# 软件定义的安全体系架构

An Architecture of Software-defined security

本文介绍了一个SDN环境中的软件定义安全架构，安全应用-安全控制器-安全设备纵向构成了安全防护体系，

## 一、简介

软件定义网络（SDN，Software-Defined Networking）通过软件的形式重构现有网络，颠覆了现有的网络的通信和管理模式，近年来在业界和学术界得到越来越多的关注。SDN的核心是将控制平面与数据平面分离，如图1所示，网络控制器平台（SDN Network Controller Platform）南向控制网络设备，北向与网络应用交互，从而设备的数据流向与应用的策略实现了解耦和。在网络设备日趋复杂、管理难以维系的数据中心、公共云服务和大型企业的网络中，大大简化了网络管理和数据交换机制，缩短了路由收敛时间，在可预计的未来很可能会改变整个数据中心、大型企业甚至骨干网的网络布局。

服务管理

网络控制器平台

APP

APP

APP

APP

网络设备

网络设备

网络设备

控制平面

数据平面

图1 SDN架构图

Figure1 SDN architecture

虽然软件定义网络和网络功能虚拟化（NFV，Network Function Virtualization）有诸多优点，但其使得网络边界模糊化、虚拟化，造成很多传统的安全产品无法适用。Garnter公司认为安全厂商也应随SDN设计新的安全解决方案，提出了三个阶段的SDN安全，并将第三个阶段称为软件定义安全（SDSec，Software-Defined Security）[2]，其要点在于安全管理平面与安全数据平面分离，并通过安全控制器（Security Controller）保护整个SDN环境，但仅限于想法，并没有给出细节或实现。

在本文中，我们设计了一个安全控制器，可与SDN环境中的各网络实体进行交互，保证数据交互的正确性；也可与各类安全产品协同，共同保护整个环境的安全。

本文主要分为四章，首先在第一章主要简要介绍SDN和SDSec，然后第二章介绍相关工作，其次在第三章主要介绍了安全控制器的机制和相关模块，随之在第四章实验部分验证了安全控制器的效率，最后在第五章给出了结论。

## 二、相关工作

左青云等人在[1]中详细介绍了基于Openflow的SDN技术的背景和架构，以及在系统高性能、一致性和可靠性方面的重要工作，但缺少安全方面挑战的介绍和分析，只提到Openflow协议的设计中安全性问题考虑不够。事实上，在SDN和虚拟化的新场景中的安全防护机制与传统的有很大差异。

一方面SDN是集中控制体系，上层的网络控制器更容易遭受拒绝服务攻击；另一方面SDN和网络虚拟化引入了新的主体，如虚拟交换机和虚拟机等，虚拟主体间的通信是不为实体安全设备可见或理解的。

SDN和虚拟化带来的挑战迫使安全防护必须理解这些变革，为了兼容当前实体安全设备，Zafar[3]等人提出的Simple方案使用SDN控制器引导网络设备，将流量引导到实体安全设备中，但是这种方案可能会造成环路，虽然作者改写openflow某些字段，但可能破坏协议一致性，造成不兼容。所以更好的办法是将安全设备虚拟化，融入到整个设施虚拟化的体系内，将以往的安全边界从网络出入口移到虚拟网络设备中。

虽然面临很多挑战，SDN也为很多传统应用的安全防护提供了新的思路，例如在防垃圾邮件、蠕虫和DDoS等，可使用SDN架构，根据OpenFlow流发现流量异常，检测是否存在攻击，并将流量快速引导。安全厂商Radware于 2013年四月发布了第一款基于SDN的抵御DdoS的产品DefenseFlow[3]，在攻击检测和流量清洗过程中使用OpenFlow协议，代替了以往的BGP+NetFlow方案，使用集中下发策略的方式大大减少路由收敛的时间。

虽然有一些SDN安全机制的探索，但总体而言当前SDN的安全研究还处于起始阶段，尚有大量的工作需要完成，特别是整个SDN安全体系的设计和建立，需要大量的工作。

## 三、软件定义安全体系设计

我们借鉴了安全控制和安全数据分离的思想，设计了一个面向设施即服务（IaaS）的软件定义安全的体系，如图2所示。与软件定义网络对应的，软件定义安全体系的核心是安全控制器（security controller），本章主要介绍了安全控制器的整体架构和主要功能模块，以及这些模块协作机制和功能实现。

如图2所示，安全控制器南向管理支持虚拟化的新型安全设备或传统的安全设备，分发安全策略和命令，实时获取和监控网络控制器和安全设备的相关数据、状态和安全告警等信息；北向与各类安全APP进行数据和策略的交互，实现安全应用的协作控制，抵御不同类型的攻击；西向与SDN的网络控制器获取拓扑和流数据等信息，并发送数据交换的命令；东向可支持多种设施虚拟化解决方案，理解云环境中的多租户场景中的租户机构、用户、虚拟机和虚拟网络等主题。

**安全控制器**

**网络控制器**

**网络设备**

交换  
命令

**安全设备 VM/Agent**

数据  
请求

状态报告

安全告警

可疑数据

交换命令

**安全控制器**

**Agent**

数据、拓扑

状态统计

数据流

**安全设备 VM/Agent**

**安全设备 VM/Agent**

安全数据流

安全  
策略

**网络APP**

**安全APP**

**安全APP**

**安全APP**

指令

安全  
事件

安全  
事件

指令

业务  
输入

业务  
输入

业务  
输出

业务  
输出

**IaaS网络虚拟化模块**

租户

网络

图2 安全控制器的交互图

### 3.1 安全控制器架构

安全控制器的内部模块组成和总体架构如图3所示，主要由若干个核心模块和面向不同场景的定制模块组成的。每个模块与控制器内部的模块或外部的安全或网络主体进行交互，生成的数据保存到缓存或数据库中，形成若干个库，如APP库、设备库、流库和策略库等。

我们对一些模块进行介绍，其中核心模块包含：

* **事件调度（Event Scheduler）** 接受各模块注册事件，将需处理的事件分发给相应模块，触发事件处理机制；
* **应用管理（APP Manager）** 管理北向的安全应用信息，接受应用的可疑数据订阅，推送满足条件的可以数据
* **设备管理（Device Manager）** 管理南向的设备应用信息，在策略解析器等模块中提供所需的安全设备
* **策略解析（Policy Resolver）** 将安全APP的抽象策略分解为网络设备或安全设备可执行的具体命令（如Openflow流）

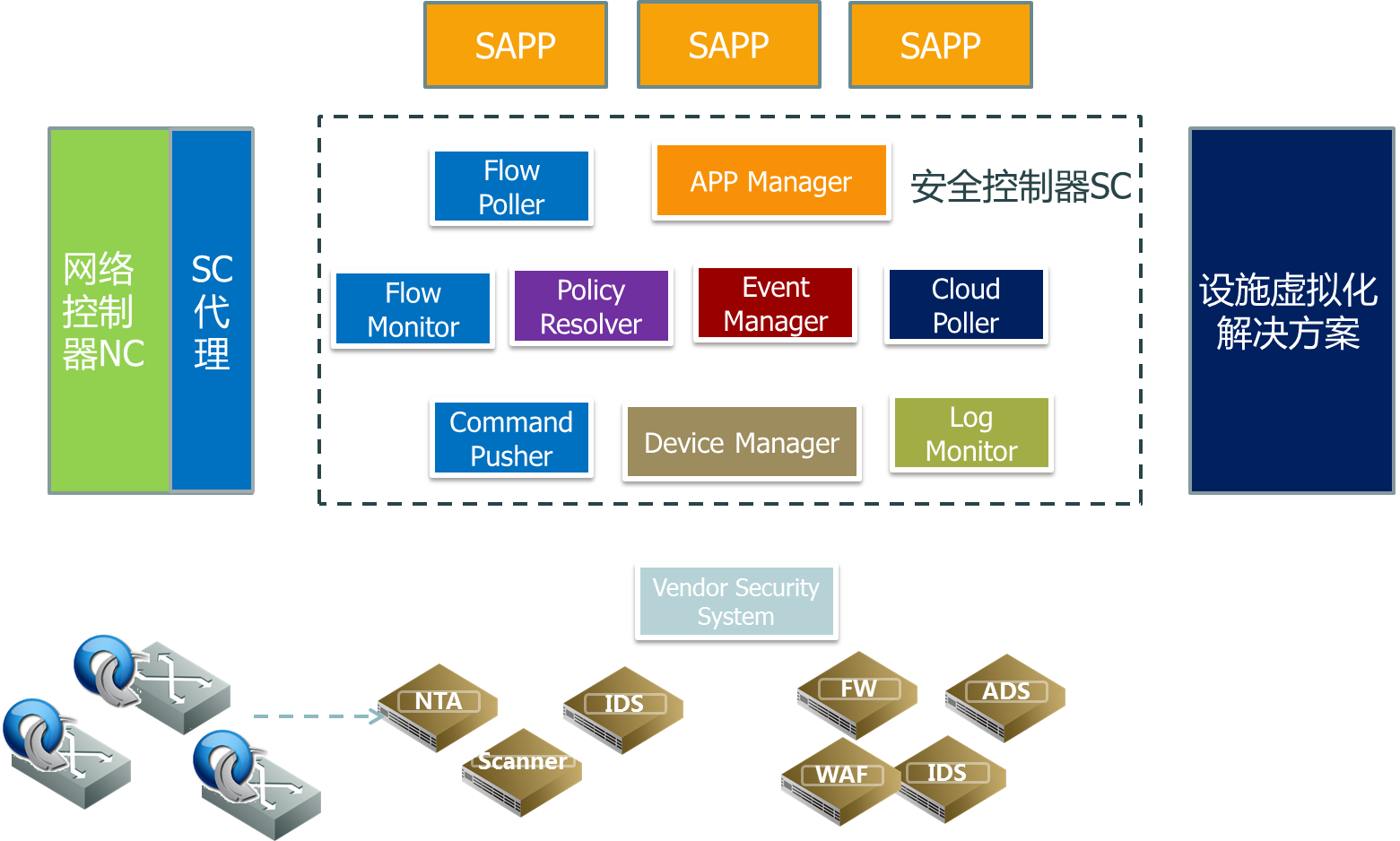


图3 安全控制器的架构

此外，还有一些重要的模块，如数据收集、可疑数据监控和命令推送，在不同场景有特定的实现，我们称之为定制模块。例如在支持OpenFlow的环境中，上述三类模块有如下实现：

* **OpenFlow流获取（Flow Polling）**从网络控制器中定时获取所有网络设备中当前的留信息
* **OpenFlow流监控（Flow Monitor）** 检测流获取模块中获得的流信息，发现可疑的数据流
* **OpenFlow流推送（Flow Pusher）** 将基于OpenFlow的命令提交给网络控制器，进而控制网络路由。

如果在其他场景（面向安全设备的应用）中，数据收集则由安全设备的数据获取模块处理，命令推送则由向安全设备下发指令的模块处理，诸如此类。

安全控制器实现的功能是有上述若干模块的协作完成的，基本的工作流为数据收集模块从东西向获得网络数据，数据监控模块根据安全应用的订阅条件寻找粗颗粒度的可疑数据，通过安全应用管理模块推送给安全应用；后者根据细粒度的算法进行决策，将命令通过安全应用管理模块下发给策略解析模块，后者根据语义解析成网络控制器或安全设备可理解的命令，最终由数据推送模块下发到控制器或安全设备。

在具体场景中，可能会存在额外的安全模块，例如日志记录和分析等，工作流也可能存在一些差异。但每个模块实现自己的功能，相对独立，这种模块的设计有以下特点：

* 各模块均提供开放、标准的WEB API，例如APP Manager和Device Manager等）都可通过RESTful的Web接口进行访问，支持GET、POST、PUT和DELETE操作，从而实现应用和设备的增加、更新和删除等功能。
* 模块之间是松耦合的，部署比较方便，管理员可以调用接口启用或禁用某些组件，安全控制器也会根据相应的应用场景，调用相应的模块（如在部署Openstack的虚拟化场景中，SC自动调用支持Openstack的东向驱动模块）；安全管理员也可以很容易根据需要编写新的模块对安全功能进行扩展。

由于整个安全系统强调的是在松耦合的系统中，提供简单的资源操作原语，使得安全管理员可以通过设计一组相关的Web调用，就能完成一系列一致性的操作，实现复杂的安全功能。

整个系统要点在于协作控制，它贯穿了安全控制器内部模块间，以及安全控制器与安全应用、设备间的交互设计。下面我们先介绍南北向的设备和应用如何与安全控制器进行交互，接着说明安全控制器内部的模块通信协作机制，然后介绍安全控制器与安全应用协作的可疑数据订阅机制，最后分析将安全应用下发的策略解析成相应命令的策略解析机制。

### 3.2 安全主体管理

在安全体系中，安全控制器处于核心地位，分别与南向的安全设备和北向的安全应用进行交互，对于这两类安全主体，安全控制器分别使用应用管理APP Manager和设备管理Device Manager进行安全主体的更新和信息维护。

