering support in the Cloud domain. The DEIDtect test-bed's OpenStack deployment has (1) a cloud controller node, (2) a network node, (3) three compute nodes connected in two LAN networks. One of the LANs is for management traffic and the other is for datapath. The network node is connected to an external node (in OpenStack terms) or WAN controller in the DEIDtect testbed environment, which is for external internet access to the cloud VMs using floating IPs.

OpenStack manages virtual networks in the Cloud, which is done using Open vSwitch on the nodes. Figures 5.6 and 5.8 show the bridges created and maintained by OpenStack for virtual network management for VMs. To deal with scalability issues, recent releases of OpenStack uses an OpenFlow Agents rather than a centralized controller to install flow entries, create logical ports, and complete other Open vSwitch operations. Hence OpenStack deployment assumes connectivity between the different nodes as show in Figure 6.1 "OpenStack Environment." The current OpenStack does not have any knowledge about the underlying network device and the topology deployed of the physical access to the network devices.

Without the details of network topology of the devices in the cloud environment we cannot implement the tap traffic rate-limiting in the cloud as done in enterprise network with metering.

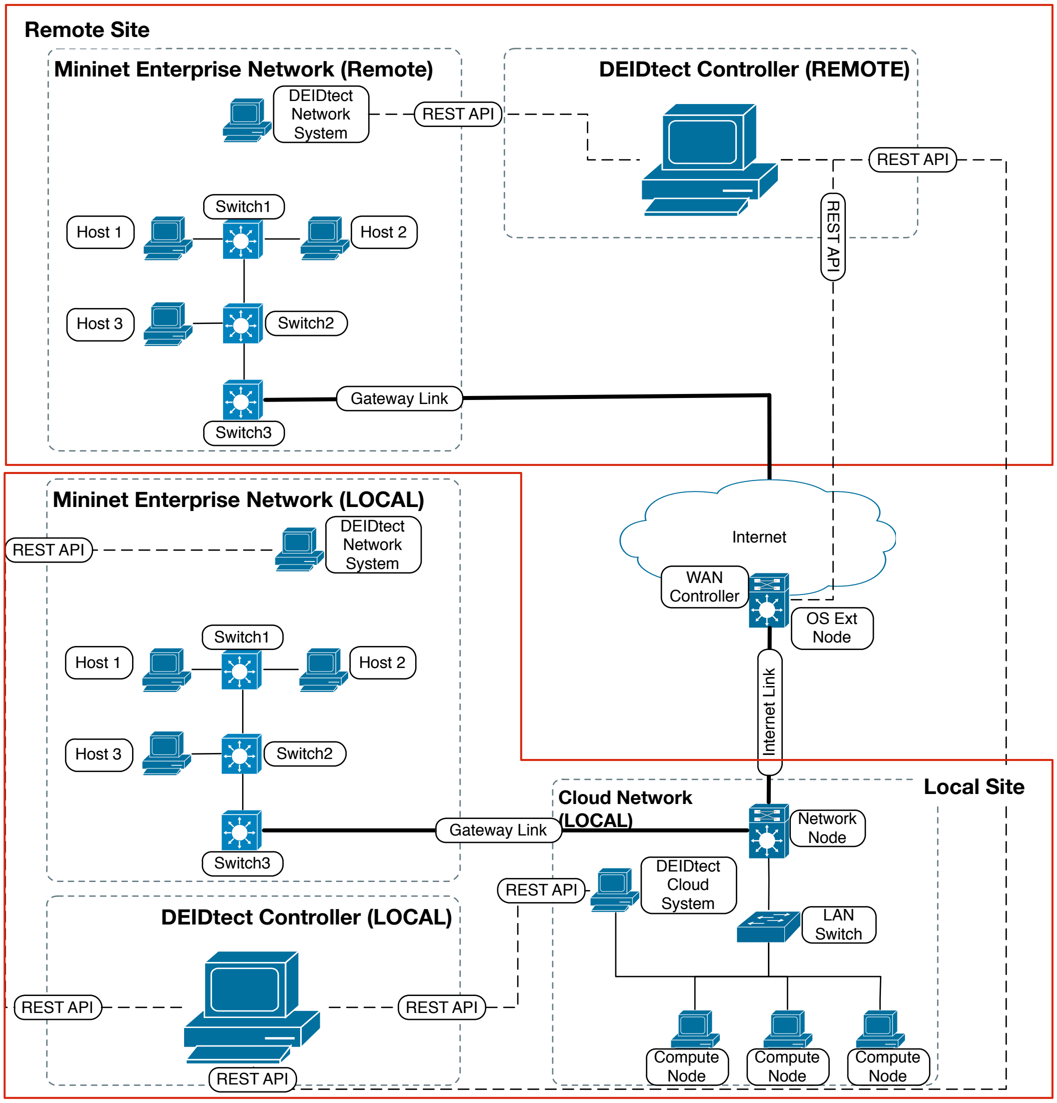


Figure 6.2: DEIDtect Logical Testbed Topology

In a scenario where the same tap traffic is to be delivered to different set of IDS having different monitoring tasks, the traffic cannot be split at the lowest possible network subtree (to avoid traffic duplication). Hence the tap traffic management is restricted to the enterprise network domain of DEIDtect architecture. This is not a limitation of the DEIDtect design but rather of the OpenStack.

The enterprise network node's gateway link is connected to the OpenStack network node via a GRE tunnel in case of a local site, as labeled in Figure 6.2. In case of a remote site, the enterprise gateway GRE link is connected to the external node, which is considered logical Wide-Area Network (WAN). The WAN controller runs in the external node and the external node is connected to the OpenStack network node via GRE tunnel.

6.2.1 Tools Used

For generating UDP traffic, *mausezahn* (mz) and *iperf* are used. For Secure Ftp server, *very secure FTP daemon* (vsftpd) is used. For traffic monitoring, the *Bandwidth Monitor NG* (bwm-ng) tool is used. The CPU usage stats are obtained from th VM using *Collection Daemon* (collectd).

6.3 DEIDtect End-to-End - Local Tap

This section shows the end-to-end traffic flow when a local tap is created. We evaluate the end-to-end working of framework for a local tap in the topology shown in Figure 6.3., i.e., a subset of the evaluation setup shown in Figure 6.2.

6.3.1 Test and Result

The traffic is being monitored at each of the links labeled in Figure 6.3: (a) Service traffic from Host 1 (H1) towards Host 3 (H3), (b) Service traffic towards Host 3, (c) Tap traffic towards cloud gateway, (d) Tap traffic received by the OpenStack Network node, (e) Tap traffic sent to the corresponding IDS VM on the OpenStack Compute Nodes. Figure 6.4 shows the network activity graph of the labels (a) to (e) seen in Figure 6.3.

Figure 6.4 shows a number of time series events corresponding to the creation of a local tap point on Switch 1 (S1) to monitor all traffic sent out on port 3 through the topology.