TODO blabla

### 3.3 模块通信协作

#### 3.3.1事件驱动的调度机制

安全控制器的主要功能是甄别可疑数据，协调内部模块的协作和下发

合理高效的模块通信和协作机制是整个安全控制器正常运行的重要保证，事件调度器负责各模块间的数据交互和指令传递。

调度机制的原理如图4所示，主要包括3个步骤：

1. 模块A在初始化时向事件调度器注册事件类型*e*，表示A希望接收所有类型为*e*的事件，同样的，其他模块X也可以注册该类型的事件；
2. 在运行时，模块B在处理完特定流程后，产生了一个事件*e*，则将事件*e*及与其相关的数据*data*添加到事件调度器的事件队列中；
3. 事件调度器实时检查事件队列，获得队首的某事件*e*，然后寻找所有注册了事件*e*的模块，依次启动这些模块处理事件*e*和数据*data*。

事件的添加和处理是一个典型的生产者和消费者问题，当一个事件处理完毕才能处理下一个事件，当某个模块处理较慢时，就会造成队列过长，对于实时性较强的应用是不可接受的，所以事件调度器只在取事件时对队列加锁，并且对每一个注册该事件的模块均启动一个新线程进行处理。

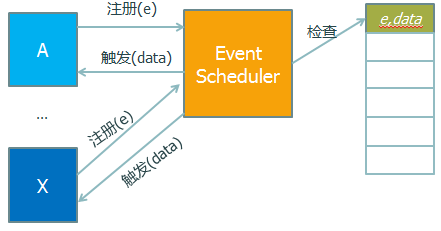


图4 调度器的工作过程

但考虑到有可能在处理该事件时，某些模块间存在依赖关系，例如出于性能考虑，在处理检测OpenFlow流时，DDoS模块过滤掉大量可疑的流，然后交由防火墙模块进行检测。为了解决依赖关系，模块在注册时指明了对事件*e*模块间的模块依赖。如图5中模块B依赖于模块A，那么调度器在遇到事件*e*时，检查模块依赖关系，发现模块A没有依赖关系，则启动其处理方法，接着发现模块B依赖于模块A，于是跳过，如此进行，分别新启动模块C和模块D的处理方法，当模块A处理完之后，此时继续调用模块B的处理函数。

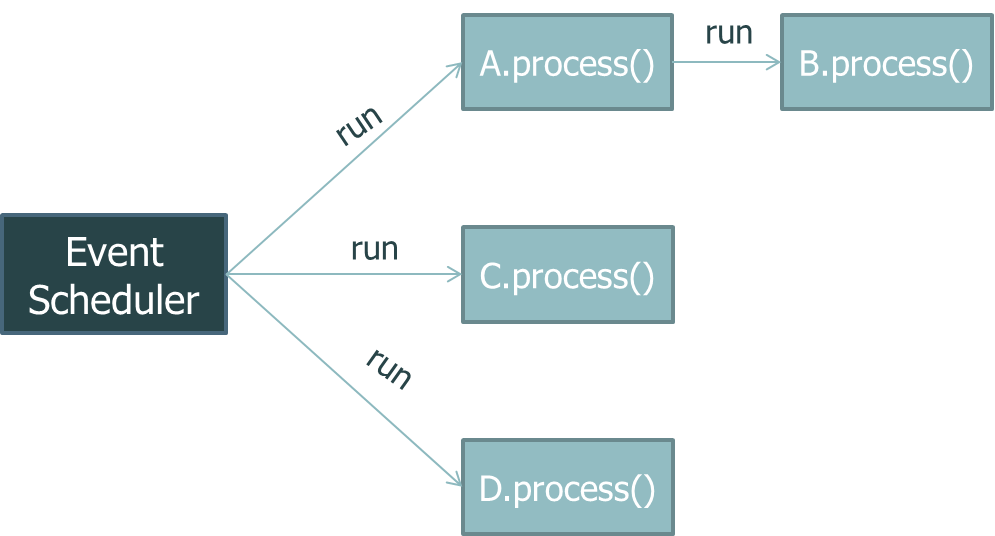


图5 模块依赖的并发处理机制

这种事件调度方式减少了队列等待时间，保证了模块并发性，节省了整体workflow的时间，同时考虑了模块间的依赖关系，保证业务可以正常运行。

#### 3.3.2消息队列与分布式部署

RabbitMQ是一个开源消息队列系统，是对AMQP（高级消息队列协议）的标准实现。消息队列（MQ）是一种应用程序间的通信方法，程序通过读写列队中的消息来通信，而无需专用连接来链接，从而分离了数据的发送和接收，实现软件解耦和异步。RabbitMQ的主要特征是面向消息、队列和路由（包括点对点和发布/订阅），可靠性、安全性高。

##### 消息的路由机制

如下图所示，P代表Producer，X代表Exchange，Q为Queue，C代表Consumer。P向Exchange传递消息时设定一个routing key，Exchange的对消息的路由方式有fanout和direct等多种，其中fanout方式将广播消息到所有与之绑定的queue，而direct方式则只发送消息给binding queue与消息的routing key匹配的queue。

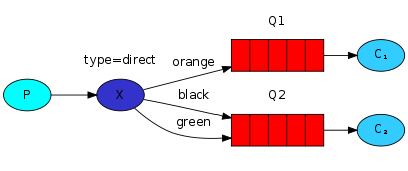


图 消息的路由机制

##### 在安全控制器的应用RabbitMQ

由于安全控制器各模块读写事件均通过事件调度器来完成，因此对事件调度器进行RabbitMQ方式的改写，由事件调度器直接与RabbitMQ Broker通信，可最大程度地避免了各功能模块的改动。

应用RabbitMQ之后，安全控制器将分布式化，不同模块运行于多个节点之上，每个节点上都包含一个事件调度器，负责将模块产生的事件转换为消息发送给RabbitMQ Broker，同时向RabbitMQ订阅此节点运行的模块感兴趣的消息类型（对应事件类型），并将收到的消息还原为事件传递给相应的模块处理。

节点2

节点1

事件调度器

事件调度器

模块A

模块B

模块C

模块D

消息

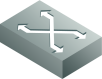
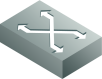
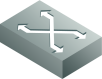
消息

事件

事件

图 RabbitMQ解耦安全控制器模块

##### STORM



流获取模块

流监控模块

流监控模块

流监控模块

流监控模块

Storm簇

Bolt

Bolt

Bolt

Bolt

1) 获取流

2)传递流

3)分发流

#### 3.2 具体实现细节

事件调度器在构造的时候，就与RabbitMQ建立连接并声明了一个topic类型的Exchange（类似direct,但是路由方式更灵活）和一个Fanout类型的Exchange，并分别创建了一个Queue和Consumer来接收消息。

由于RabbitMQ只支持字节流格式的消息，而事件是以对象形式存在的，所以需要将事件对象进行序列化和反序列化。首先需要为事件类和其中所嵌套包含的类添加java.io.Serializable接口，以使之可被序列化，同时编写一对方法实现对象↔序列的相互转换。

当模块调用addEventListener()方法向事件调度器注册收听事件类型，事件调度器同时调用RabbitMQ的queueBind()方法向Broker订阅相应的事件类型，即指定binding key。

模块调用addEvent()方法向事件调度器提交事件时，事件调度器将其序列化并用basicPublish()方法将其发送给Broker，参数中包含一个routing key，Broker中的Exchange将使用它来与各queue的binding key匹配，以将消息发送给运行有该模块的节点的事件调度器。

对于APP的订阅条件等数据，则适用于Fanout（广播）类型的Exchange，确保运行了被订阅模块的所有节点都能收到订阅信息。

事件高度器启动之后，将不断调用nextDelivery()方法来监听接收消息，一旦接收到消息，根据Delivery的properties属性确定其类型属于事件或订阅条件等，并将其反序列化还原为原来的对象，分由不同的业务逻辑处理。

### 3.4 应用通信交互

在3.4节中我们介绍了安全控制器内部的数据共享机制，可由事件调度器进行交互通信，然而安全控制器和北向安全应用的交互形式更为复杂，在分布式的环境中，两者很可能不在一台主机上，需要额外的网络通信协议和数据推送模式。

#### 3.4.1 订阅-推送机制

在分布式的设计中，安全控制器和安全应用都会检查环境中可疑的数据，并作出相应的判断和决策。安全控制器首先对输入的数据进行粗颗粒度筛选，甄别出一些可疑的数据，然后再由安全应用根据已有知识进行细粒度的核查，确定最终的策略。

在安全系统中，安全控制器和安全应用间的交互是基于“订阅-推送”的模式。主要过程是，当安全应用SAPP在安全控制器上注册后，将所关心的可疑数据模式以统一的格式提交给安全控制器，后者记录该模式后，每当收到数据后开始匹配，并将所有符合要求的数据推送给SAPP。

安全应用SAPP订阅时，发送subscription=(*id*, *et*, *module*, *pattern*)，其中*id*为应用编号，*e*为触发事件类型，*module*为处理模块，*pattern*为可疑数据匹配模式。APPManager收到之后，将订阅subscription分配给模块*module*，如程序清单1所示，(3)在调度器注册指定遇到事件*e*时转到模块*module*进行处理，进而(4)在模块*module*中添加了一个订阅处理，申明遇到事件*e*时对数据匹配*pattern*模式。：

Function: add\_subscription

Input: *id*, *et*, *module*, *pattern*

Output:void

*m* = findModule(*module*) (1)

scheduler.addEventListener(*et*, *m*); (2)

m.addSubscription(*et*, *pattern*) (3)

return (4)

图6 添加订阅伪代码

之后当调度器遇到事件*e*=(*et*, *data*) 时（*et*为类型*、data*为数据），转交由模块*module*处理；模块*module*开始匹配pattern中所有模式，如果符合要求，则触发一个MALICIOUS\_DATA\_FOUND事件，最终将data传递给APPManager的APP Data Pusher子模块，后者将数据推送到SAPP。

#### 3.4.2 可疑数据检测

由于各个模块对可疑数据的存储和处理机制不尽相同，有的使用内存，有的使用数据库；各种匹配模式也较为复杂，安全控制器中使用了一种较为通用的模式去甄选可疑数据。

我们使用一种树状的结构表示匹配模式，如图7所示。每个叶节点表示一个具体的计算操作，记为操作订阅Operation subscription，运算包括三部分：左端表达式、运算符和右端表达式，表达式为若干变量的加减乘除运算，运算符为比较运算符（如大于、小于、等于、不大于和不小于）。例如图中左下角的节点表示当前流的byte\_count是否小于某一条参考流的byte\_count-1000，运算类型为整形，返回满足比较条件的数据项。

每一个中间节点表示若干子树的运算结果的复合值，记为复合订阅Compound subscription，运算包括两个部分：子树和复合运算符，子树记为该节点下面的子匹配模式，复合运算符将上述子树返回数据项做复合运算（与或非），将运算结果返回父节点。

还有一类中间节点也属计算范畴，但这些节点的输入是集合，在计算前需做集合运算，称为集合订阅Set subscription，其包括四个部分：左子树集合、集合运算符、右表达式和比较运算符输入是子树的数据项集合，集合运算符包括：计数（COUNT）、求和（SUM）和求平均数（AVG）等，将集合运算符作用于左子树集合，则会获得一个数值，然后类似于操作订阅，计算表达式是否满足条件，如满足则向父节点返回左子树全部数据项，否则返回空。