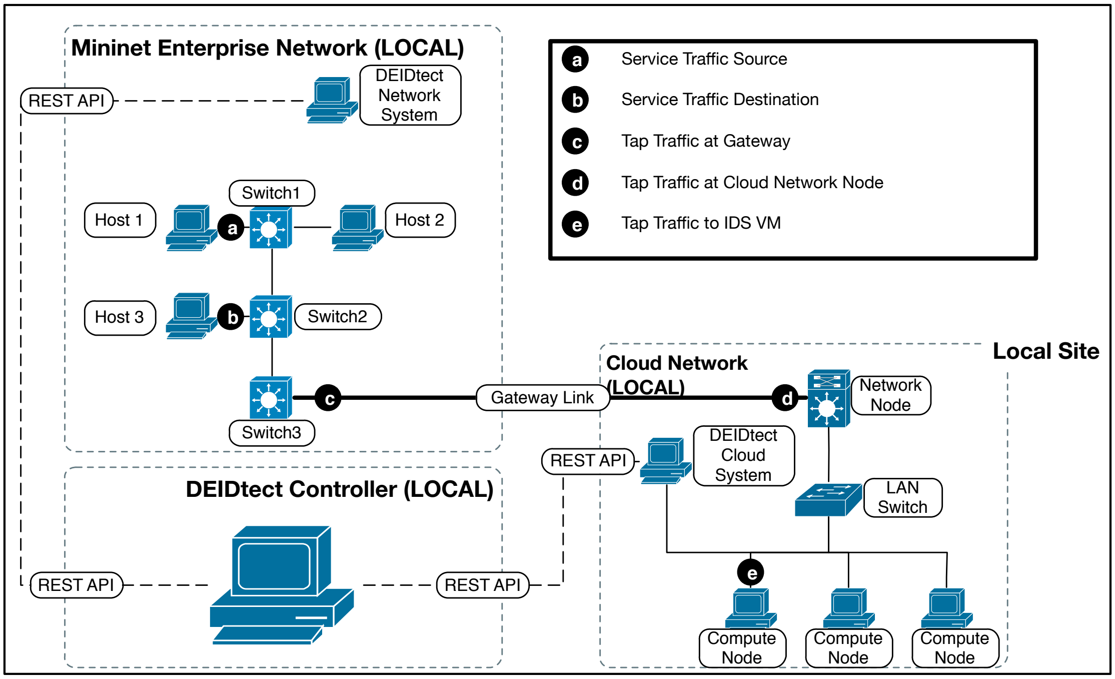


Figure 6.3: DEIDtect Local Tap - Topology

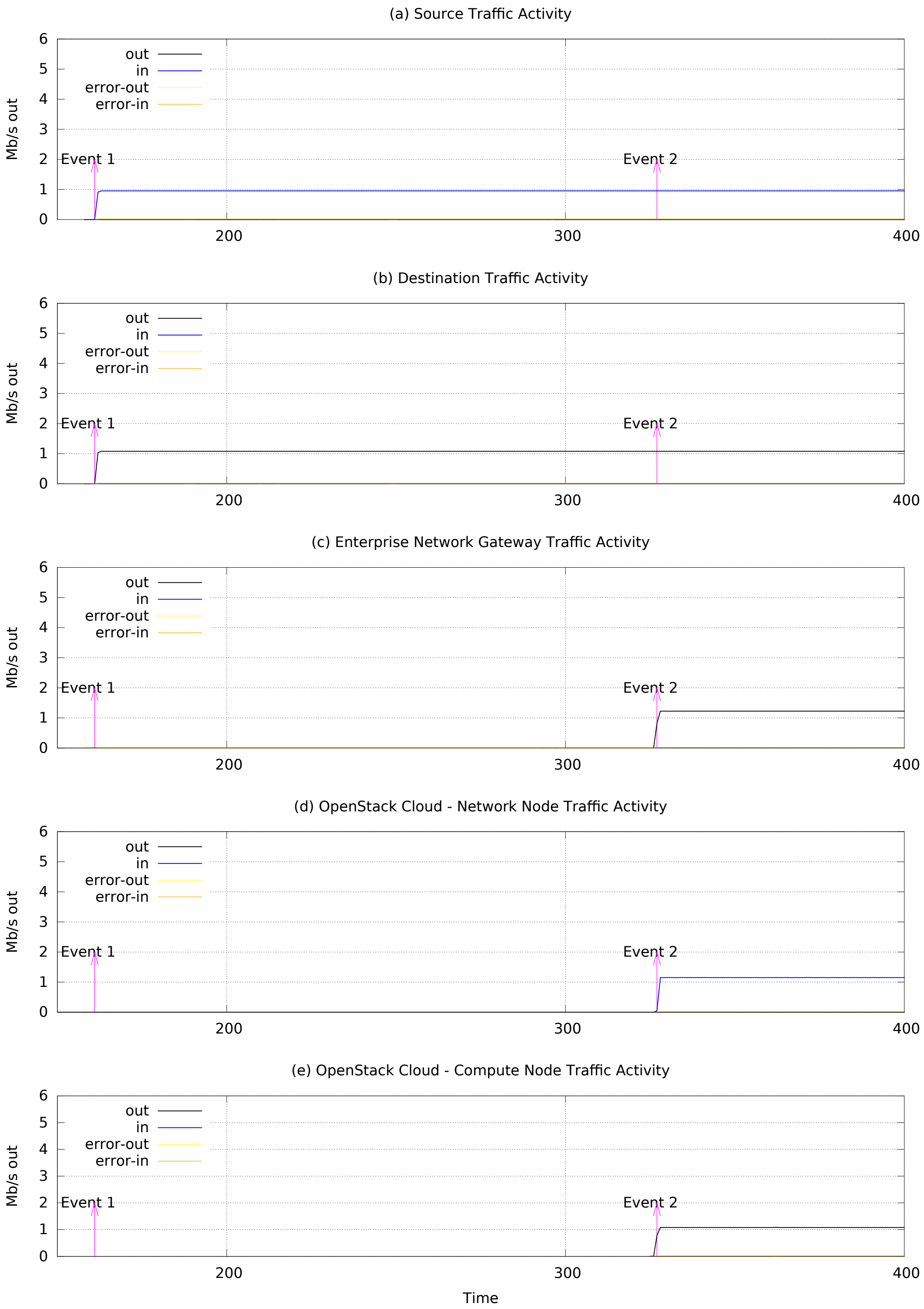


Figure 6.4: DEIDtect Local Tap Event Graph

We now describe each event shown in the figure: (1) Traffic sent from H1 to H3. Figure 6.4 (a) shows the traffic sent from H1. (b) shows traffic received on H3, (2) Create Tap point on S1 port 3, Figure 6.4 (c) shows tap traffic out of the gateway node, (d) shows tap traffic received by the OpenStack Network Node and (e) shows tap traffic sent to the IDS VM created for the corresponding tap point.

From the results we can see how the tap traffic is tunneled towards the IDS VM for a local tap by DEIDtect, as per the workflow in Figure 5.1.

6.4 DEIDtect End-to-End - Remote Tap

This section shows the end-to-end traffic flow when a remote tap is created. We evaluate the end-to-end working of framework for a local tap in the topology shown in Figure 6.5.

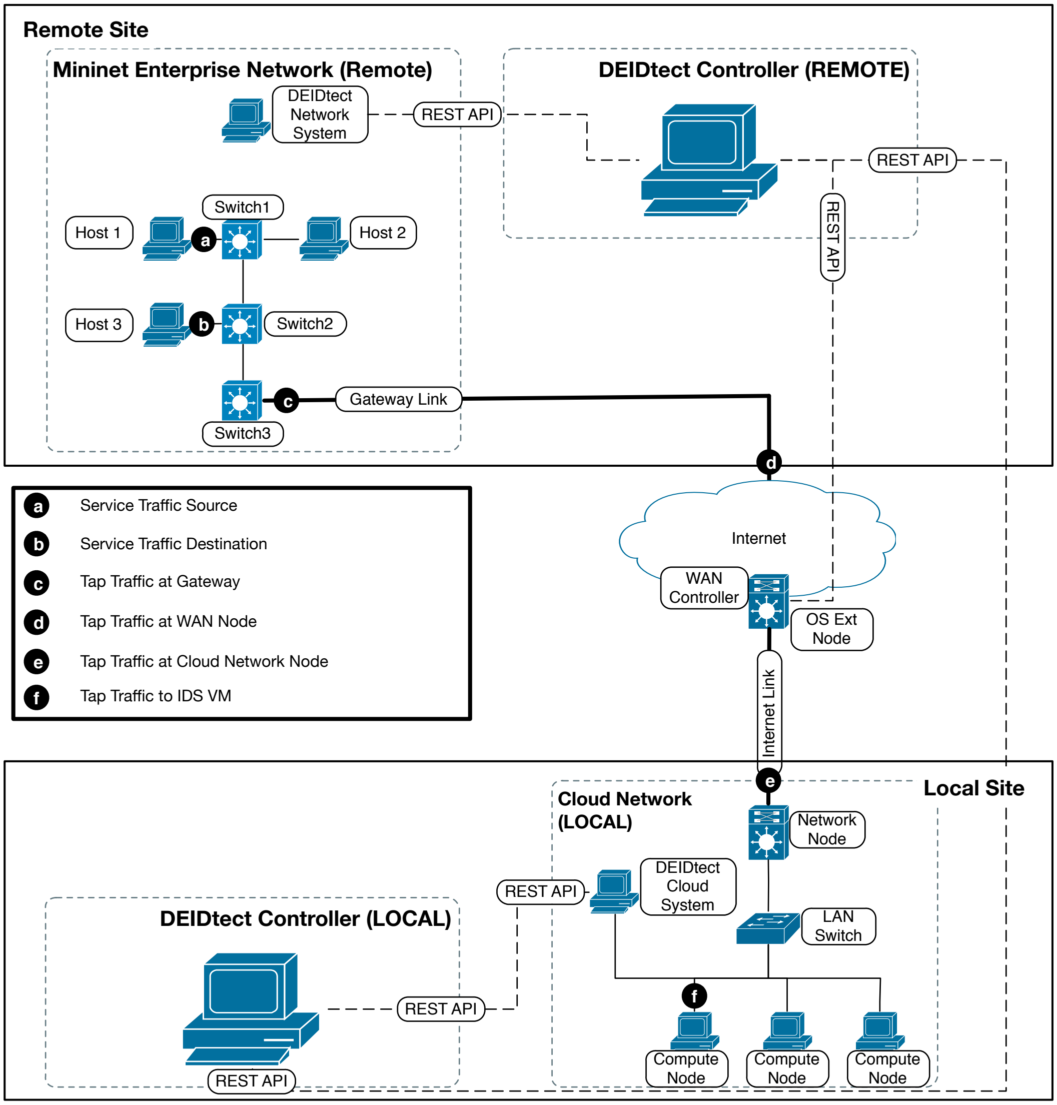


Figure 6.5: DEIDtect Remote Tap - Topology

6.4.1 Test and Result

The traffic is being monitored at each of the links labeled in Figure 6.5: (a) Service traffic from Host 1 (H1) towards Host 3 (H3), (b) Service traffic towards Host 3, (c) Tap traffic towards cloud gateway, (d) Tap traffic received by the WAN node, (e) Tap traffic received by the OpenStack Network node, (f) Tap traffic sent to the corresponding IDS VM on the OpenStack Compute Nodes. Figure 6.6 shows the network activity graph of the labels (a) to (f) seen in Figure 6.5.

Figure 6.6 shows a number of time series events corresponding to the creation of a remote tap point on Switch 1 (S1) to monitor all traffic sent out on port 3 through the topology. We now describe each event shown in the figure: (1) Traffic sent from H1 to H3. Figure 6.6 (a) shows the traffic sent from H1. (b) shows traffic received on H3, (2) Create Tap point on S1 port 3, Figure 6.6 (c) shows tap traffic out of the gateway node, (d) shows tap traffic received by the WAN node, (e) shows tap traffic received by the OpenStack Network Node and (f) shows tap traffic sent to the IDS VM created for the corresponding tap point.

From the results we can see how the tap traffic is tunneled towards the IDS VM for a remote tap by DEIDtect, as per the workflow in Figure 5.2.

6.5 DEIDtect Ease Of Use

We illustrate DEIDtect's ease of use by showing how DEIDtect provides a high-level abstraction to the underlying tedious process in creating/destroying a monitoring instance.

The creation or deletion of a tap using DEIDtect is done by a single REST request to the DEIDtect framework.

The general REST request format for the current implementation is shown below.

```
curl http://{DEIDTECT_IP}:120/deidtect/{add/del}/{IDS VM name}/
        {switch DPID}/{port to monitor}/{vlan ID}/{tunnel ID}
```

This can be further simplified by managing the VLAN, TUNNEL ID and a dynamic tap name allocation which will require only the create/delete action sent along with the switch name and switch port to DEIDtect Core. The simplified version in case of managing the IDs and name is shown below.

```
[Managed] curl http://{DEIDTECT_IP}:120/deidtect/{add/del}/
        {switch DPID}/{port to monitor}
```