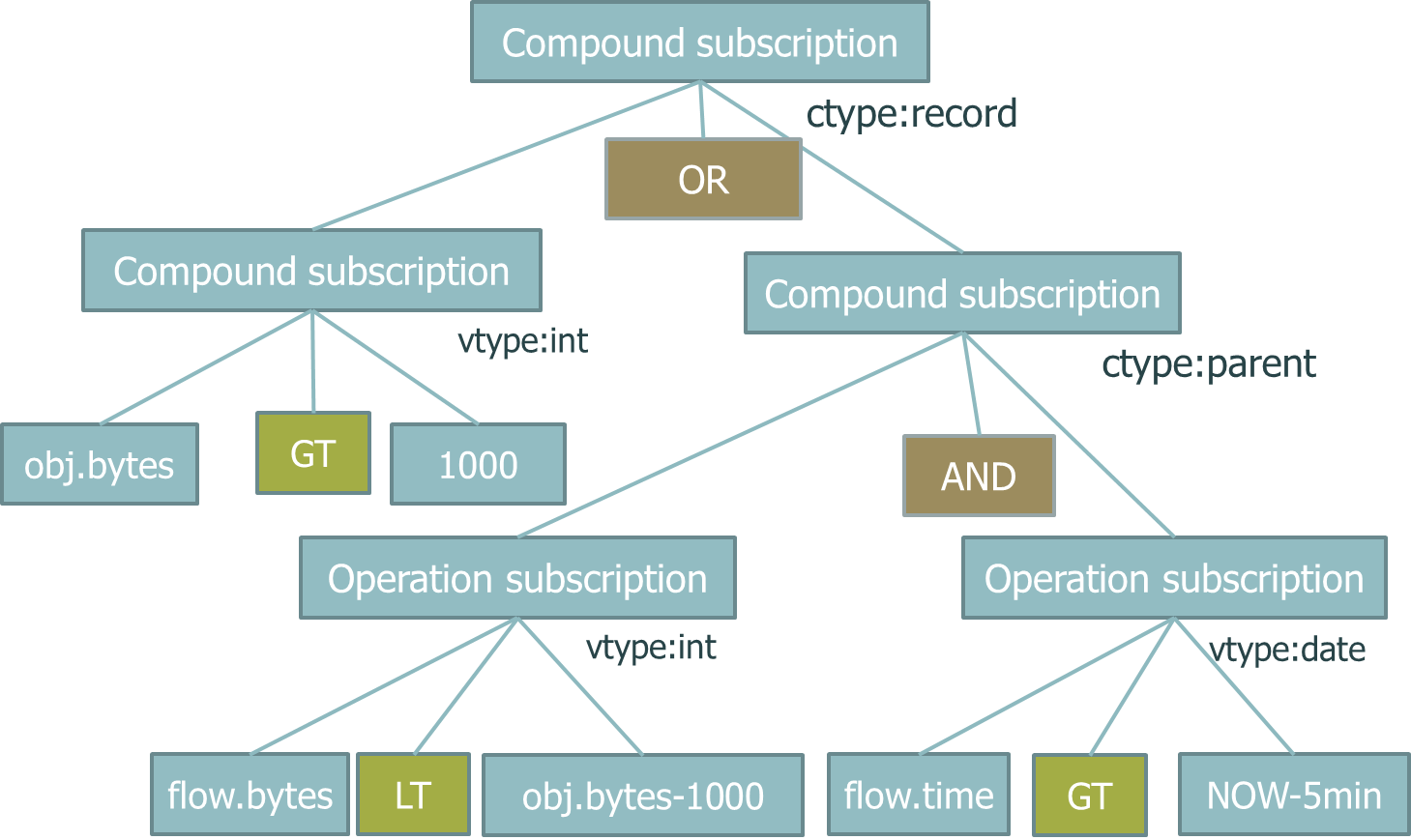


图7 树状订阅匹配模式示意图

在含有计算操作（操作订阅和集合订阅）中，表达式中可含有“obj”字符串，表示一个用于计算的参考数据项，图6左下角的obj.bytes表示一个参考流的byte\_count字段；同样的，表达式中还可含有“db”字符串，例如db.flow.time表示数据库flow表中的time字段。这样，匹配模式支持内存计算和数据库计算，也可支持包含两者的混合计算。

* 对象计算
  + object.x>100
  + object.x + object.y = 100
* 数据库计算
  + flows.date > ‘2010-09-09 11:00:00’
* 混合计算
* flow.bytes < object.bytes + 100

通过使用三类订阅模式，可以组成不同的订阅条件，使得SAPP可以自由地选择自己感兴趣的可疑数据。我们下面给出三类典型的安全应用所需的订阅条件：

* Anti Sync-Flood APP

对于抵御Sync Flood的DDoS攻击，该APP只需找到“目的地址相同、byte\_count小于3的openflow流” 数目大于2000，那么这些流即为可疑数据。

* count(object.dstaddr=ref.networkaddr and object.pkg\_count<3)>2000
* Anti Worm APP

以W32/Nicha.Worm这种蠕虫病毒为例，该蠕虫攻击的数据包大小为92，协议为icmp，那么订阅条件应为

* + object.byte\_count /object. pkg\_count=92 and object.protocol=0x01
* Anti Spam APP

对于垃圾邮件而言，其特征为“目的端口为25，且源地址相同的流”数量大于20

* count(object.dst\_port=ref.dst\_port=25 and object.src\_ip=ref.src\_ip)>20

那么这些应用只需提交上述简单的订阅条件，安全控制器随后在收到数据后启动检测过程。以Anti Sync-Flood APP为例，安全控制器将收到的原始OpenFlow流作为输入，递归地比较SAPP所注册的匹配模式，获得若干满足条件的数据项集合，将上述所有可疑的数据推送APP端。

安全控制器和安全应用均提供了基于资源的REST服务，具体接口的定义参见附录1。安全应用可随时根据环境和策略调整自己的订阅条件，只需向安全控制器的REST服务发送PUT请求即可，灵活简单。

### 3.5 策略解析

安全控制器通过北向的REST API接收安全应用下发的安全策略。由于安全应用策略模型主要基于SVO(*Subject, Verb, Object*)的语法格式，并且涉及包括租户、用户、流等多个维度，因此安全控制器需要通过策略解析功能将安全策略转换为网络控制器能够识别的安全指令。

#### 3.5.1 安全策略的模型

策略分为复合策略和原子策略两级。复合策略中包含优先级、下发APP等信息，每条复合策略包含多条原子策略。原子策略包括三部分，如图1所示，主要包含一个语义标识，一个主体，一个谓语和一组客体，通过SVO的语法结构可用于标识SDN环境中控制层需要执行的策略指令。一条原子策略的语义即指定主体（subject，通常是网络控制器或安全控制器）、对目标（object，通常是主机或匹配某种特征的流量）的动作（action，对应流的操作：重定向、清洗、阻断、镜像等）。

* 语义标识（isNegated）：若该标识置为1则策略为否定策略；否则为肯定策略。
* 主体（Subject）：每条策略只有一个主体，用于确定产生策略描述动作的主体对象，如网络控制器、安全控制器等。
* 谓语（Action）：每条策略包含一个谓语，用于描述主体对客体需要进行的动作类型，如创建、删除、重定向等。
* 客体（Object）：一条策略可以和多个客体相关联。每个客体对象由名称、流向、标识以及值类型组成。
* 名称（Name）：是关于客体所属云环境中信息维度的描述，包括Tenant， User， Flow， Subnet等。
* 流向（AimType）：用于描述客体是信息的起点或是终点，值为Source或是Destination。
* 对象标识（ObjectId）：每条流均通过MD5算法产生唯一的序列号进行标识。
* 值类型（ValueType）：用于解析策略时生成客体属性表达式。

Policy

Subject

Action

ObjectList

Object

Object

……

Name

AimType

ObjectId

ValueType

图8 安全策略结构

通过解析，目标被对应到N条流表匹配项（如：目标描述为“流向某用户所有主机的流量”，该用户有3台主机，则对应成3条匹配项；目标类型为流时则对应成1条匹配项），每条策略动作被对应成M条流操作（如：“清洗流量”动作对应成2条流操作：目标流量重定向到清洗设备、清洗设备出来的流量重定向回目标主机）。综上，一条原子策略可被解析成N\*M条流命令，下发给SC的命令模块。

图8.1 安全策略与命令的对应关系

#### 3.5.2策略解析流程

安全控制器接收到安全应用下发的安全策略后进行策略解析操作，将安全策略模型转换为流命令Command。其中流命令Command包含了指令动作（Action），操作类型（ActionType）以及一组指令对象CommandObject。每个指令对象CommandObject中均包含网络控制器端下发命令所需的静态流名称（flowname）、源和目的端的mac信息（source\_info，destination\_info）。网络控制器端的安全代理模块接收到安全控制器发送的流命令完成进一步的指令下发。策略解析为流命令可分为如下5个步骤，如图2所示：

1. 查询DeviceManager，获取底层安全设备的mac地址macDevice；
2. 若Policy的isNegated标志位为1，则流命令指令动作设为Deny；否则指令动作设为Allow；
3. 查询CloudAgent模块生成的多维度映射表，将Policy中的客体描述的Tenant，User，VM对象的IP地址映射至具体的VM的Mac地址VMMac。
4. 通过Policy中客体的名称、对象标识、第一步生成的macDevice和第三步生成的VMMac构造流命令对象，添加至流命令列表中。
5. 循环执行3~4步直至策略客体组中的策略对象遍历结束。

若策略的标志位isNegated为1则将流命令Action设置为Deny；否则设为Allow

Yes

No

策略对象遍历结束？

生成PUSH\_FLOW事件，将流命令推送至网络控制器的安全代理

根据策略对象包含的IP地址信息和CloudAgent模块提供的方法完成TENANT－＞USER－＞VM，IP－＞MAC的映射生成macVM

根据策略对象和macDevice以及macVM构造流命令对象，添加至流命令列表中

初始化流命令列表，查询DeviceManager模块获取底层设备mac地址macDevice

图9策略解析流程

#### 3.5.3策略的一致性的保证

由不同应用先后下发的安全策略，可能存在对相同主体（subject）对相同客体（object）实施相互矛盾动作（action）的情况。因此，策略解析模块采用如下的机制和流程保证策略的一致性。

策略冲突的判断条件是：(1)主体有交集；(2)对象有交集；(3)动作存在语义矛盾。**两条策略是否冲突的检测遵循如下原则：**

* 1. 检查两者的主体是否相同
  2. 若相同，检查对象的匹配是否相同或存在包含关系
  3. 若对象相同或互包含，检查动作是否冲突

以流量调度策略为例，详细说明判断过程：

1. 执行主体都是NC，略过
2. 判断出现这三种情况之一时，则认为对象相同或相互包含：
   1. 对象a的每条匹配都与对象b相同
   2. 对象a的部分匹配都对象b的部分匹配相同，另一部分匹配包含于b的一部分匹配（b包含a）
   3. 对象b的部分匹配都对象a的部分匹配相同，另一部分匹配包含于a的一部分匹配（a包含b）
3. 检查流动作是否矛盾（矛盾关系预先），若矛盾
4. **冲突后处理**
   1. 为存在冲突的若干策略比较优先级（两两比较，机制见下一条），并存入优先级队列；
   2. 若是下发到别的主体（如NC）执行的策略，下发时附带上该策略的优先级
   3. 若是主体是SC自己的策略（即第三种：安全策略），执行时按照从优先级高到低进行
5. **当两策略冲突时，优先级的比较依据（从高到低）：**
   1. 比较发起者（下发策略的APP等）的权限，权限高者优先级高
   2. 比较主体的权限，权限高者优先级高
   3. 比较对象的具体程度：具体者优先级高。如：流量调度类规则，FlowMatch的项越多，则越具体，优先级越高
   4. 比较动作类型优先级，为每种动作均设定优先级，如CLEAN的优先级大于REDIRECT，而绑定IP和绑定MAC动作的优先级相同等等

策略一致性的检测流程如下：

标记无法处理，

并抛出事件

比较主体优先级

比较对象的具体程度：具体者优先级高

比较动作类型优先级（如CLEAN>REDIRECT）

比较发起者（下发策略的APP）权限

是否是否定性策略（否定策略优先级高）

得出比较结果，生成优先级数值，更新优先级队列

相同

相同

相同

相同

相同

不同

不同

不同

不同

不同

### 3.6 日志管理

安全控制器中，日志分为设备告警日志和安全控制器自身的运行日志两类，两种日志统一由LogManager模块管理。当SC分布部署于多个节点之上时，推荐只启动一个LogManager模块，这样就能在单一节点上统一管理日志，监控各节点上SC进程及各个安全设备的运行状况。

日志和处理也是以事件形式进行的，对应的事件类型为RECEIVED\_LOG，当收到设备告警或者SC运行中产生重要信息时，都可以调用EventScheduler的addEvent()方法向队列中添加RECEIVED\_LOG事件，两类不同的LOG通过事件参数来区分：设备告警日志对应的是LogEventArgs，SC运行日志对应参数类型为ConsoleLogEventArgs。这两种事件对应着不同的处理逻辑。

SC采用了分布式部署，不同功能的SC模块运行于不同的节点之上，各自的运行日志也分散输出于各自所在节点，这对日志的监控、分析和管理带来不便。因此有必要添加一种集中输出的方式，将关键的日志信息集中输出于一个运行着日志管理模块的节点之上。

#### 3.6.1 具体实现

LogManager模块当前功能是集中管理设备产生的告警日志，因此重用此模块即可。此模块已监听事件类型RECEIVED\_LOG，运行日志也可以归为此类型，只需再设计一种相应的事件参数ConsoleEventArgs类即可。

|  |
| --- |
| ConsoleEventArgs |
| +hostName: String;  +logString: String;  +module: String;  +line: int ;  +logLevel: LogLevel; |
| +toString():String |

参数说明：

logLevel --> TRACE, DEBUG, INFO, WARN, ERROR

hostName--> 节点名

module --> 模块名

line --> 行号（TODO）

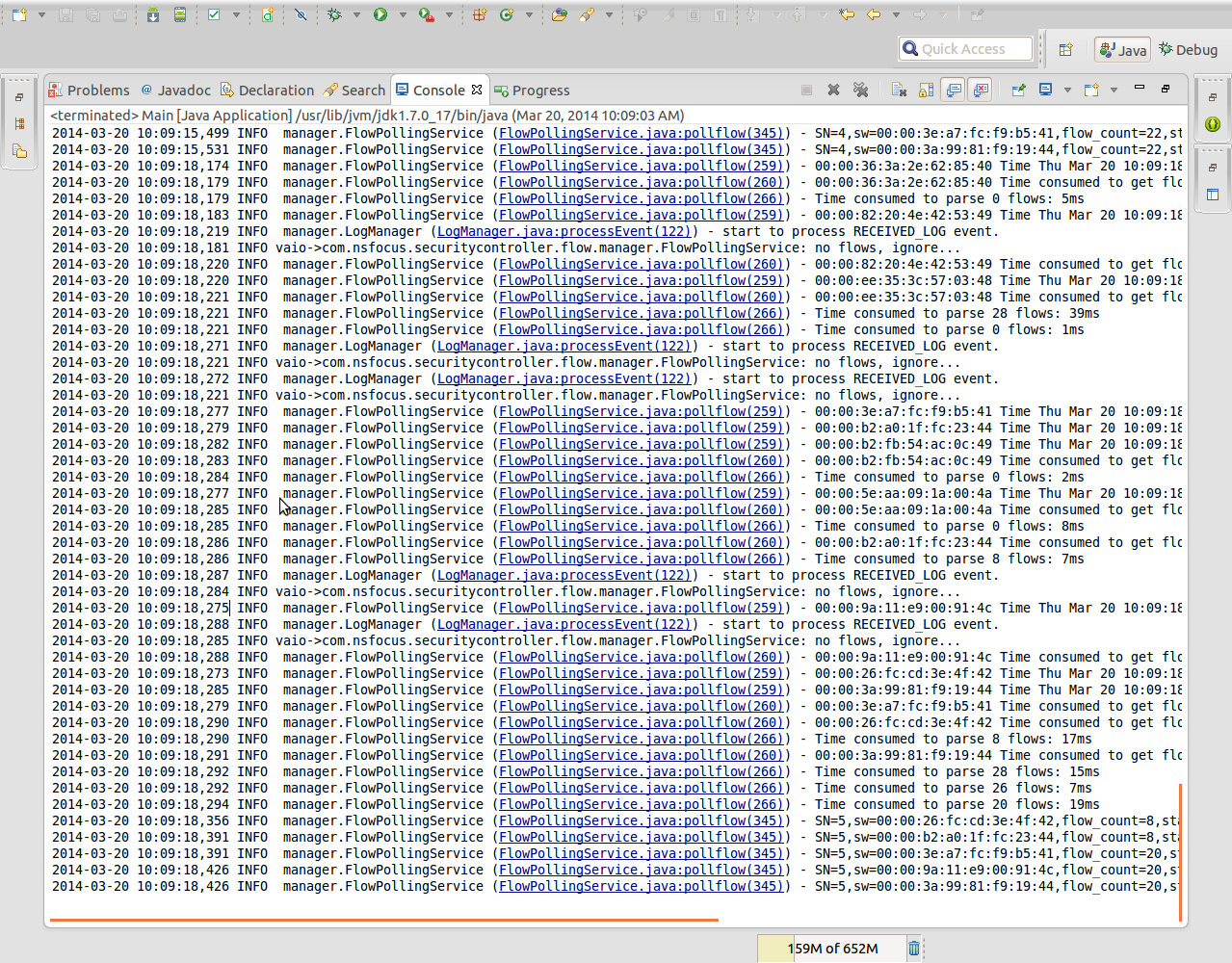
logString --> 日志内容

当LogManager收到RECEIVED\_LOG事件之后，会判断EventArgs类型，如果是ConsoleEventArgs，则将此日志在标准输出中打印出来。

通过这个类，可以标识运行日志的时间、级别、来源节点、来源模块和对应代码行号（未完成），最后是日志的消息主体。

一般来说，为了避免日志太过杂乱和增加EventScheduler不必要的负担，推荐只将一些关键的SC日志信息通过LogManager管理，比如模块的启动信息和一些错误日志等等。

#### 3.6.2 运行效果



### 3.7 网络控制器Agent部署

#### 3.7.1 安全策略实施的业务流程

层级：tenant,user,vm  
目标ID：  
动作：clean,IDS  
优先级：1,2,3

match：port,ip,mac  
action：redirect，mirror  
device：mac,ap

priority: 1,2,3

动作：clean

存储

设备详细信息

资产详细信息

考虑到多个不同类型安全设备进行组合， Security Controller对策略的解析将会变得比较复杂，需要考虑到与已有策略之间的关联性，策略之间冲突的检测和处理（由另外的上层模块来做），另外还需要考虑不同粒度的策略之间可能的包含关系，例如VM级的策略可能会覆盖原有Flow级的策略。当同类型设备有多个时，还需要设计算法使得所选设备在传输开销和负载等方面达到最优。

为方便处理策略冲突的情况，可以综合APP和策略类别等因素为解析结果添加相应的优先级。

对于比较常见的重定向和流复制类的操作，最终解析结果应为一个或多个如下特征对。

|  |  |
| --- | --- |
| **Match** | In\_port,MAC,IP,端口等 |
| **起始点** | 可由MAC值或者attachment point标识 |
| **终止点** | 同上 |

策略解析完成以后，将以Match提取特征码作为Key建立HashMap保存，当新的策略解析时，如查询到HashMap中已有相同Match，就需要与已有策略进行合并，合并结果更新到HashMap中，并生成相应的指令发送给Floodlight进行实施。对于重定向动作，则重定向应从当前重定向路径中最末设备的出口开始（或插入于当前路径的开始处），如果为流复制，则应在原有复制的基础上添加一条复制路径。

对于策略的解除，由于当前是以动态流的方式进行牵引，所以只需要删除路径中第一条流即可。

最后对于网络控制器Floodlight端，也需要做相应的修改，主要为支持流的拆除。

#### 3.7.2 floodlight端Agent接口设计

##### 3.7.2.1 REDIRECT\_FLOW（流量牵引）

以下样例，实现了”将所有目的地址为VM1的流量经过Dev1和dev2处理，再返回VM1"，commandlist中包含三个元素，对应三个流量牵引操作:

1. vm1 ---> dev1 （所有到vm的数据 ）
2. dev1 ---> dev2 （包含修改源mac操作）
3. dev2 ---> vm1（根据源mac区分来自设备处理过的流，优先级>1中的重定向流

{

"commandlist": [

{

"priority": 100,

"flowname": "USERac33682274c74a5fb3663677a0794fd5\_0",

"endpoint": {

"ap": "30.0.0.1%%tap1",

"mac": "52:54:00:a9:b8:b4"

},

"startpoint": {

"no\_inport": true,

"mac": "fa:16:3e:ea:8c:f7"

},

"matcharguments": {

"dl\_dst": "fa:16:3e:ea:8c:f7"

}

},

{

"priority": 100,

"flowname": "USERac33682274c74a5fb3663677a0794fd5\_1",

"endpoint": {

"ap": "30.0.0.12%%tap1"

},

"startpoint": {

"ap": "30.0.0.1%%tap2",

"mod\_dl\_src": true,

"mac": "52:54:00:a9:b8:b5"

},

"matcharguments": {

"dl\_dst": "fa:16:3e:ea:8c:f7"

}

},

{

"priority": 100,

"flowname": "USERac33682274c74a5fb3663677a0794fd5\_2",

"endpoint": {

"mac": "fa:16:3e:ea:8c:f7",

"overide": true

},

"startpoint": {

"ap": "30.0.0.12%%tap2"

},

"matcharguments": {

"dl\_dst": "fa:16:3e:ea:8c:f7",

"dl\_src": "52:54:00:a9:b8:b5"

}

}

],

"type": "REDIRECT\_FLOW"

}

参数说明：

Type：string --> 操作类型，redirect\_flow代表流量重定向

Commandlist --> 命令列表，其中每一子项代表在指定的两点之间建立一条流

Startpoint/endpoint --> 起/终点信息

ap：string --> attachment point，格式为ip%%portname,如30.0.0.1%%tap1  
 mac ：string--> MAC值，Agent会优先使用ap寻找接入点

no\_inport ：boolean--> 不匹配ingress port（比如要重定向“所有到xxx的流”）

mod\_dl\_src ：boolean--> 修改源mac值，为true时mac不能为空

overide：boolean --> 将目标所在switch的回注流优先级+1，避免被再次重定向

Matcharguments --> 配置参数

//DONE：是否可以直接套用Floodlight中的OFMatch类。答可以

String dl\_src

String dl\_dst

int ether\_type [ 0x0800 | 0x88cc…. ]

String nw\_src

String nw\_dst

String nw\_proto [ icmp | tcp | udp ]

String uri

int tp\_src

int tp\_dst

本例所牵引的流量是“所有目的地址为xx的流“，为了防止处理后的流量返回原目的主机时再次被匹配和牵引，需要对处理后的流量进行标记，采用的是修改源mac的方法（也可考虑修改VLAN），这样在流量返回时同时匹配目的地址和被修改的特定SRC\_MAC，并配以更高的priority，就能避免与未被处理过的流量混淆。参数中的mod\_dl\_src和overide（分配更高priority)就是为此种情况设置的。

//Done一条流两次由同一交换机的同一端口进入，如何区分？

答：将数据包加上相应的设备标记

出现的场景：

1. 将dst\_mac = xxx的流处理以后返回目的所在交换机。流量会两次从br-tun进入br-int。目前的处理办法是将处理过后的流量修改src\_mac，和原始流量区分，但不具有通用性，比如Match中包含Src则不宜修改。
2. dev1和dev3位于同一节点，dev2位于另一节点，流量流经dev1 --> dev2 --> dev3，则流量会两次由br-tun的patch端口进入br-dev。
3. 此种情况比较特殊，可视为两个策略的对象存在交集但相互不包含。例如策略一为将src\_mac = xxx的流转入dev1，策略二为将dst\_mac=yyy的流转入dev2，那么对于同时满足src\_mac = xxx，dst\_mac = yyy的流，将存在以下的问题。（1）流优先级相同，则匹配哪一条规则取决交换机实现，具有随机性，此种结果是不允许的。（2）假设src\_mac匹配优先级较高，则若源和目的在同一交换机上，dev1和dev2在同一节点上，则满足src\_mac = xxx，dst\_mac = yyy的流从dev1上返回br-int时被策略二匹配，再次从同一端口进入设备所在交换机，此时应该匹配策略二的规则使其进入dev2，为了和第一次进入dev1时的流区分，也需要对其Match进行标记，也就是说策略间如果存在交集（非包含关系），则需要对（1）交集（2）两个补集进行单独的处理，操作将会比较繁琐，流规则也会比较复杂。

可能的解决办法：

1. 避免同一流多次经同一端口进入同一交换机。（此办法不行，因为策略实现需要顺序经过某些设备）
2. 为处理过的流加上标记，但不能改变原有的Match，或者修改之后能够相应地改回来。

##### 3.7.2.2 Redirect\_Flow(redesign)

考虑到网络控制器会接收到来自不同APP而不仅仅是安全应用的策略，因此为了保证安全策略与其它应用下发的策略相容，即保证生成的OpenFlow流规则的一致性，应将策略解析的最后一步亦即根据网络上下文环境解析策略，生成一系列的OpenFlow流规则置于网络控制器端完成。

//TODO:问题：如果考虑到第三方上层应用的策略，如何保证安全应用的策略不与之冲突？利用全局流表？

接口中应该包含完整的策略信息。1、信息必须是完备的，比如设备接口的详细信息；2、策略语义应该是完整的，即对何种特征的流进行何种操作，操作须是完整的，不能是其中一个子集。

重定向是一个比较有代表性的操作，所以首先定义重定向的数据格式。

{  
 "commandlist": [  
 {  
 "policyPriority": 100, //策略的优先级  
 "commandName": "USERac33682274c74a5fb3663677a0794fd5\_0",  
 "hardtimeout":0,

“idletimeout”:500,  
 "matchArguments": {//对应着OFMatch  
 "wildcards": "int",  
 "inputPort": "short",  
 "dataLayerSource": "string",  
 "dataLayerDestination": "string",  
 "dataLayerVirtualLan": "short",  
 "dataLayerVirtualLanPriorityCodePoint": "byte",  
 "dataLayerType": "short",  
 "networkTypeOfService": "byte",  
 "networkProtocol": "byte",  
 "networkSource": "int",  
 "networkDestination": "int",  
 "transportSource": "short",  
 "transportDestination": "short"  
 },  
 "devices": [//可能有多个device  
 {  
 "deviceid": "string",

"tag":1,  
 "ingress": {//进出口信息包含“交换机ip：端口名”和MAC  
 "mac": "52:54:00:a9:b8:b8",  
 "ap": "192.168.1.10:tap1"  
 },  
 "outgress": {  
 "mac": "52:54:00:a9:b8:b9",  
 "ap": "192.168.1.10:tap2"  
 }  
 }  
 ]  
 }  
 ],  
 "type": "REDIRECT\_FLOW"  
}

##### 3.7.2.2 DROP\_FLOW

{

"type": "DROP\_FLOW",

"commandlist": [

{

"id": "p3",

"policyPriority": 100,

"idleTimeout": 5000,

"hardTimeout": 0,

"commandName": "USERac33682274c74a5fb3663677a0794fd5\_0",

"matchArguments": {

"dataLayerSource": "52:54:00:a9:b8:b8",

"dataLayerDestination": "52:54:00:a9:b8:b9",

"dataLayerType": 2048,

"networkProtocol": 6

}

}

]