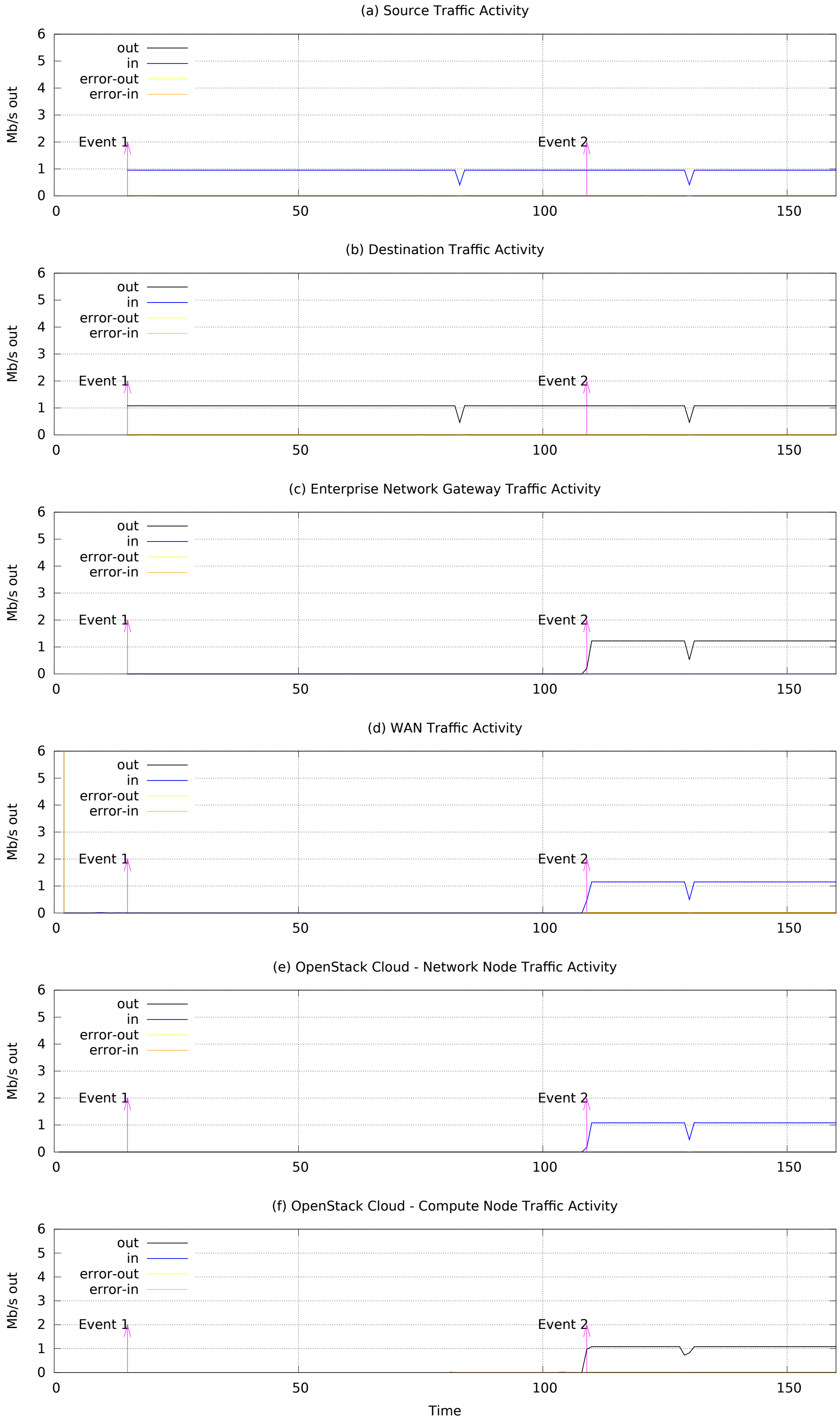


Figure 6.6: DEIDtect Remote Tap Event Graph

6.5.1 Examples

Adding and deleting a monitoring point for DEIDtect running on “155.98.38.103” in a local Enterprise Network with DPID ‘0000000000000001’ on port 3, using isolation Vlan ID 100 and Tunnel ID 7 and naming the newly created monitoring IDS *monpt*, is done by the following curl request.

```
[ADD] curl http://155.98.38.103:120/deidtect/add/
    monpt/0000000000000001/3/100/7

[DEL] curl http://155.98.38.103:120/deidtect/del/
    monpt/0000000000000001/3/100/7
```

The State-of-the-art monitoring setup requires manual effort to setup the optical taps at the network device links and the corresponding IDS system, which is guaranteed to take significantly longer to realize compared to the usage of DEIDtect.

6.6 DEIDtect Cloud IDS Detection

DEIDtect is built on the insight that traffic monitoring can be done by replicating traffic and delivering the replicated traffic using tunneling protocols like VLAN, MPLS, or GRE. In this section we show that the tunneled traffic can be used for monitoring using the IDS.

In this section we evaluate two cases of IDS detection: one where attack traffic is sent directly to the IDS system and another where attack traffic is replicated using the DEIDtect framework.

6.6.1 Metrics

The metric being used is the *detection in direct traffic vs DEIDtect tunneled traffic*.

6.6.2 Test and Result

To test the Bro detection in a direct traffic scenario, we have setup two pc3000 nodes directly connected in Emulab, as shown in Figure 6.7. The “attacker system” sends the below listed traffic directly to the “Bro system”:

- Port scan
- File transfer
- TCP syn attack

The test result is successful when the detections are the same with the “direct” and the “tunneled” traffic. This is sufficient to prove that the tunneled traffic can be used to detect anomalies.

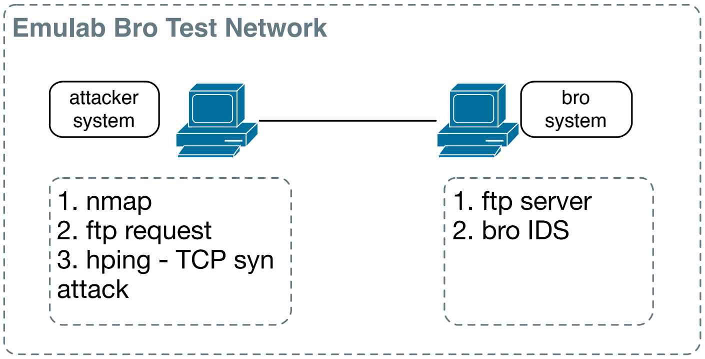


Figure 6.7: Bro IDS Detection Test Topology

For the tunnel traffic scenario, we use the same testbed as shown in Figure 6.2, creating a monitoring point in local enterprise network Switch S1 and port 3. The attacker is the Host 1 (H1) and the destination is Host 3 (H3). Creating a monitoring point only monitors the traffic going out of port 3, and if we want to monitor the ingress traffic, then we have to create a monitoring point in Switch S2 and port 1. In this experiment we monitored the outgoing traffic at S1 and port 3.

Table 6.1 shows the tests performed on each setup, and their results. The result shows that the tunneled traffic can be used for detecting the anomalies using IDS.

6.7 Mininet - CPqD User Space Switch Benchmark

To establish a baseline in this section we show the base performance of the CPqD switch used in the mininet environment for Enterprise Network Environment.

6.7.1 Metrics

The metric being used is the *traffic throughput achieved* at different traffic rates when using VLAN actions.

6.7.2 Test and Result

The experiment is performed in the simple test topology, which has Host 1 (H1) connected to Switch 1 (S1) connected to Host 2 (H2) via Switch 2 (S2), shown in Figure 6.8.

Table 6.1: Bro Detection Results

S.No	Test	Detection Result	
		Bro System	Bro IDS VM
1	Port Scan	Yes	Yes
2	FTP	Yes	Yes
3	TCP Syn attack	Yes	Yes

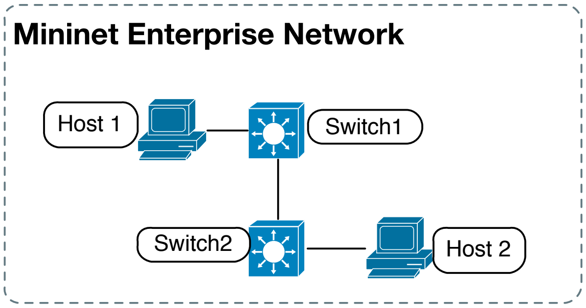


Figure 6.8: CpQD Benchmark Topology

UDP traffic is generated from Host 1 (H1) to Host 2 (H2). The flows from H1 to H2 in Switch 1 (S1) and Switch 2 (S2) are manually configured to add VLAN 100 at S1 for all the traffic from H1, S2 pops the VLAN tag and forwards to H2. The flow rules for each switch are shown below.

```
[Switch 1] dpctl unix:/tmp/s1 flow-mod cmd=add,table=0 in_port=1
        apply:push_vlan=0x8100,set_field=vlan_vid:100,output=2
```

```
[Switch 2] dpctl unix:/tmp/s2 flow-mod cmd=add,table=0 vlan_vid=100
        apply:pop_vlan,output=2
```

Figures 6.9 and 6.10 show the throughput achieved at different traffic rates. The destination shows a slightly higher traffic rate, which is the effect of adding VLAN header on top of it. The results show that the CPqD user switch performs well for traffic rates up to 4 Mbps, after that the throughput is unpredictable, with many drops and the maximum output to be approximately 5.5 Mbps, even at 8 Mbps of input traffic.

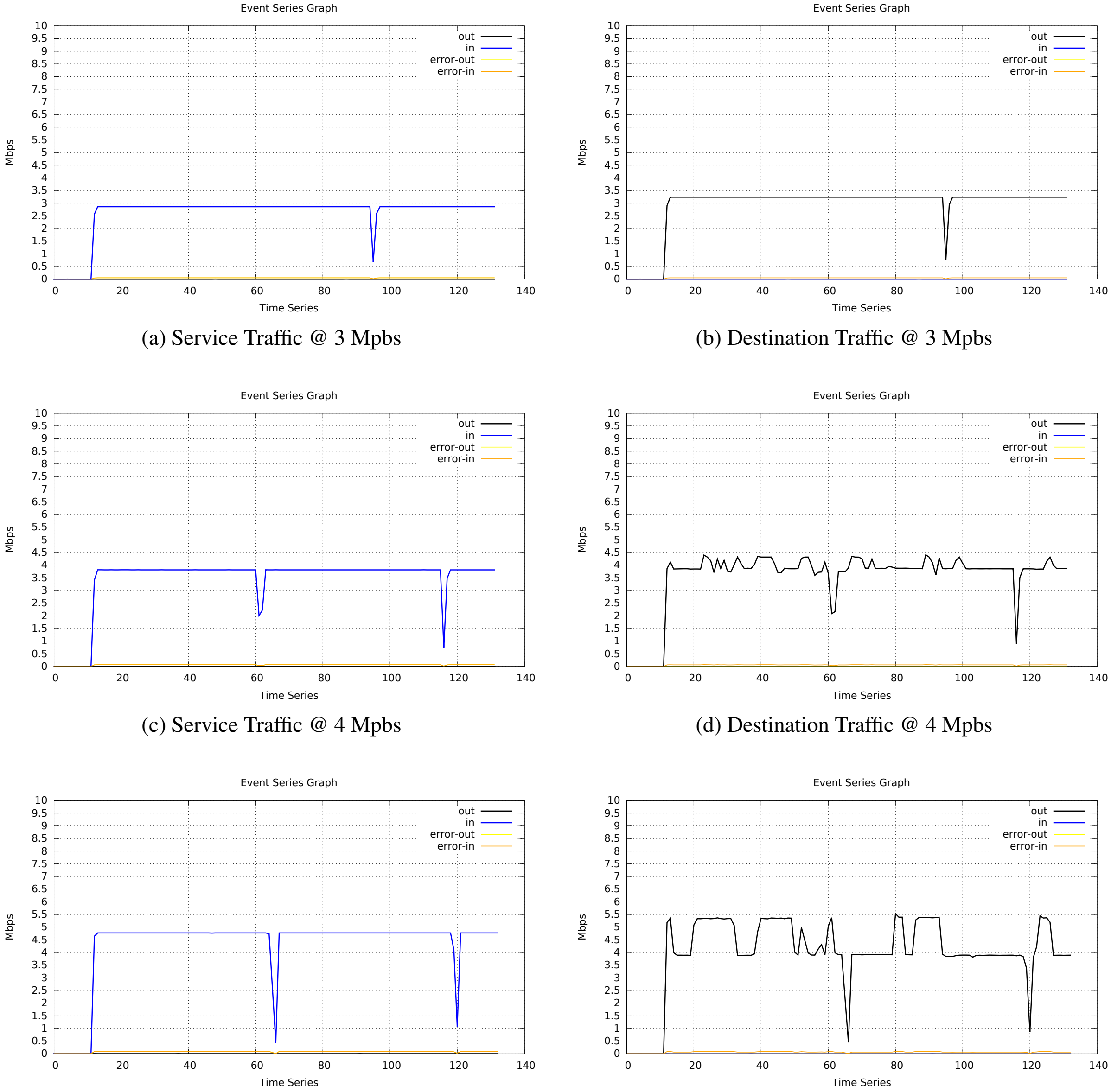
From the experiment, the CPqD user switch operating range in the given topology without traffic rate loss is up to 4 Mbps. This will be the range used for the rest of the experiments.

6.8 Bandwidth Management for Tap Traffic

In this section we evaluate ALB rate-limiting of DEIDtect framework. The rate-limiting is a dynamically controlled sliding window-like mechanism to allow service traffic to be minimally impacted by tap traffic. This evaluation shows how the tap traffic is getting rate-limited to different rates compared to the current service traffic rates, which reside in the same path of the tap traffic. Though the networks are usually overprovisioned to accomodate higher rates of traffic than expected, using DEIDtect in such networks makes good use of the available bandwidth for traffic monitoring. This is done in a more controlled way by the DEIDtect ALB rate-limiting feature.