}

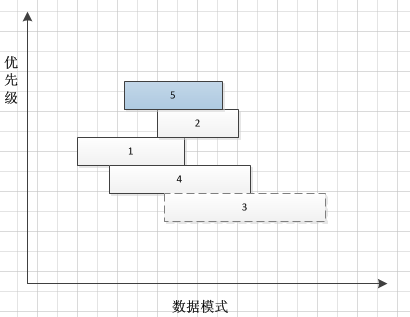
#### 3.7.3 多策略兼容的实现

前面第二章已经提到，当多个策略要对同一流（或有包含关系的流）进行操作时，需要确保所有策略能够得到正确的实施。

考虑到Floodlight存在其它应用的策略，因此策略解析的工作放到Floodlight端来完成，相应地SC向Floodlight发送的操作命令应当包含完整的语义，例如将某种模式的流量牵引到某设备。

下面以重定向类的操作为例，分析当收到一个新的策略时，如何与已实施的策略进行比对，并下发正确的流牵引指令。

##### 3.7.3.1 较低优先级的相关策略

当一条新的策略进入时，首先要找出所有数据模式相关的已有策略。所谓相关，即数据模式之间存在相等、包含或交叉关系的策略。较低优先级的策略关系到当前策略处理完之后，应该当流量如何分配，即**流量去向**的问题。

需要注意的是，一旦出现当前数据模式的父集（如上图策略4），就会形成“屏蔽”效果，之下的其它策略（策略3）的流量将不再与当前策略有关。

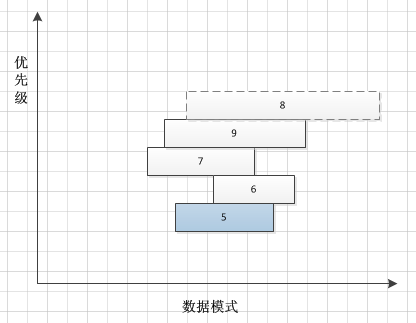
较低策略的处理更为简单，只要依次下发 Match = Pattern n,Device5 ---> Device n的流即可，只需流优先级与策略优先级有相同的次序，即可保证流量得到正确的分配。

对于一个新的设备来说，其流优先级的分配也是一个新的空间，无需考虑与已有流优先级的冲突问题。

流规则下发的同时，应当同时保存与当前策略**出口流量相关的策略**2、1、4和它们的流优先级，以备后续相关策略进入时参考。

##### 3.7.3.2较高优先级的相关策略

相应地，优先级较高的相关策略关系到当前策略**流量的来源**。

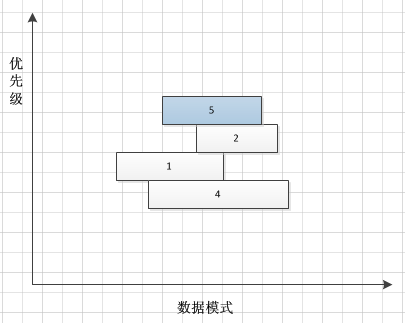


同样的，较高策略也存在“屏蔽”的现象，比如策略9就屏蔽掉了优先级更高的策略8的到策略5的流量。

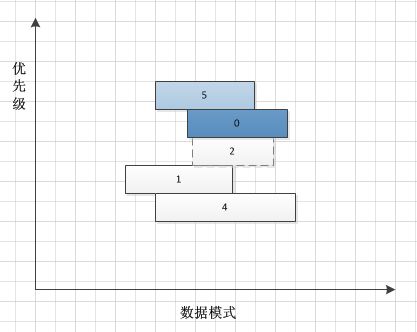
依次处理较高策略，对于策略4，需要增加一条由策略4的安全设备流量出口 ---> 策略5安全设备流量入口的流，其流的Match为策略5的数据模式，优先级应该小于策略1，这样就完成了流量来源的处理。

##### 3.7.3.3 出口流量相关策略

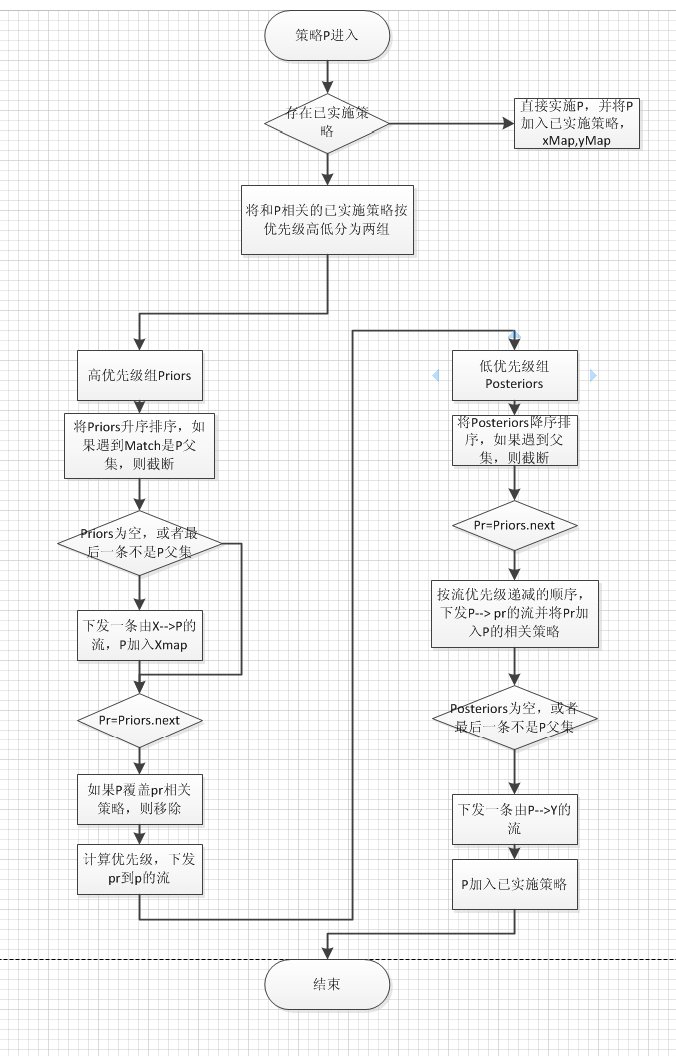
前面提到，为了流优先级管理和相关策略查找的便利，每一条策略应当保存与其出口流量相关策略。



比如上例中的策略5实施完之后，就记录了出口流量相关策略2、1、4及其相关的流优先级。当有新的相关策略加入时，就需要更新此记录。



当策略0加入之后，就屏蔽了5 ---> 2的流量，此时就应当在记录中删除2，并删除相应的流规则，策略0可以使用原策略2的流优先级。



例如接收到一个新的策略，要对VM进行流量清洗操作，如果已有一策略是牵引此VM流量经过某安全设备，那么根据优先级的大小在当前安全设备前/后插入一个ADS设备即可，对应着产生相应的流拆除和建立指令，下发给Floodlight。

再以包含关系为例，如果发现此VM所属USER的所有流量已经被某安全设备处理，但优先级较低，那么应该优先让此VM的流量经过ADS，再让ADS处理后的流量进入为此User分配的安全设备，而不是直接返回网络。此过程则只有流建立的指令产生。

##### 3.7.3.4 策略对象包含/等价关系的检测

如果策略的对象是VM以上（User 、Tenant等）的级别，包含和等价关系比较好确定，利用知识库中有关云环境（如Openstack）的信息即可。但粒度到达流级别，等价和包含关系的检测稍微复杂，比如某IP和某MAC实质为同一流，而某些更细粒度（比如同时指定了源/目的ip和端口）的流，很可能会被某些只指定了MAC或IP的流包含。再加上OpenFlow的Match域中12元组基本可以任意组合，所以产生的形式会多种多样，需要建立一个完备的机制来检测流量的包含关系。

粒度大于VM的策略，应该分解为多个VM粒度的子策略来保存，因为VM对应着OFMatch中的MAC。

Match又可以分为源和目的两部分，有的Match只有源信息，有的只有目的信息，不同的Match的处理方式是不一样的。

|  |  |  |  |
| --- | --- | --- | --- |
| **Match信息** | **流牵引的起点** | **流牵引的终点** | **备注** |
| 源+目的 | 源所在交换机 | 目的所在交换机 | 一条完整的路径 |
| 源 | 源所在交换机 | 直接回注网络，由NC处理 | 不完整的路径，依赖NC的路由算法 |
| 目的 | 目的所在交换机 | 目的所在交换机 | 完整的路径，由于起点和终点相同，需要对处理过的流量进行标记以便区分。 |

1、当策略A开始解析时，首先找出所有优先级大于（List<Policy> higher）和小于A（List<Policy> lower）的策略，分两组保存，每条策略同标注与A的包含、被包含或相交关系。查找的过程中，一旦遇到B包含或等于A，则过程可以停止。

2、优先级大于A的相关策略关系到A流量的源头，最上游源头1）原始主机2）Match为父集关系的策略。因为每条策略都保存了优先级低于自身的相关策略的流优先级，只需在策略相关设备出口添加一条如下优先级的流即可：低于同时存在于List<policy>related和List<Policy> lower中的最低策略的流优先级。

3、对于在List<Policy> lower中的策略，只需要按优先级顺序将A策略处理过后的流量导向各策略即可，终点为原始目的或者父集策略。

##### 3.7.3.5 关于流优先级的分配和管理

正确地分配流规则优先级就可以实现对不同关系数据流的施加不同的策略链处理。总体的需求有：

1、数据模式具有相关性的流优先级不能相同

2、数据模式不相关的流可以分配相同的优先级

3、每一个交换机+IN PORT是一个独立的优先级分配空间，不同的空间可以分配相同的优先级（如果没有指定匹配In Port，那么每个交换机是一个优先级分配空间）

4、已经分配的流优先级应该以一定形式记录，方便在建立新的流时参考并避免冲突。

##### 3.7.3.6 OF流生成细节

SwitchPort起始+终点

List<NodePortTuple>

路由引擎

Flow Entries

路由引擎

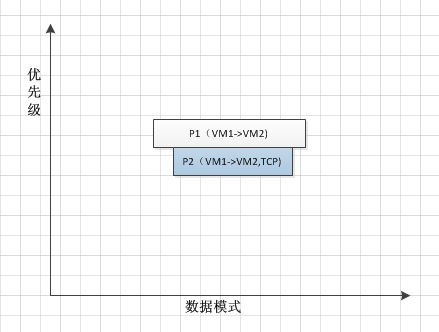
pushFlow

首先使用RouteEngine计算出从起点到终点（SwitchPort类）的路径，路径也是一个Swith类似Port形式（只存储存储ID不指向实例）的List，我们需要在相邻的两个SwitchPort之间下发一个OpenFlow的Flow Entry（以下简称flow）。

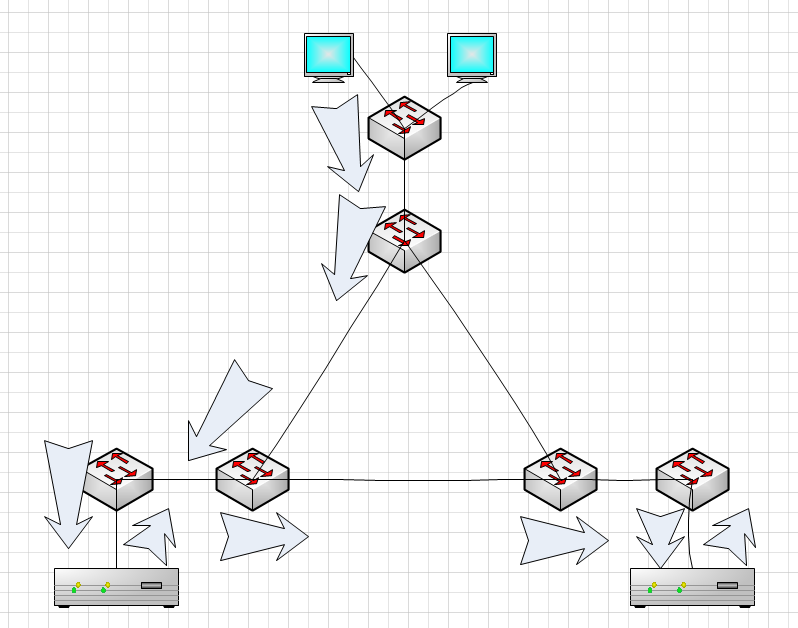
一般来说，flow的Match由策略的Match来生成，其中In\_port的设定分情况处理。如是从安全设备出发，则一般有明确的InPort（设备的出口），如果Match没有源信息或者只指定了四层信息，则意味着没有源主机，此时不能匹配InPort。

##### 3.7.3.7 演示效果

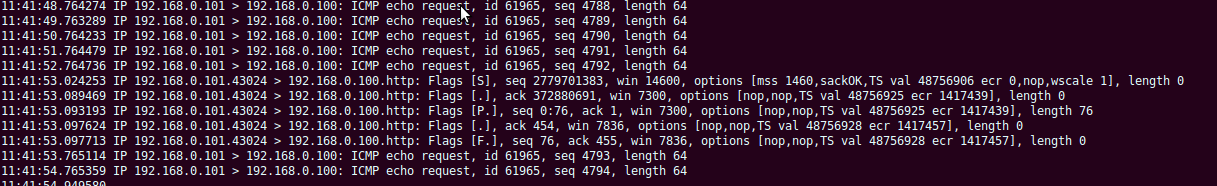
两条策略，较高优先级的策略p1要求将所有VM1->VM2的数据重定向经过Device1，较低优先级的策略P2要求将VM1->VM2的TCP数据包重定向经过Device2。



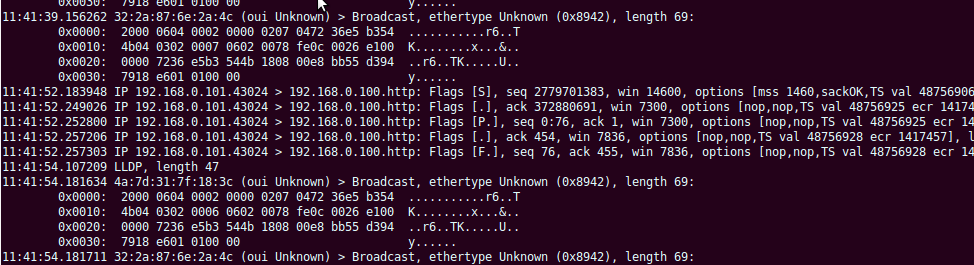
以下测试所用的网络拓扑图，测试用虚拟主机VM1和VM2以及安全设备Device1和Device2都在不同的交换机上，它们通过一组（三个）相互联通的Tunnel交换机连接在一起。



策略实施以后，Device1可以同时接收到icmp和http的数据。



而Device2只能接收到Http的数据包。



#### 3.7.4 流规则冲突的检测与处理

##### 3.7.4.1 修改匹配域的选择

当数据包前后两次以上进入同一物理端口，而每次执行的动作不同时，就需要对数据包进行标记，也就是修改数据包头，以使其Match不同的OpenFlow Entry。对OpenFlow1.0版本来说，能修改和匹配的header field中，Vlan和TOS，包括源Mac都可以作为修改的目标。

1、如果原数据没有Vlan信息，那么加入Vlan基本不会对后续的转发等造成影响，但是在多租户环境中，Vlan是一种常用的隔离技术，修改之后可能会造成一系列问题。

2、TOS（DSCP）是IP中的一个Field，只对IP类型的数据包有效，实际网络中常见的数据，未尝不可。

3、源MAC，一般来说交换机转发更多的是比对目的MAC，修改源MAC不会对转发的正确性有大的影响，但是可能对各主机和控制器的地址学习有造成困难。

6.2 环回的检测

前面据说的两次进入同一个物理端口，即形成了环路。由于数据流Match之类存在的交叉和包含关系，一条流是否形成环路，并不是显而易见的，而是需要通过一定的算法来确定。

### 3.8 安全设备部署

#### 3.8.1 单个节点上安全设备的部署

br-tun

br-in

br-out

Device

gre

br-tun

br-dev

patch

原连接方式

新连接方式

gre

Device

Device

Device

Device

最初设计br-int,br-out是因为当时无法使虚拟机的两网卡连接于同一bridge（启动过程中卡死），通过改进KVM网络设置，这一问题早已解决，故无需再使用两个bridge。br-dev的作用和连接方式都类似br-int，其上连接多个虚拟安全设备，通过br-tun连接到整个虚拟网络当中。安全设备通过单个brige连入网络也可以使控制器牵引流量的操作变得更为简单。

关于单网桥连接多个安全设备的可行性，已进行模拟试验（单物理节点）验证。编辑kvm虚拟机启动脚本，在OVS端口建立时置位其config中的no-flood位，即可避免广播风暴的产生，无需再做其它过滤或任何配置。

通过手动添加openflow流规则的方式，实现了vm1🡪ips1🡪ips2🡪vm2，vm2🡪ips2🡪ips1🡪vm1的流量牵引，性能和稳定方面表现良好，未出现虚拟卡死、网络风暴等问题。

### 3.9 知识库模块

知识库的功能包括：获取和维护不同域中各类实体的信息，建立实体间的映射关系和拓扑；收集各实体的日志、报告、相关事件等动态信息；为其他模块（如流量分析等）提供分析处理的上下文知识；为信誉分析、日志分析提供基本数据。

#### 3.9.1 知识实体查询

知识实体（以下简称实体）指的是SC知识库获取的SDN云环境中各类对象的信息，如云环境中的租户、用户、网络、子网、端口、虚拟机，SDN环境中的交换机、网络设备、网络拓扑，安全域中的安全设备、以及实体物理机器、管理系统用户等等。

实体的类型用“域”+“子类型”方式描述，目前，知识库支持下列域中的实体：

|  |  |  |
| --- | --- | --- |
| 域名称 | 类型码 | 备注 |
| 云计算域 | cloud | 与云环境相关的实体 |
| 网络域 | network | 与网络控制器相关的实体 |
| 安全域 | security | 与安全相关的实体，接入SC的安全设备、APP等 |
| 外部域 | ext | 实体机器、文件、URL等等 |
| 认证域 | auth | 认证系统中的用户、通过认证的设备地址（mac等） |

不同类型的实体之间存在对应两种关系：（1）一对一的对应/从属关系，如，一台虚拟机对应到一个端口，网络等，在程序中称为“关联（related）”；（2）一对多的下属关系，如一台交换机实体之下附属有多个端口实体，在程序中称为“附属（affiliated）”。下表展示了不同实体的对应关系：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 域 | 子类型 | 描述 | 对应/从属实体 | 下属实体 |
| 云计算域  (cloud domain) | network | 云网络信息 | tenant（对应） | subnet  router  port  vm |
| **subnet** | **云子网信息** | **network(从属)** | **user**  **router**  **port**  **vm** |
| vlan | 虚拟子网 | network(从属)  vlan\_map(从属) | vifport |
| vlan\_map | 虚拟子网集合 | network(从属) | vlan |
| **router** | **云路由信息** | **network(从属)**  **subnet(从属)** | **port** |
| port | 云端口信息 | network(从属)  subnet(从属)  subnet(从属)  vm(对应相同的固定IP)  network域的device(对应相同的mac地址)  vifport (对应相同mac地址) | vm |
| vifport | 虚拟子网端口信息 | port(对应相同mac地址) | - |
| vm | 虚拟机信息 | network(从属)  subnet(从属)  user(从属)  vm(对应相同的固定IP)  NETWORK\_DEVICE(对应相同的固定IP) | - |
| floating\_ip | NAT绑定的浮动IP | port(对应相同的固定IP)  vm(对应相同的固定IP) | - |
| tenant | 云租户信息 | network（对应） | user |
| user | 云用户信息 | tenant(从属) | subnet  vm |
| 网络域  (network domain) | controller | 网络控制器信息 |  | switch |
| switch | 交换机 | controller(从属) | switch\_port |
| switch\_port | 交换机端口信息 | switch(从属)  network域的device (对应相同的mac地址) |  |
| **NETWORK\_DEVICE** | **网络设备信息** | **port（物理地址相同）**  **switch(从属)**  **switch\_port(对应相同的mac地址)** |  |
| flow\_table | 交换机流表信息 | switch(从属) | flow |
| flow | 流信息 | flow\_table(从属) |  |
| 安全域  (security domain) | ***device*** | ***安全设备*** | ***port（物理地址相同）***  ***vm（物理地址相同）*** |  |
| app | 安全应用 | 未详细设计 | 未详细设计 |
| KNOWN\_VULNERABILITIES | 已知漏洞 | 未详细设计 | 未详细设计 |
| 外部域  (ext domain) | machine | 物理机器 | 未详细设计 | 未详细设计 |
| user | （管理系统的）用户 | 未详细设计 | 未详细设计 |
| file | 文件 | 未详细设计 | 未详细设计 |
| url | URL | 未详细设计 | 未详细设计 |

**知识库实体查询的REST接口**

1. **获取某类实体的列表**

|  |  |
| --- | --- |
| URI | /sc/knowledgebase/{domain}/{type} |
| 功能说明 | 获取某类实体的列表 |

1. **根据ID查询实体**

|  |  |
| --- | --- |
| URI | /sc/knowledgebase/{domain}/{type}/{id} |
| 功能说明 | 获取单个实体 |

1. **查询某个实体的某类关联实体**

|  |  |
| --- | --- |
| URI | /sc/knowledgebase/{domain}/{type}/{id}/related/{related\_domain}/{related\_type} |
| 功能说明 | 获取某个实体的某类关联实体（单个） |

1. **查询某个实体的某类附属实体列表**

|  |  |
| --- | --- |
| URI | /sc/knowledgebase/{domain}/{type}/{id}/affiliated/{affiliated\_domain}/{affiliated\_type} |
| 功能说明 | 获取某个实体的某类下属实体列表 |

查询返回的实体描述信息（JSON）均包含下列信息：

1. 实体类型
2. 实体唯一标识（一般为ID，某些实体为物理地址）
3. 其他属性信息
4. 关联实体列表（键值对形式，键为类型，值为实体ID）

附属实体列表（键值对形式，键为类型，值为实体ID的list）

**知识库服务的远程调用**

如果要在远程的机器（如分布式运行的节点）上调用某个函数以得到结果，可以使用RPC（远程过程调用）机制。

客户端的实现是：封装成了消息调度模块(IEventManager)的接口方法makeRPCCall，其他模块只需调用消息调度器（EventManager）的这个接口方法即可实现RPC。接口方法的参数说明如下：

|  |  |
| --- | --- |
| serviceName | 服务名称（目前只支持KnowledgeManager） |
| methodName | 调用方法名称（目前支持queryEntity/queryRelatedEntity/ isEntitiesRelated） |
| args | 参数（Object对象数组） |

知识库服务方法详细说明：

|  |  |  |
| --- | --- | --- |
| queryEntity接口 | | |
| 参数 | （KnowledgeType）inputEntityType | 实体的类型 |
| （KnowledgeEntityAttribute ）inputAttrType | 实体的属性类型 |
| （String） inputAttrValue | 实体的属性值 |
| 返回 | （Object） | 查询的实体对象 |
| queryRelatedEntity接口 | | |
| 参数 | （KnowledgeType）inputEntityType | 实体的类型 |
| （KnowledgeEntityAttribute ）inputAttrType | 输入实体的属性类型 |
| （String） inputAttrValue | 输入实体的属性值 |
| （KnowledgeType）outputEntityType | 输出实体的类型 |
| 返回 | （Object） | 与输入实体相关的实体对象，若无满足条件的对象则返回null |
| isEntityRelated接口 | | |
| 参数 | （KnowledgeType）typeA | 实体A的类型 |
| （KnowledgeEntityAttribute ）attrTypeA | 实体A的属性类型 |
| （String） attrValueA | 实体A的属性值 |
| （KnowledgeType）typeB | 实体B的类型 |
| （KnowledgeEntityAttribute ）attrTypeB | 实体B的属性类型 |
| （String） attrValueB | 实体B的属性值 |
| 返回 | （boolean） | A、B是否有关系 |

#### 3.9.2 全局流表查询

#### 3.9.2.1当前流全局路径信息查询接口

符合以下的规范格式：

{

"type":<实体类型>

“id”或者"hardware\_address": <唯一标识>,

<其他属性>,

"affiliated": {

<附属类型A>:[<附属实体A1的ID>,<附属实体A2的ID >,……]

<附属类型B>:[<附属实体B1的ID>,<附属实体B2的ID >,……]

},

"related": {

<附属类型C>:<附属实体C1的ID>,

<附属类型D>:<附属实体D1的ID>

}

},

返回的是计算出的Map<String,MatchPath>对象

接口URL: /sc/globalflow/

HTTP方法：GET

响应示例：

{ “status”:”ok”,

“result”:{

"1d70eaa83f28f339deb6aedce933aa60": {

"matchlist": [

{

"wildcards": 2097152,

"inputPort": 0,

"dataLayerSource": "38:aa:3c:f0:53:1e",

"dataLayerDestination": "00:11:22:03:04:01",

"dataLayerVirtualLan": -1,

"dataLayerVirtualLanPriorityCodePoint": 0,

"dataLayerType": 2048,

"networkTypeOfService": "0",

"networkProtocol": 6,

"networkSource": "111.0.0.106",

"networkDestination": "125.39.118.99",

"transportSource": -26263,

"transportDestination": 80,

"networkDestinationMaskLen": 32,

"networkSourceMaskLen": 32,

"match": "00:11:22:03:04:0138:aa:3c:f0:53:1e2048-1-1125.39.118.99326111.0.0.10632080-262632097152"

}

],

"pathlink": [

"00:00:00:1e:08:09:64:36",

"00:00:e0:db:55:1f:99:b7"

]

},

}

}

【查询流表项】

功能：输入要查询的流的字段值，返回符合条件的所有流以及其经过的交换机路径。

接口URL：[/sc/globalflow/](file:///C:\sc\globalflow\)