6.8.1 Metrics

The metric being used is the *different service traffic rate vs tap traffic rate-limit*.



(a) Service Traffic @ 3 Mpbs

(b) Destination Traffic @ 3 Mpbs

(c) Service Traffic @ 4 Mpbs

(d) Destination Traffic @ 4 Mpbs

(e) Service Traffic @ 5 Mpbs

(f) Destination Traffic @ 5 Mpbs

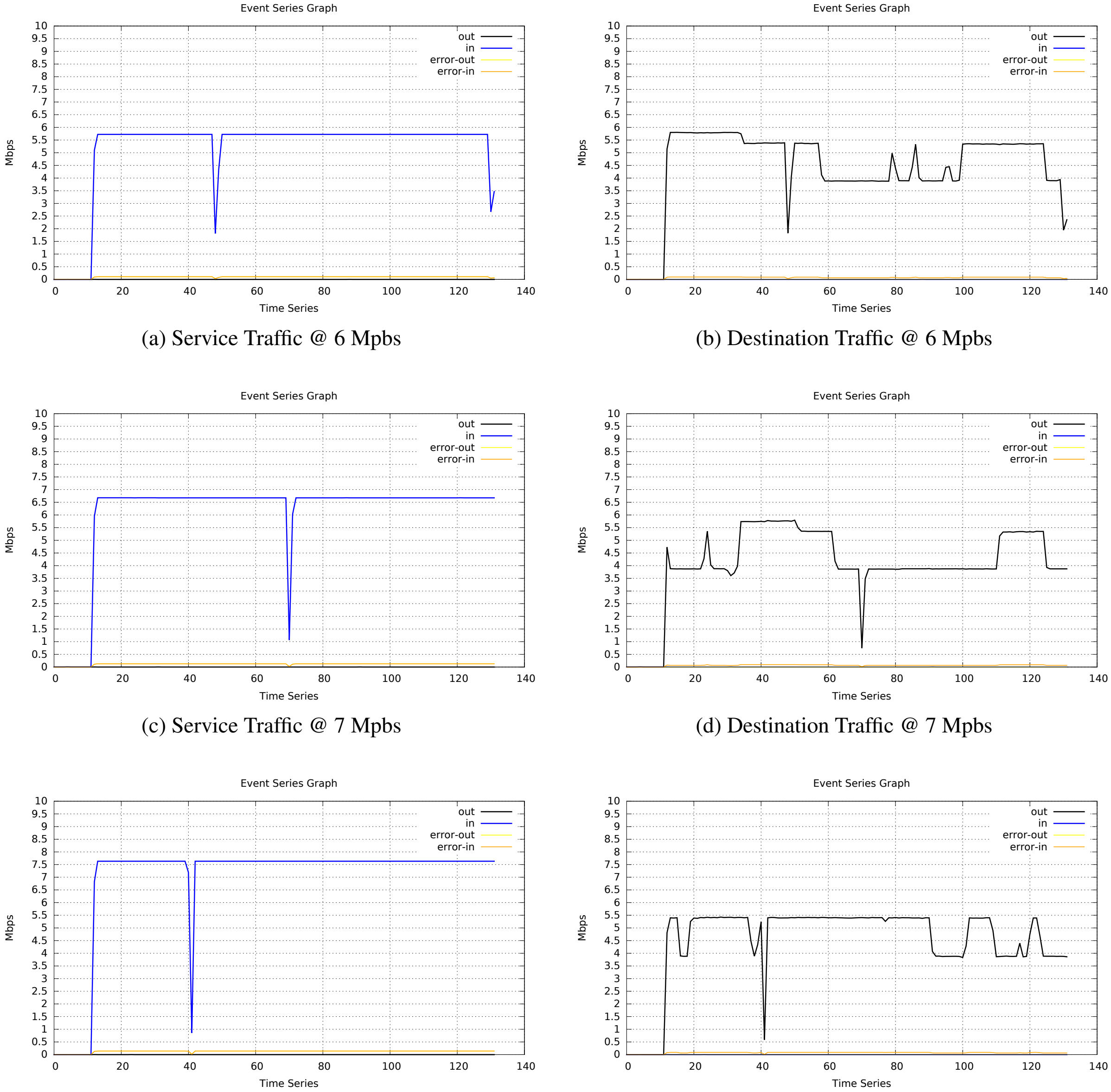
Figure 6.9: (i) CPQD - Performance Results

6.8.2 Test and Result

The experiment is performed in the testbed shown in Figure 6.2 marked as Local Site. The DEIDtect Local Site's Enterprise Network is configured with switch S3's port 2 as its gateway port, which is connected to the Cloud environment of DEIDtect.

The Enterprise network is emulated using Mininet. The experiment creates a minimal number of hosts and switches to demonstrate the framework's capability.

UDP traffic is sent from Host 1 (H1) to Host 3 (H3), and a monitoring tap is installed at switch S1's output port 3 after 20s of the traffic generation. This creates tap traffic to be sent from switch



(a) Service Traffic @ 6 Mpbs

(b) Destination Traffic @ 6 Mpbs

(c) Service Traffic @ 7 Mpbs

(d) Destination Traffic @ 7 Mpbs

(e) Service Traffic @ 8 Mpbs

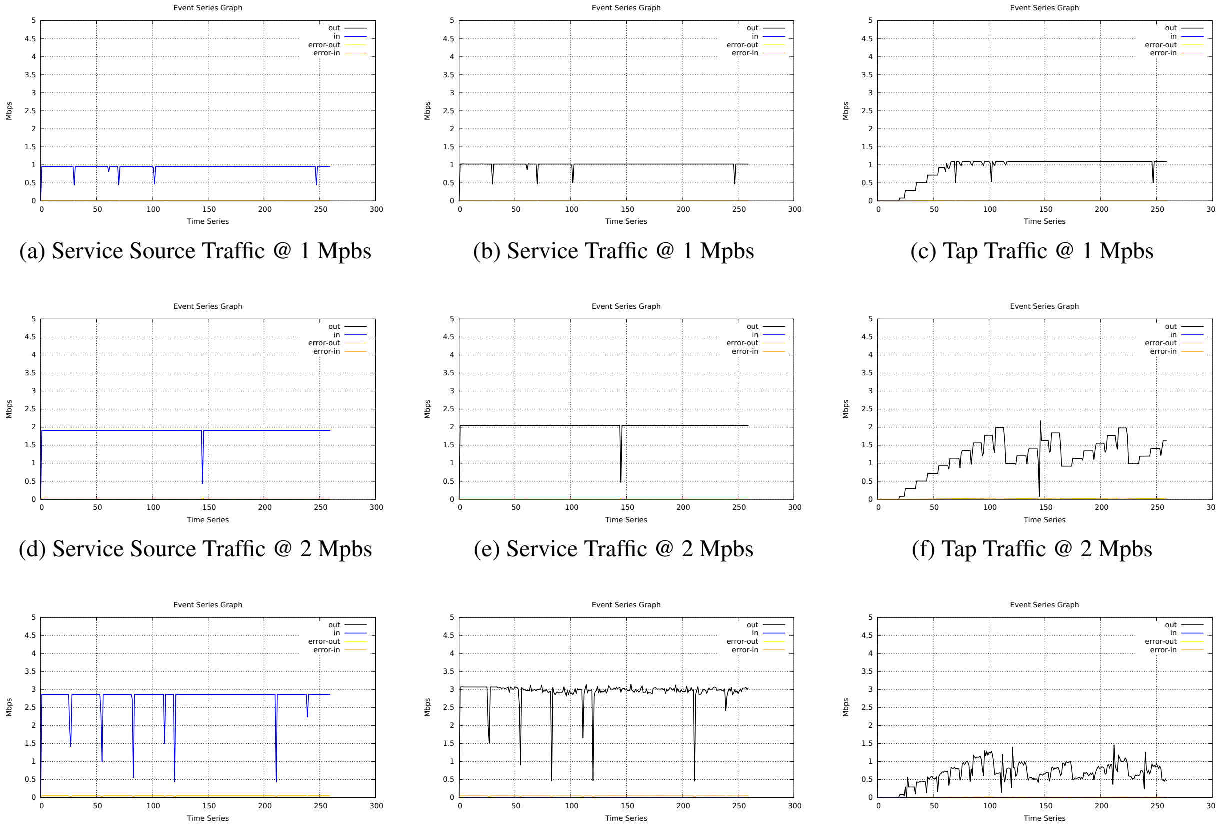
(f) Destination Traffic @ 8 Mpbs

Figure 6.10: (ii) CPQD - Performance Results

S1's output port 3 towards the gateway. This scenario creates both service traffic and tap traffic in the switch S1 port 3. The tap traffic has to be regulated by the ALB rate-limiting feature of DEIDtect to reduce or increase depending upon the increase and decrease of service traffic.

Figure 6.11 shows the source service traffic graph in the first column, and destination service traffic graph delivered at the destination Host H3 in the second column of the graph, and the corresponding tap traffic graph delivered at the gateway port, which is switch S3 port 2, in the last column. The traffic rates used to evaluate are 1, 2, and 3 Mbps of service traffic sent from H1 to H3.

As we can see the tap traffic increases slowly when there is enough bandwidth to accomodate



(a) Service Source Traffic @ 1 Mpbs

(b) Service Traffic @ 1 Mpbs

(c) Tap Traffic @ 1 Mpbs

(d) Service Source Traffic @ 2 Mpbs

(e) Service Traffic @ 2 Mpbs

(f) Tap Traffic @ 2 Mpbs

(g) Service Source Traffic @ 3 Mpbs

(h) Service Traffic @ 3 Mpbs

(i) Tap Traffic @ 3 Mpbs

Figure 6.11: DEIDtect ALB - rate limiting

both service and tap traffic, and decreases rapidly in case of service traffic disruption for each of the traffic rates. This shows that DEIDtect regulates the tap traffic as designed to keep the service traffic disruption as minimal as possible.

The tap traffic graph shows a number of spiked drops. These are the points where the rate-limit is being updated. The rate-limit update to the metering table causes this drop, which is a function of the switch implementation.

6.9 DEIDtect granularity of tap

This evaluation shows that the monitoring tap can be created for specific type of traffic in DEIDtect against state-of-the-art systems that capture all traffic being monitored.

6.9.1 Test and Result

The experiment is performed in the testbed shown in Figure 6.2 marked as Local Site. The DEIDtect Local Site's Enterprise Network configured with switch S3's port 2 as its gateway port, which is connected to the Cloud environment of DEIDtect.

TCP traffic is sent from Host 1 (H1) to Host 3 (H3) @ 2 Mbps throughout the experiment, TCP traffic is sent from Host 2 (H2) to H3 @ 8 Mbps for 120 seconds. A monitoring tap is installed at switch S2's output port 2 at 20^{th} second of the traffic generation.

Figure 6.12 shows the tap traffic in case of **TCP** filtering. From the graph it can be seen that once the TCP source stops sending traffic at 120s, the tap traffic also stops, hence illustrating that the monitoring tap is only for TCP traffic. This shows how the ALB Rate-Limiting is applied on the tap traffic.

Figure 6.13 shows the tap traffic in case of **UDP** filtering. As the UDP traffic is sent throughout, the tap traffic is seen throughout, even after the TCP source has stopped, illustrating that the monitoring tap is only for UDP traffic. As before, this shows how the ALB Rate-Limiting is applied on the tap traffic.

The results show the granularity of the DEIDtect framework.

6.10 Whitelisting for Tap Traffic

This evaluation is to show the effect of ALB-Whitelisting on the tap traffic, and to show how it helps deliver traffic of interest to the IDS against traffic which is not necessary for the IDS which is classified as normal traffic.

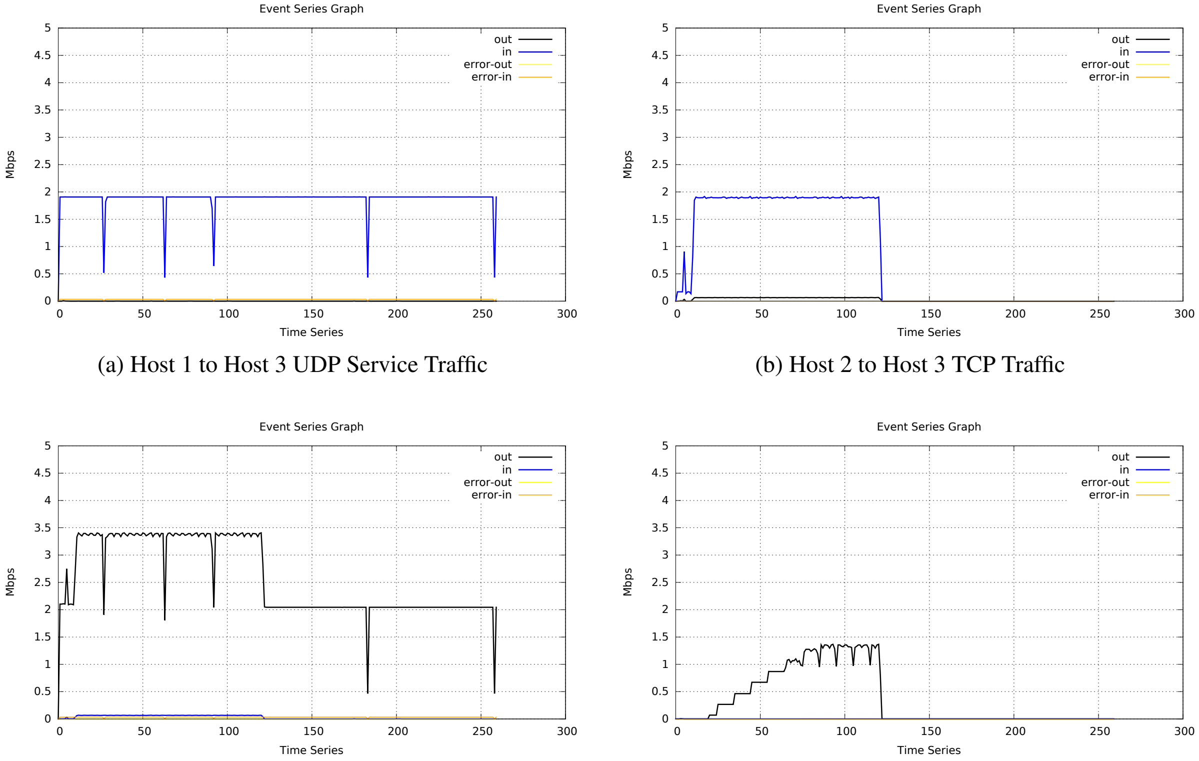
6.10.1 Test and Result

The experiment is performed in the testbed shown in Figure 6.2 marked as Local Site. The monitoring tap is created in switch S2 to monitor traffic towards port 2 in Figure 6.2 in the local enterprise site. In this experiment the IDS VM has already been created to detect the Whitelisting traffic.

Figure 6.14 shows TCP traffic is sent from H1 to H3 @ 2 Mbps throughout the experiment, and at "event 1" an FTP get request is sent from H1 to H3. TCP traffic is sent from H2 to H3 @ 2 Mbps for 60 seconds of the experiment. Host H3 hosts a secure FTP server. The monitoring point is created at 20 seconds to tunnel the traffic towards the existing IDS VM.

The IDS VM issues a Whitelisting request in the event of a successful file transfer. Once this request is sent to DEIDtect Core, it is relayed to the corresponding Enterprise DEIDtect Network System and pushes the corresponding drop rules of the specified traffic.

Figure 6.14 shows the traffic delivery at each of the monitoring ports. Figure 6.14 (d) shows the whitelist window for traffic from H1 to H3 because of the "event 1." Hence the traffic between H1 and H3 is not seen in the gateway. This is illustrated, as the H2 traffic generation has been stopped at 60 seconds. The whitelist drops the traffic for 60 seconds as per the current implementation, and after the "whitelist window," the traffic is resumed, as seen in the graph after 135 seconds.



(a) Host 1 to Host 3 UDP Service Traffic

(b) Host 2 to Host 3 TCP Traffic

(c) Host 3 : Service Traffic Delivery

(d) Tap Traffic - TCP filter

Figure 6.12: DEIDtect - TCP Tap

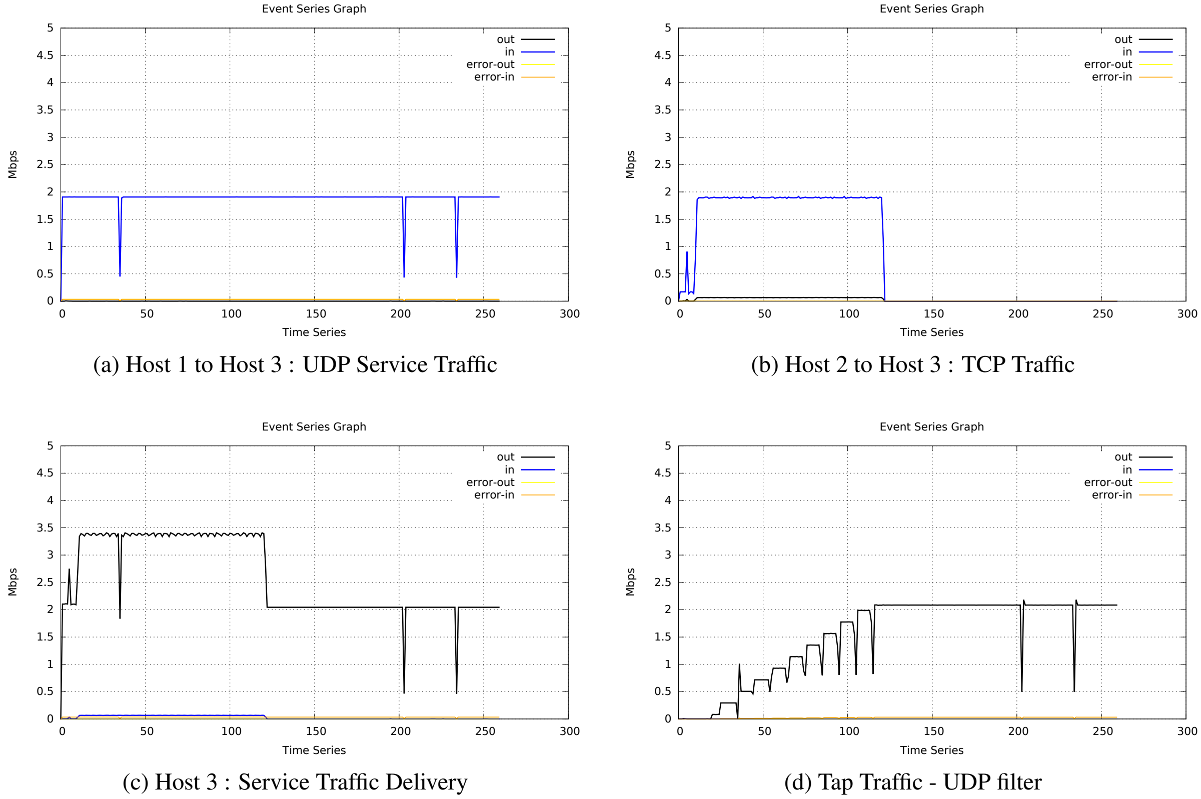
Thus ALB-Whitelisting plays an important role in regulating the traffic towards the IDS in a smart way in cases where there are many monitoring points which share the same link giving way for the tap traffic which the IDS wants to analyze.

6.11 DEIDtect ALB - IDS scaling

In this section we evaluate ALB - IDS scaling framework. Scaling happens when the current configuration of the IDS is not able to handle the load which is directly proportional to the CPU load on that VM. In such a scenario DEIDtect is able to scale the IDS VM to handle the load.

We will evaluate the scaling of IDS based upon the IDS VM usage. Depending on the amount of traffic being diverted to the IDS, the CPU usage will vary, and this is used to evaluate how the scaling occurs in these cases.

As mentioned before, the network topology information is not available in the current OpenStack version. Hence the flow distribution using network topology placement to make the necessary cluster arrangement as in Bro Cluster Topology Setup [4] is not possible. For the current implementation DEIDtect uses Host-based IDS load balancing for Bro cluster mode to handle more traffic than in standalone mode.



(a) Host 1 to Host 3 : UDP Service Traffic

(b) Host 2 to Host 3 : TCP Traffic

(c) Host 3 : Service Traffic Delivery

(d) Tap Traffic - UDP filter

Figure 6.13: DEIDtect - UDP Tap

This is done using Linux **PF_RING**, which is a means of distributing packets from the NIC to multiple applications simultaneously. By having multiple workers in each thread in a multicore system, a Bro Host-based cluster can be configured to handle high rate of traffic.

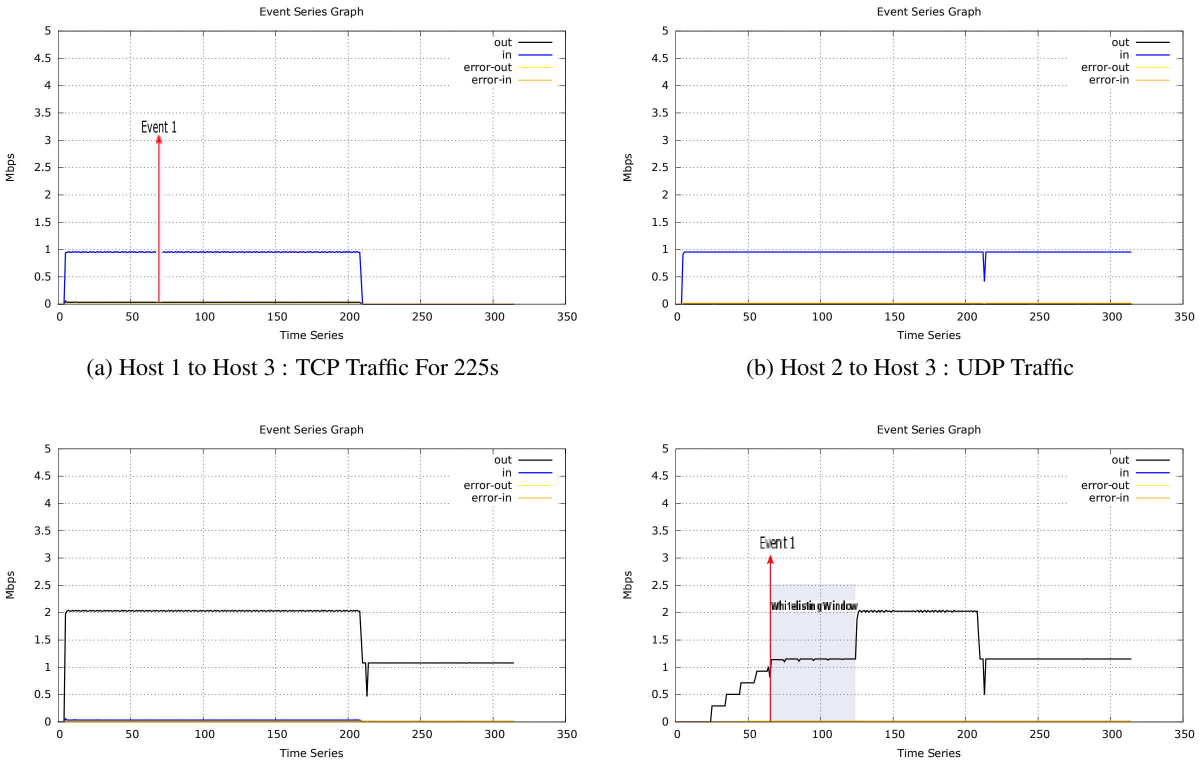
We will only evaluate the CPU usage, as the packets processing has already been studied for the cluster-based setup by Weaver in Stress Testing Cluster Bro [31].