HTTP方法：POST

请求参数（json）：要查询的流的字段值

{

"dataLayerSource": <source\_mac>,

"dataLayerDestination":<dest\_mac>,

"networkProtocol": <nw\_layer\_protocol>,

"networkSource": <source\_ip>,

"networkDestination": <dest\_ip>,

"transportSource": <source\_port>,

"transportDestination":<dest\_port>

}

响应：符合查询条件的所有流及其交换机路径，格式同【查看所有流表项】接口

备注：

1. 以上字段为“与”关系，若某查询条件为空则不用输入，如查询源IP为192.168.19.1的流，则请求体为：{"networkSource": “192.168.19.1”}。
2. 查询的实现是分两步，一是找到符合查询条件的所有流，第二步是通过这些流的match对应流经的交换机路径。目前暂时用遍历所有全局流表项的方法实现。今后对全局流表程序进行优化时，可考虑增加有利于查询搜索的数据结构。

#### 3.9.2.2 历史流量信息查询接口

API接口：

【请求】

方法：GET

URL：（[]中为可选参数） /sc/globalflow/records[?src\_mac={src\_mac}&

dst\_mac={dst\_mac}&

src\_ip={src\_ip}&

dst\_ip={dst\_ip}&

src\_port={src\_port}&

dst\_port={dst\_port}&

src\_mac={src\_mac}&

\* starttime={starttime}

endtime={endttime}]

其中，starttime, endtime, time均为UNIX时间戳（毫秒数）。

该接口的功能是：查询在starttime和endtime之间（或时间等于time）、符合指定字段筛选条件的历史流量的总字节数和总包数。

【响应】

符合上述条件的历史流量的总packet数和总字节数，格式如下：

{

"status": "ok",

"result": [

{

"totalByteCount": 136061,

"totalPacketCount": 141

}

]

}

## 四、应用案例

安全控制器有松耦合的模块架构和开放的应用接口，可以完成多种复杂的安全保护场景。我们首先介绍一种面向DDoS攻击的安全系统联动防护方案，这种方案是由北向安全应用驱动的；然后介绍另一种由南向安全设备驱动的联动防护方案，可深度检测可疑的攻击，预测可能的攻击。

### 4.1 数据中心DDoS防护方案

在云数据中心中，DDoS攻击是一种比较严重的威胁。在我们的第一种场景中，安全系统联动防护方案是由北向安全应用驱动，将安全策略逐级推送至南向安全设备，即“自北向南”的安全防护机制，如图10所示。以抵御DDoS攻击为例，安全系统的防护步骤如下：

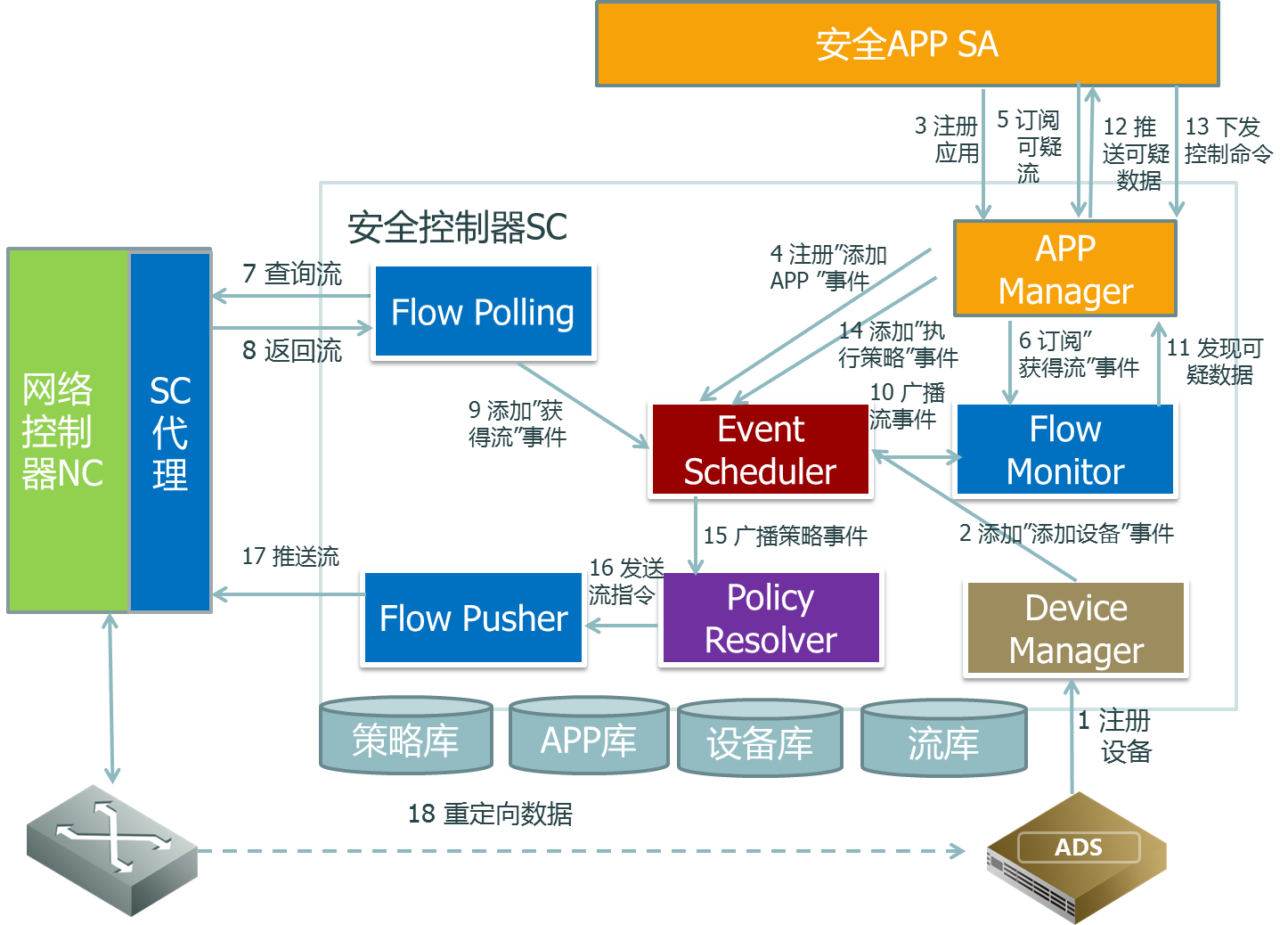


图10 安全应用驱动的安全系统联动防护方案示意图

**(1) 安全设备和应用注册** 南向安全设备ADSDevice在启动时会向安全控制器发送一个注册请求，包含设备编号、设备类型和物理地址，安全控制器的Device Manager模块在设备库中记录这些信息，同时为其添加网络地址，以保证双方可网络通信。

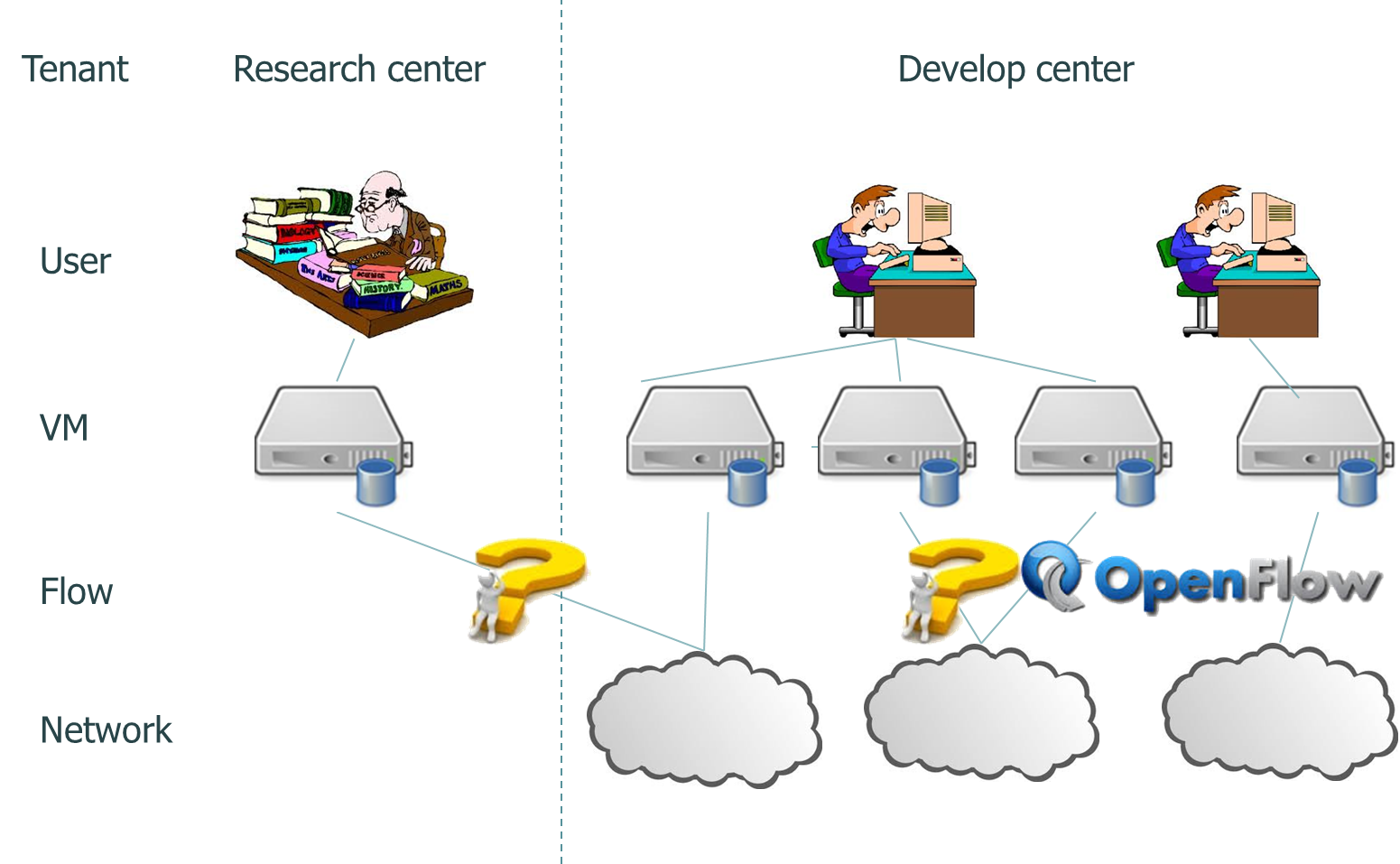
北向安全应用ADSAPP也会在启动阶段向安全控制器发送注册请求，包含应用编号和应用类型，安全控制器的APP Manager模块会在应用库中添加该应用的信息。

**(2) 应用订阅** 很多应用需从安全控制器端获得数据，如ADSAPP定期获取流信息以判断是否存在DDoS攻击，安全应用在注册完毕之后根据需要会向安全控制器发送订阅请求，包含触发事件条件（FLOW\_RETRIEVED）、处理模块（Flow Monitor）、满足条件的数据模式（参见3.4.2）和APP推送URL；安全控制器则在事件调度器中添加相关触发条件，将数据模式通知相关处理模块。

**(3) 流数据获取** 安全控制器的Flow Polling模块定时通过REST方式查询网络控制器，收集所有交换机近期的流信息，后者将流信息保存到内存或数据库中，触发FLOW\_RETRIEVED事件。

**(4) 流数据预处理推送** 事件调度器在收到FLOW\_RETRIEVED事件时，启动Flow Monitor模块的处理机制，该模块根据3.4.2中的订阅条件匹配所有满足条件的流项，组成一个列表；然后将可疑流列表推送到安全应用之前注册的URL。

**(5) 安全应用决策** 安全应用收到可疑流项后，进行更细颗粒度的攻击检测，构建异常行为和攻击模式的知识库，识别已知或未知的攻击。在虚拟化和云环境的场景中，安全应用需要根据可疑数据判断不同级别的攻击行为，最终可根据攻击模式确定不同层面的防护策略。

图11 多租户概念关系图

当前的DDoS攻击，攻击者一般攻击单台服务器，但在虚拟化环境中，由于物理带宽是固定的，攻击者可能会攻击某个租户的多台VM，但是每台VM的流量不会超过既定阈值，所以应对这类攻击，安全APP进行细粒度的安全防护决策时需要考虑虚拟化云环境中多租户的环境需求，这也使得安全判定流程较之传统的防DDoS攻击策略更加复杂。

云计算环境中存在多类需要保护的对象，通常通过网络虚拟平台构建多租户的网络环境可划分不同的级别的主体概念：租户机构Tenant，用户User，主机VM，流Flow、虚拟子网Network和其他逻辑划分，如图11所示。一般而言，一个租户机构下可存在多个用户；一个用户拥有多个VM和多个虚拟子网；一个虚拟子网中部署若干台VM，一台VM会并发产生多条流。不同租户的虚拟子网、VM和流间是隔离的。