6.11.1 Metrics

The metrics analyzed is the IDS VM CPU usage between *Standalone and Cluster* mode.

6.11.2 Test and Result

The IDS VM created by DEIDtect will be reporting the CPU usage periodically to the DEIDtect framework. Based upon this usage, DEIDtect will request the DEIDtect Cloud System to scale the IDS. But with the current limitation of the unavailability of the physical network topology information, DEIDtect requests the IDS itself to move from standalone to cluster mode. The cluster mode will be running a manager, a proxy and four workers pinned to each of the cores of the IDS VM. The current IDS VM flavor is four VCPUs, 8GB RAM, and 80GB of hard disk size.



(a) Host 1 to Host 3 : TCP Traffic For 225s

(b) Host 2 to Host 3 : UDP Traffic

(c) Host 3 : Service Traffic Delivery

(d) Tap Traffic - whitelist H1 to H3

Figure 6.14: DEIDtect - ALB Whitelist

Figure 6.15 shows the CPU usage over the span of the experiment. With the mininet environment, we could not saturate the link to the IDS VM, hence for testing we simulated an event trigger where the VM is made to report a higher CPU usage against the actual CPU. Figure 6.15 “Event 1” shows the transition from standalone to cluster mode, where all the cores become active to handle more traffic load. The packet handling is not studied for the same reason mentioned above, but this has been studied and published by Weaver in Stress Testing Cluster Bro [31].

From the results we can see that DEIDtect ALB - IDS scaling works as claimed.

6.12 Data Loss in IDS

DEIDtect regulates the tap traffic by Adaptive Load Balancing which comprises two ways of limiting. Rate-limiting is a blind rate limit of packets based upon the service traffic rate at each monitoring port. These packets are dropped by the switch based upon the firmware implementation. DEIDtect has no control of how these are dropped, while Whitelisting drops flows which are classified as normal traffic by the IDS.

In this section, we analyze the effect of packet loss of the tap traffic being made by the rate-limiter and how ALB-Whitelisting plays a role in such scenarios.

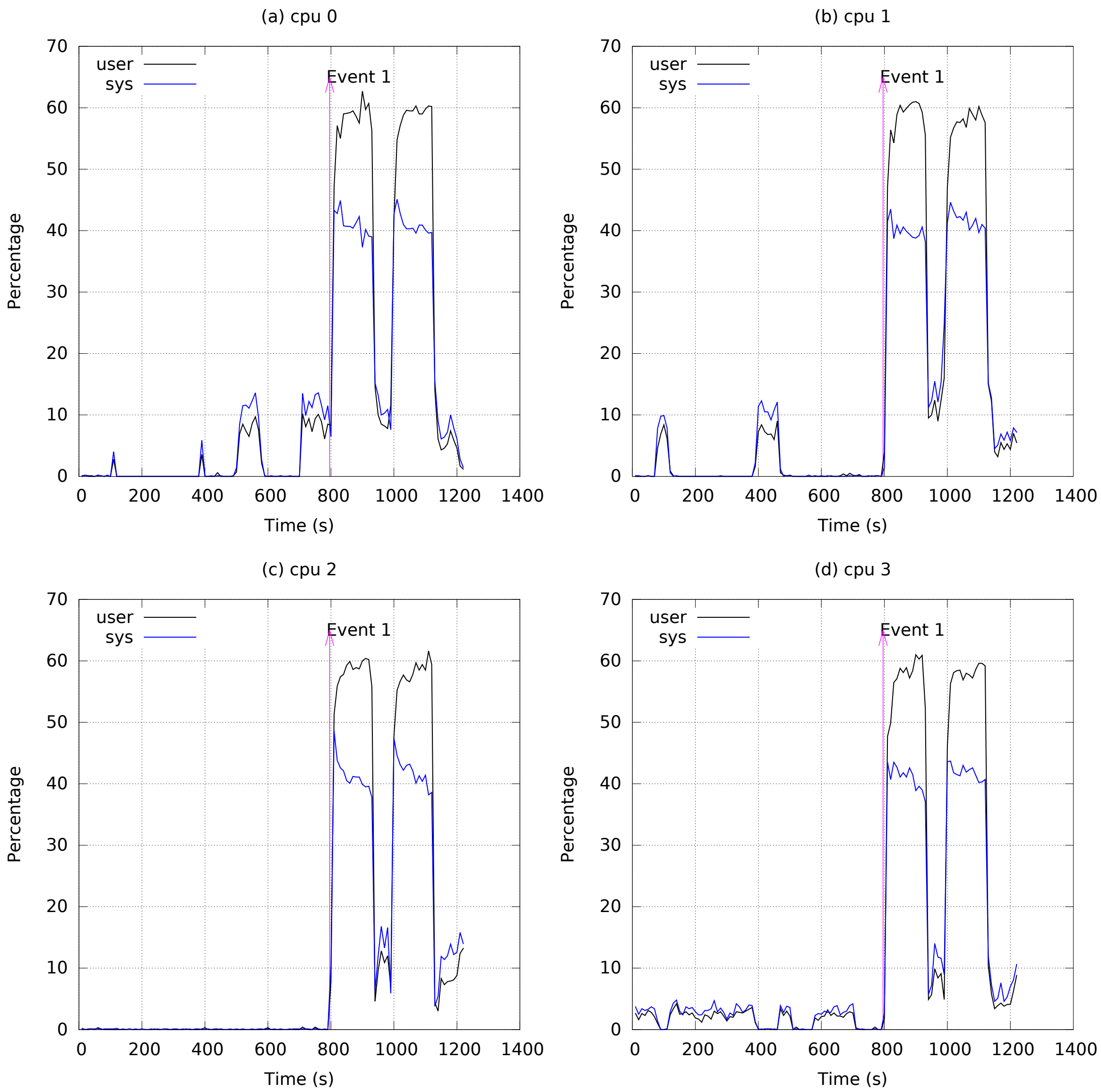


Figure 6.15: DEIDtect IDS Scaling Cpu Usage

The experiment is performed in the testbed shown in Figure 6.16 marked as Local Site. The PCAP file should be real-time enterprise traffic. Since http://digitalcorpora.org packet traces offer such traces over weeks, we will be using once such small trace for our experiment.

6.12.1 Test and Result

Figure 6.16 comprises labeled points (a) TCP traffic from H1 to H3 @ 2 Mbps, (b) PCAP replay traffic from http://digitalcorpora.org/corpora/scenarios/2009-m57-patents/net/net-2009-11-14-09:24.pcap.gz sent towards H3 @ 1 Mbps, (c) Destination port and the Tap port in a 4 Mbps link, and (d) Tap traffic for the corresponding Tap (Max Tap traffic rate configured @ 2 Mbps).

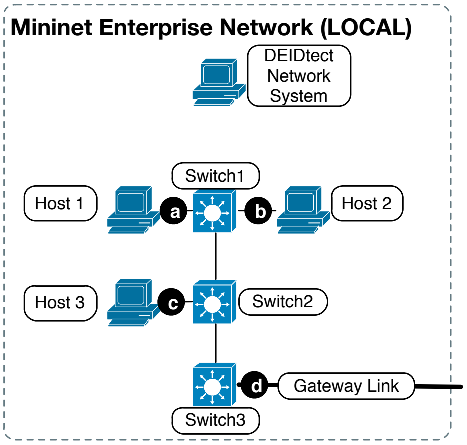


Figure 6.16: DEIDtect Tap Traffic Packet Loss Topology

All the traffic activity graphs shown will show the traffic activity at each of the labeled points. The traffic activity graph has event labels in it. Event: (1) TCP traffic sent from H1 to H3, (2) TAP installed in Switch 2 port 2 Figure 6.16 (c), and (3) PCAP replay traffic from H2 to H3. The label WhiteStart and WhiteEnd shows the time-line event where the TCP traffic from H1 to H3 is Whitelisted by DEIDtect.

The final packet capture is done only for traffic other than between H1 to H3, which is the PCAP traffic being replayed. The trace-summary given based upon the packet capture at the IDS is obtained to analyze the effect of packet loss in IDS detection. The PCAP traffic source is started well after DEIDtect ALB-Rate-limiting has warmed up to its maximum rate-limiting state.

6.12.1.1 Ground Truth Detection Results

For the ground truth about the detection, we capture the PCAP traffic without data loss in the IDS for the same topology. Figure 6.17 shows a number of time-series events corresponding to IDS detection: (1) No TCP traffic is sent, (2) Create Tap point on S2 port 2, (3) play PCAP file. The resulting traffic activity is shown in Figure 6.17. The IDS host was able to receive 628 packets from the PCAP trace.

Without any data loss to the tap traffic, the respective trace summary which consists of number of source and destination IP detected, is shown in Table 6.2. This serves as the baseline for the evaluation of DEIDtect with and without Whitelisting in terms of detection on account of traffic loss.

6.12.1.2 DEIDtect Nonwhitelist - Detection Results

Without any Whitelisting for the same experimental setup, IDS captures the traffic from H2 alone which is the PCAP replay traffic. Figure 6.18 shows a number of time-series events corresponding to IDS detection: (1) Send UDP traffic from H1 to H3 @ 2 Mbps, (2) Create Tap point on S2 port 2,

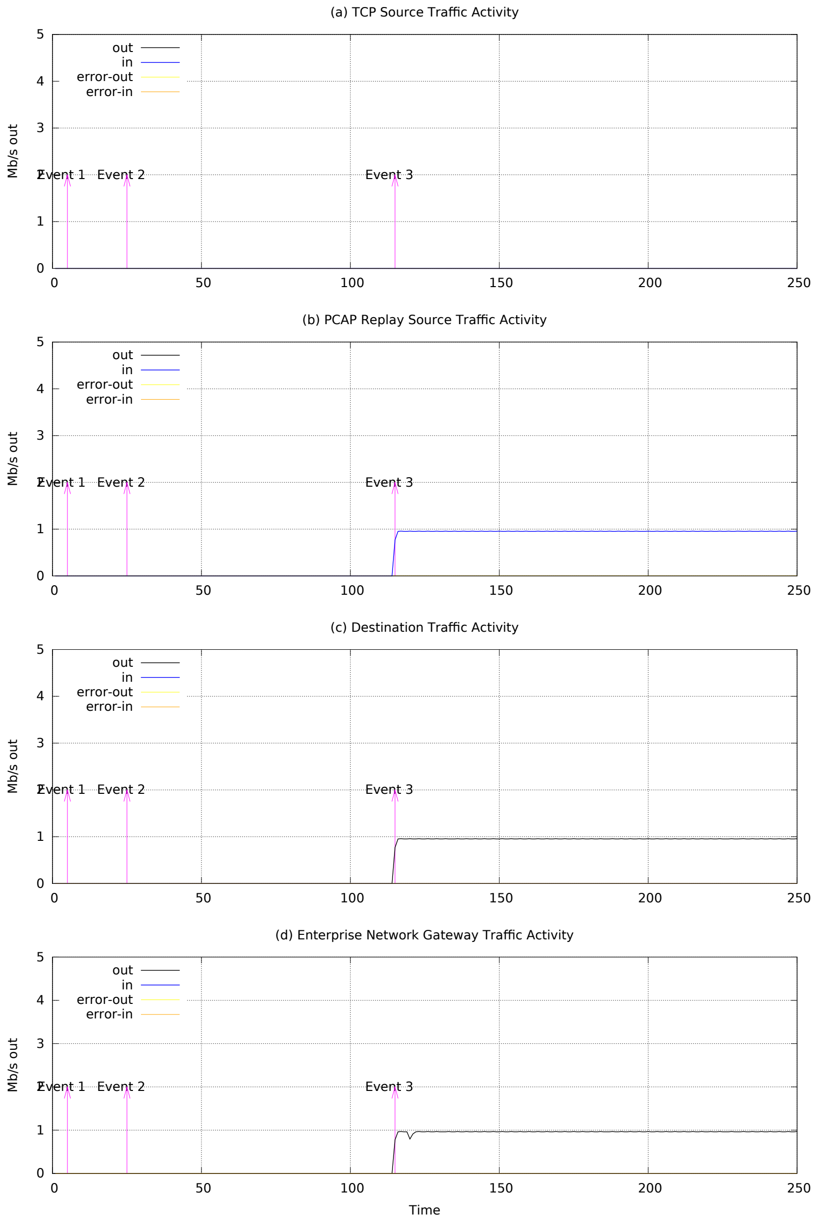


Figure 6.17: Ground Truth - IDS Detection Traffic Graph

(3) play PCAP file. The resulting traffic activity is shown in Figure 6.18. The IDS host was able to receive 446 packets from the PCAP trace.

The graph shows that once all the traffic sources start sending traffic, a total of 3 Mbps of traffic is being sent to H3, but the Tap traffic is capped at 2 Mbps from the 4 Mbps link, hence 1 Mbps of traffic is lost. This loss causes the PCAP replay traffic packet capture in the IDS to be less accurate. As shown in Table 6.3, IDS VM was able to detect only a few connections from the PCAP trace.

Table 6.2: Ground Truth Result -Trace Summary

S.No	Source Detected	Destination Detected
1.	192.168.1.103	192.168.1.255
2.	192.168.1.102	192.168.1.1
3.	192.168.1.104	207.46.232.182
4.	192.168.1.150	192.101.21.1
5.		192.43.244.18
6.		4.2.2.4
7.		4.2.2.3
8.		255.255.255.255

6.12.1.3 DEIDtect Whitelist - Detection Results

For the same experimental setup, in the Whitelisting period which drops traffic from H1 to H3 to be discarded in the tap, traffic is shown in Figure 6.19. The IDS host was able to receive 585 packets from the PCAP trace.

This results in a better detection, as shown in Table 6.4. Though this shows all the detection to be the same as the ground truth results, this summary is obtained from the smaller number of packets compared against the "Ground Truth," which may lead to detection degradation if any packet loss is incurred.

This evaluation shows that ALB-Whitelisting helps in terms of IDS detection accuracy.

6.13 Summary of Results

A summary of our results is presented below.

- We show that DEIDtect is able to monitor traffic at any point of an enterprise network with great ease in "local" and "remote" sites.
- DEIDtect is able to detect anamolies with the tunnel traffic approach with IDS in the cloud.
- DEIDtect is able to dynamically manage the Tap traffic rate with minimal disruption to service traffic.
- DEIDtect is able to Tap specific type of traffic, giving fine-grained control of what is being monitored.
- DEIDtect is able to drop packets in a smart way using Whitelisting.
- The combined effect of rate-limiting and Whitelisting delivers packets of interest to the IDS, which helps detect anamolies.
- DEIDtect is able to dynamically scale up/down based upon the load of the IDS in cloud.

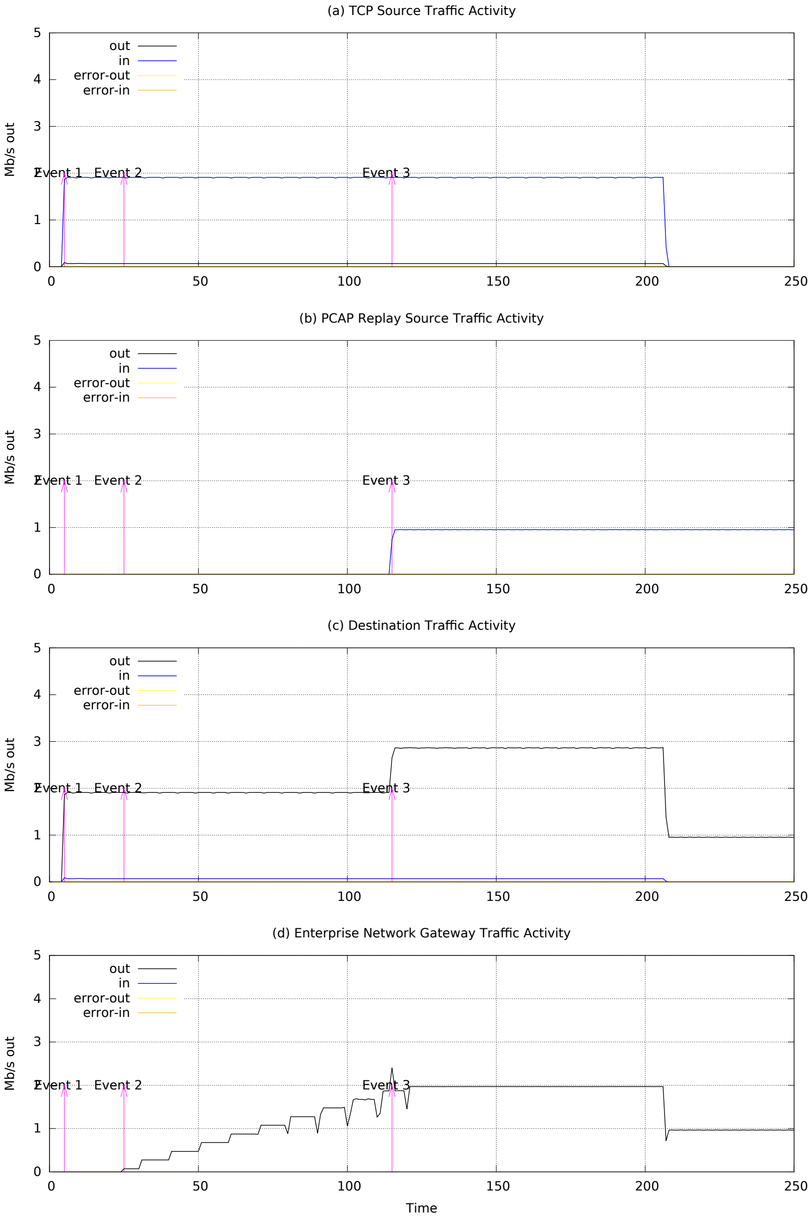


Figure 6.18: DEIDtect Nonwhitelist - IDS Detection Traffic Graph

Table 6.3: Nonwhitelist Result - Trace Summary

S.No	Source Detected	Destination Detected
1.	192.168.1.103	192.168.1.255
2.	192.168.1.102	255.255.255.255
3.	192.168.1.104	

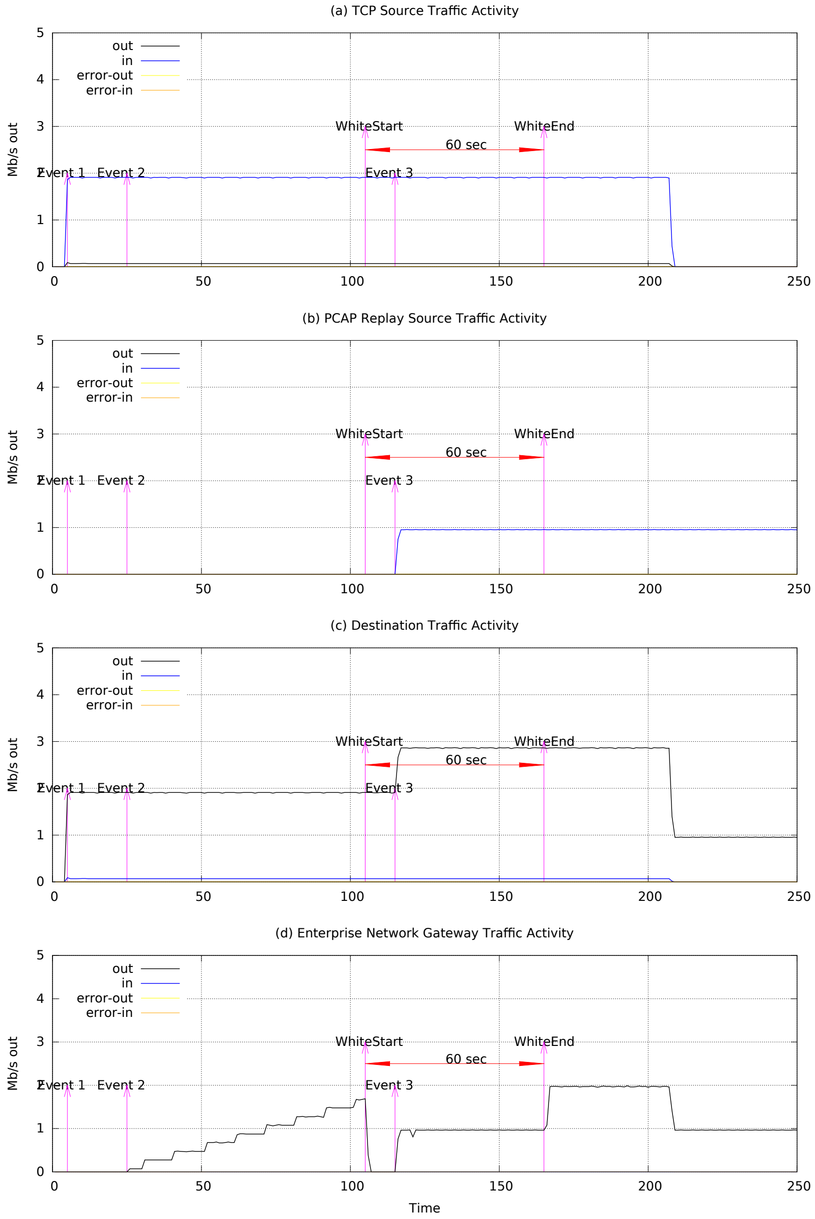


Figure 6.19: DEIDtect Whitelist - IDS Detection Traffic Graph

Table 6.4: Whitelist Result - Trace Summary

S.No	Source Detected	Destination Detected
1.	192.168.1.103	192.168.1.255
2.	192.168.1.102	192.168.1.1
3.	192.168.1.104	207.46.232.182
4.	192.168.1.150	192.101.21.1
5.		192.43.244.18
6.		4.2.2.4
7.		4.2.2.3
8.		255.255.255.255

CHAPTER 7

PRACTICAL CHALLENGES AND FUTURE WORK

7.1 Challenges

This section talks about the challenges towards the deployment of DEIDtect in the real world, and potential future work.

7.1.1 Enterprise Network Features

The DEIDtect framework relies on OF's multiple-flow table support for the safe tapping of the traffic. Multiple-flow tables are being supported from OF 1.1 and up. This gives the flexibility to extend the packet-processing pipeline to multiple levels, as opposed to the single-table support in OpenFlow version 1.0. Multiple-flow tables give the flexibility to have fine-grained control of traffic of interest at each level of the pipeline. Figure 2.4 shows the pipeline for two table. The flexibility of having multiple levels of match on the pipeline is a critical requirement for DEIDtect.

We investigated an alternative way to replicate safe tapping without using multiple tables. The group table which is supported in all OF versions was a viable solution. The group tables give the flexibility to take multiple actions for a set of matching flows. Figure 7.1 shows this functionality for one flow having two actions. The limitation of this approach is that it does not provide the granularity of the multiple-flow table feature.

In Figure 2.4, if we want to have only TCP flows sent out of port 3, the corresponding pipeline match will have the TCP match in the IDS flow table. But this is not possible in the group table approach. All the actions operate on one copy of the packet. Despite this limitation, it is possible to realize fine-grained safe tapping with the group table by having fine-grained flow differentiation in the next-hop switch.

A major OF 1.0 restriction was that metering features are supported from OF 1.1 onwards, which is crucial for ALB's functionality. Hence DEIDtect requires OF 1.1+ as the minimum requirement for deployment.

For DEIDtect the switch has to have the following capabilities:

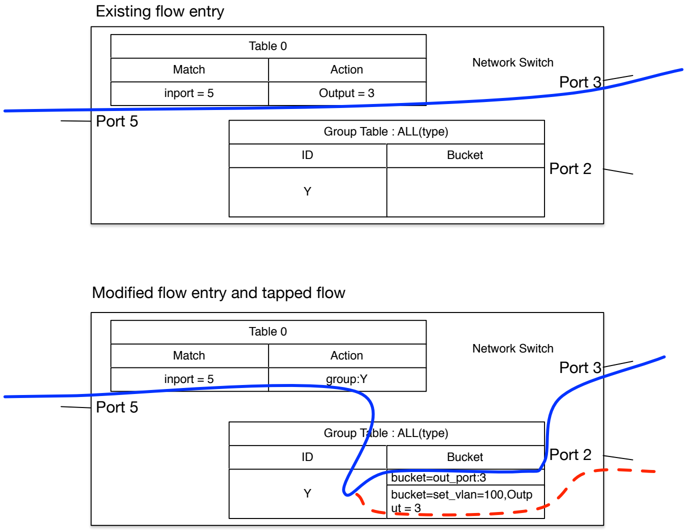


Figure 7.1: Safe Tapping using Group Tables

1. Hardware Multi-Table support - if there is only software multi-table support we might as well use software switching.

2. Meter-Table supporting Drop band type - for rate limiting of traffic.

3. Flow Table with VLAN PUSH,VLAN MOD and VLAN POP actions are crucial for isolation enforcement.

4. Flow Table Match flexibility is required to allow targeting of specific tap traffic. As such match fields with few constrains on usage is desirable.

7.1.2 Cloud Network Access

In a cloud environment there are many ways of deployment in different network topology settings based upon requirements and demand. This is decided by the cloud administrators. Some cloud softwares capable of managing the network devices, and some assume connectivity between nodes, as in OpenStack [26].

The cloud-deployment software provides a rich programming interface to interact with the cloud to enable third-party application development. But many of the core administration tasks are kept isolated from external access. DEIDtect requires flow installation capability in an SDN-enabled cloud environment to manage the tap traffic rate in the cloud network, as well. This is possible only when we have access to the underlying network devices. And the firewall settings should be accessed by DEIDtect, as the IDS requires all types of traffic to be delivered to it, which is most likely to set off the firewall in the cloud to kick in, which is an undesirable effect. DEIDtect also requires some of the configurations to be pushed to the VM, which are to be created for IDS for the Whitelisting and dynamic scaling. The Cloud software is required to have dynamic data injection or configuration-management support like Cloud-init [2] or Chef [3].

7.1.3 Inter-Domain Access

The DEIDtect framework has different domains to make the whole orchestration happen, and hence requires access to the domains by some means to control the Enterprise and the Cloud environment, as described in Section 2.

7.2 Future Work

The DEIDtect framework provides the ability to deploy a monitoring point in an enterprise or a campus network, and functions across domains in a distributed way which opens up new use cases, as described in Chapter 3.

We want to provide a way to make use of this framework to write applications on top of it to make use of the interdomain distributed capability, which has never been deployed before to create new opportunities for the security administrators.

With the knowledge about the physical network topology, the tapping can be optimized to reduce the duplicate traffic within the network.

With DEIDtect we have more information about the traffic such as the switch in which the traffic is originating along, with the information from the IDS detection results. This can be helpful for the security administrator to identify the problem quickly.

CHAPTER 8

CONCLUSION

This thesis aims to address the challenge of ease and flexibility and scalability to the Network Intrusion Detection deployment. We presented our work on DEIDtect architecture which exploits increased cloud- and software-defined networking deployments to realize an elastic distributed intrusion detection framework. DEIDtect effectively decouples the location of a network being protected from the location of the security tools performing security functions. This flexibility enables DEIDtect to realize new distributed security functions between partnered organizations. We presented the detailed design and implementation of DEIDtect and illustrated its functionality using emulated enterprise environment and OpenStack cloud environment. To realize the full potential of DEIDtect, our future work includes developing security applications that can exploit the unique cross-domain functionality of DEIDtect.

REFERENCES

[1] www.snort.org.

[2] https://cloudinit.readthedocs.org/en/latest/.

[3] https://www.chef.io/solutions/configuration-management/.

[4] Bro cluster setup. https://www.bro.org/sphinx-git/cluster/index.html.

[5] Dshield - Internet Storm Center. http://www.dshield.org/howto.html.

[6] Scipass - secure openflow based sciencedmz. http://globalnoc.iu.edu/sdn/scipass.html.

[7] AZODOLMOLKY, S., WIEDER, P., AND YAHYAPOUR, R. Cloud computing networking: challenges and opportunities for innovations. *IEEE Communications Magazine 51*, 7 (July 2013), 54–62.

[8] BAUCKE, S., MESTERY, K., SHAIKH, A., AND WRIGHT, C. Opendaylight: An open source sdn for your openstack cloud. *An Open-Stack Summit, Hong Kong* (2013).

[9] CHIU, D.-M., AND JAIN, R. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst. 17*, 1 (June 1989), 1–14.

[10] CPQD. OpenFlow 1.3 Software Switch. http://cpqd.github.io/ofsoftswitch13.

[11] GRANCE, M. The dids (distributed intrusion detection system) prototype. In *Proceedings of the Summer USENIX Conference*, pp. 227–233.

[12] GROVES, R., AND BENETTI, B. Microsofts demon-datacenter scale distributed ethernet monitoring appliance. *Presented during Sharkfest, Berkeley, CA* (June 2012).

[13] GUOK, C., LAKE, A., KRZYWANIA, R., AND BALKCERKIEWICZ, M. Inter-domain controller (idc) protocol specification.

[14] HEORHIADI, V., REITER, M. K., AND SEKAR, V. New opportunities for load balancing in network-wide intrusion detection systems. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 361–372.

[15] HIBLER, M., RICCI, R., STOLLER, L., DUERIG, J., GURUPRASAD, S., STACK, T., WEBB, K., AND LEPREAU, J. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference* (2008), pp. 113–128.

[16] HOUIDI, I., MECHTRI, M., LOUATI, W., AND ZEGHLACHE, D. Cloud service delivery across multiple cloud platforms. In *Services Computing (SCC), 2011 IEEE International Conference on* (2011), IEEE, pp. 741–742.

[17] Janakiraman, R., Waldvogel, M., and Zhang, Q. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on* (2003), IEEE, pp. 226–231.

[18] Kissel, E., Fernandes, G., Jaffee, M., Swany, M., and Zhang, M. Driving software defined networks with xsp. In *Communications (ICC), 2012 IEEE International Conference on* (2012), IEEE, pp. 6616–6621.

[19] Lantz, B., Heller, B., and McKeown, N. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 19:1–19:6.

[20] Lehigh, K., and Khalfan, A. Multi-Gigabit Intrusion Detection with OpenFlow and Commodity Clusters. www.openflowhub.org/download/attachments/3244813/SPC-Present.pdf.

[21] Lo, C.-C., Huang, C.-C., and Ku, J. A cooperative intrusion detection system framework for cloud computing networks. In *Parallel processing workshops (ICPPW), 2010 39th international conference on* (2010), IEEE, pp. 280–284.

[22] Matias, J., Jacob, E., Sanchez, D., and Demchenko, Y. An openflow based network virtualization framework for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 672–678.

[23] Mazzariello, C., Bifulco, R., and Canonico, R. Integrating a network ids into an open source cloud computing environment. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on* (2010), IEEE, pp. 265–270.

[24] Mechtri, M., Zeghlache, D., Zekri, E., and Marshall, I. Inter-cloud networking gateway architecture. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on* (Dec 2013), vol. 2, pp. 188–194.

[25] Messmer, E. Start-up morphs open-source security system for research networks into commercial platform. http://www.networkworld.com/news/2013/071613-broala-271856.html.

[26] Openstack. Openstack - Open source software for creating private and public clouds. https://www.openstack.org/.

[27] Shanmugam, P. K., Subramanyam, N. D., Breen, J., Roach, C., and der Merwe, J. V. Deidtect: Towards distributed elastic intrusion detection. In *Proceedings of the ACM SIGCOMM Workshop on Distributed Cloud Computing* (Aug. 2014).

[28] Shin, S., and Gu, G. Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *Network Protocols (ICNP), 2012 20th IEEE International Conference on* (Oct 2012), pp. 1–6.

[29] Sproull, T., and Lockwood, J. Distributed instrusion prevention in active and extensible networks. In *Active Networks*. Springer, 2004, pp. 54–65.

[30] Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., and Tierney, B. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Recent Advances in Intrusion Detection* (2007), Springer, pp. 107–126.

[31] Weaver, N., and Sommer, R. Stress testing cluster bro. In *DETER* (2007).