在DDoS攻击中，攻击者可能采取两种攻击方式：不伪造源地址和伪造源地址。在第一种场景中，安全应用可根据流信息找到依次定位到相应的VM、用户和租户机构；而在第二种场景中，安全应用获得的是虚假网络地址和虚假物理地址的流信息，无法直接定位到租户，但考虑到OpenFlow中有vlan项，指示了流所属的虚拟子网，每一个子网都是归属于某个租户，所以安全应用可以先后确定流所对应的虚拟子网和租户。

所以两类攻击的需要保护的主体级别分析过程应为：

* 如果数据包源地址是真实的
  + Flow -> VM -> User -> Tenant
* 如果数据包源地址是虚假的
  + Flow -> Network -> Tenant

具体分析过程：TODO

* + 分别在各个维度根据预设阈值（各维度有不同的阈值）来判断是否存在DDoS攻击
  + 合并三个维度上的策略。合并的时候，如果USER已经包含在TENANT策略中，则忽略，否则加入。同理，VM策略也只有在VM不包含在TENANT或USER中时才加入，否则忽略
  + 在判断高维度时，应减去低维度中已判定存在DDOS攻击单元的流量，以避免影响总体的流量统计

**(6) 策略解析** 安全控制器收到ADSAPP下发的策略之后，策略管理模块Policy Resolver根据策略所定的级别和安全控制器所了解的知识将相应的策略解析为网络控制器、网络设备和安全设备可理解的命令，如策略为“将租户A的所有数据进行清洗”，那么解析过程首先就是将租户A解析为A的所有VM，然后寻找离这些VM最近的流量清洗设备ADSDevice，最后组织成“将源或目的为VM的流重定向到ADSDevice”的指令，并将该指令传输给网络控制器。

**(7) 流策略下发** 网络控制器中包含有一个安全控制器的代理Agent模块，用于转换和下发流命令。当Agent收到(6)中的指令后，计算出VM到ADSDevice的路由路径，将所有源或目的为VM的数据都沿着该路径传输，数据经过清洗设备后再经由虚拟交换机传递到网络控制器中，后者最终根据正常的路由策略将清洗过的数据转发。

### 4.2 SDN环境中的安全设备联动检测防护方案

SDN环境对物理安全产品提出的最大挑战是安全产品需要理解SDN中的通信协议、并且能够无缝融入到SDN环境中。然而当前的安全产品大多是基于封闭的硬件的，一方面无法直接部署到虚拟化的场景中，另一方面也无法与网络控制器和安全控制器进行交互，进行安全预警或采取安全行为。本节主要探讨各类安全设备如何应用于SDN环境，并且如何与安全控制器进行交互，实现由安全设备驱动的攻击检测和抵御。

#### 4.2.1 安全设备注册

那么在可预见的未来，安全控制器将会与三类安全设备交互：

(1) SDN-enable安全设备，安全厂商会将自家的安全产品改造，使之支持SDN，garnter预计到2014年，9家TOP10的安全厂商会支持OpenFlow，这类安全产品往往是Fabric的形式，在一台实体设备中可动态启动多个虚拟的安全设备，每个虚拟安全设备都能理解虚拟化环境中的输入数据；

(2) 传统安全设备，很多厂商的安全产品可能无法扩展支持OpenFlow，还有大量的用户需要继续使用之前购买的安全设备。

(3) 厂商安全系统，安全厂商还有自己的安全系统，对其各类安全设备进行管理，所以安全控制器无法与其安全设备直接交互。

所以我们的安全系统既需要支持实体或虚拟的OpenFlow enable的安全设备，还需要兼容传统的安全设备和安全系统。

可行的解决方案是提供开放的应用接口，那么SDN-enabled设备可直接发送附件1中的POST REST请求，自动完成设备注册过程；而对于传统设备，则可由管理员通过脚本，半自动地进行注册过程。对于(3)中间接交互的安全设备，可以使用如下机制实现注册：

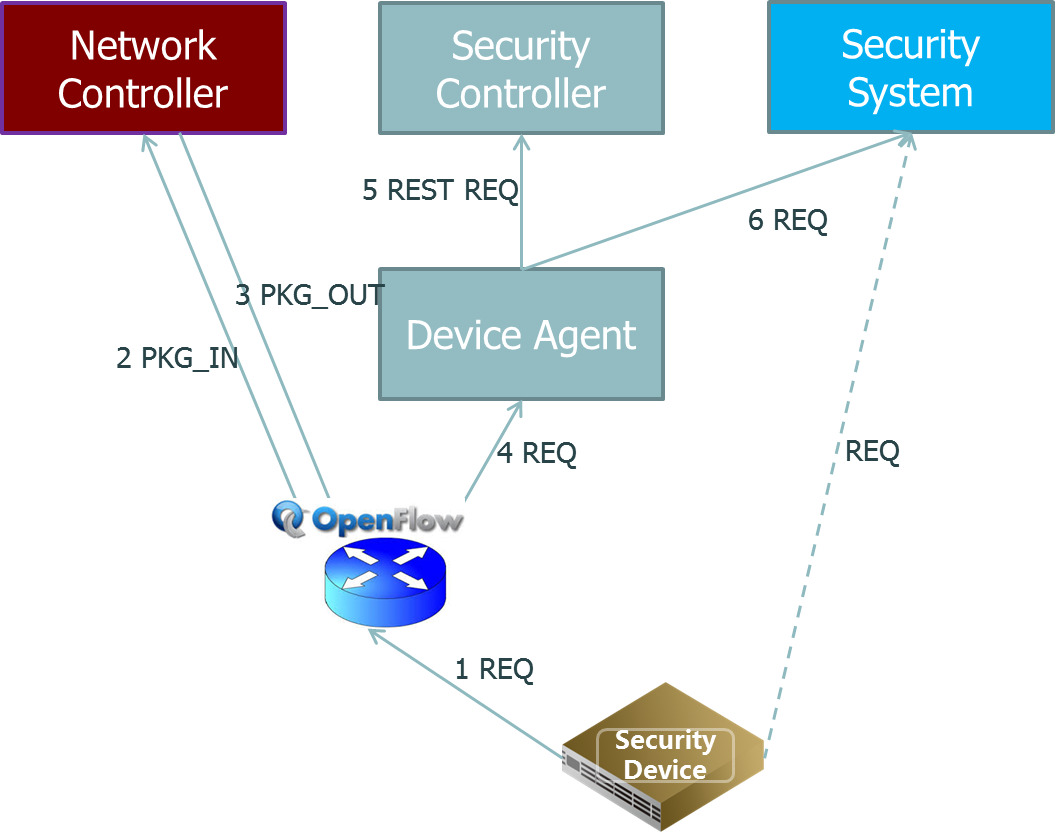


图12 利用SDN实现设备自动注册

**(1) 注册请求** 首先安全设备向安全系统发送加入请求REQ，该请求会经过OpenFlow交换机，此时交换机会向网络控制器发送一个PACKET\_IN的消息，询问如何处理该数据包；

**(2) 重定向请求** 网络控制器收到消息后，内部的安全控制器插件在路由模块之前进行处理，根据目的网络地址和端口可判断该数据包为安全设备加入请求，则将该数据包的路由目的地设置为一个安全设备的代理Device Agent；

**(3) 代理注册** Device Agent代理收到该请求后，一方面将该请求转换为安全控制器发布的标准REST请求，向安全控制器提交一个设备注册的请求，另一方面将原请求正常转发给安全系统。

**(4) 双重注册** 最终安全设备将厂商自定义的设备加入请求发送给安全系统，也完成了在安全控制器端的注册，整个过程对安全设备和安全系统而言是透明的。

#### 4.2.2 安全设备驱动的攻击检测机制

在第四章中，我们给出了一个安全控制器抵御DdoS攻击的实例，安全应用驱动安全控制器获取、检测流信息，最终将可疑流重定向到底层安全设备，此时安全设备是一个被动处理的角色。但是还有很多应用场景，安全设备是主动方，检测可疑数据、发现潜在的攻击，那么安全控制器和安全设备的交互模式有所不同，本节以IDS设备为例，介绍如何抵御相关的攻击，整个攻击检测的流程如图所示。需要说明的是图中出于简化流程的考虑，将事件传递部分的流程省略，所以各模块的通信在图中没有经过事件调度器，而是直接相连。

以安全设备驱动的攻击检测机制主要包括以下步骤：

**(1) 设备注册** 安全设备（如IDS设备、TAC威胁分析中心等）注册过程与第四章中类似，安全控制器在设备库中添加该设备

**(2) 恶意报告推送** 当IDS设备在安全控制器中注册成功后，网络流量在网络控制器的引导下，经过该安全设备。IDS可以实时对所经过的数据包进行分析，当检测到疑似攻击时，向安全控制器的设备管理模块发送样本报告。

**(3) 报告处理** 设备管理模块一方面将样本通过事件调度器发送给应用管理模块，进而推送到异常处理的APP，经由APP进一步审核后进行决策；另一方面将每条报告生成日志，通过日志存储模块存放到日志数据库中。

**(4) 攻击分析** 如果该报告内容在大概率上可被判定为是恶意事件，那么日志监控模块将通过应用管理模块向日志分析APP推送一个分析请求，后者进而在数据库中检查这个报告的上下文，找出所有相关的日志，寻找历史上的攻击行为，并且预测将来可能的攻击，建立攻击者的attack profile。

(5) 策略下发 安全应用根据分析结果生成若干安全策略，控制器的策略解析模块将策略分解为若干策略，下发到安全设备或网络设备。

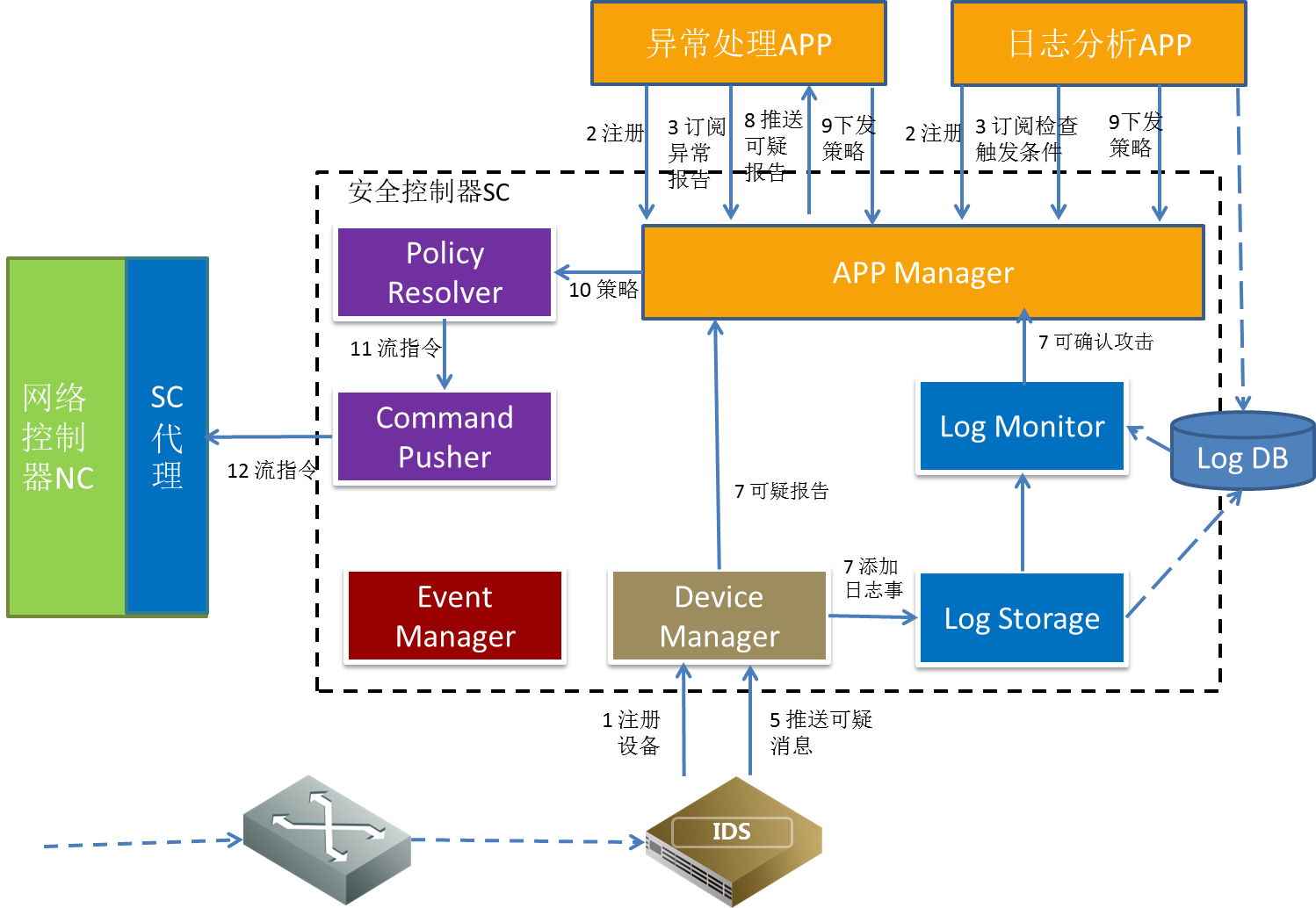
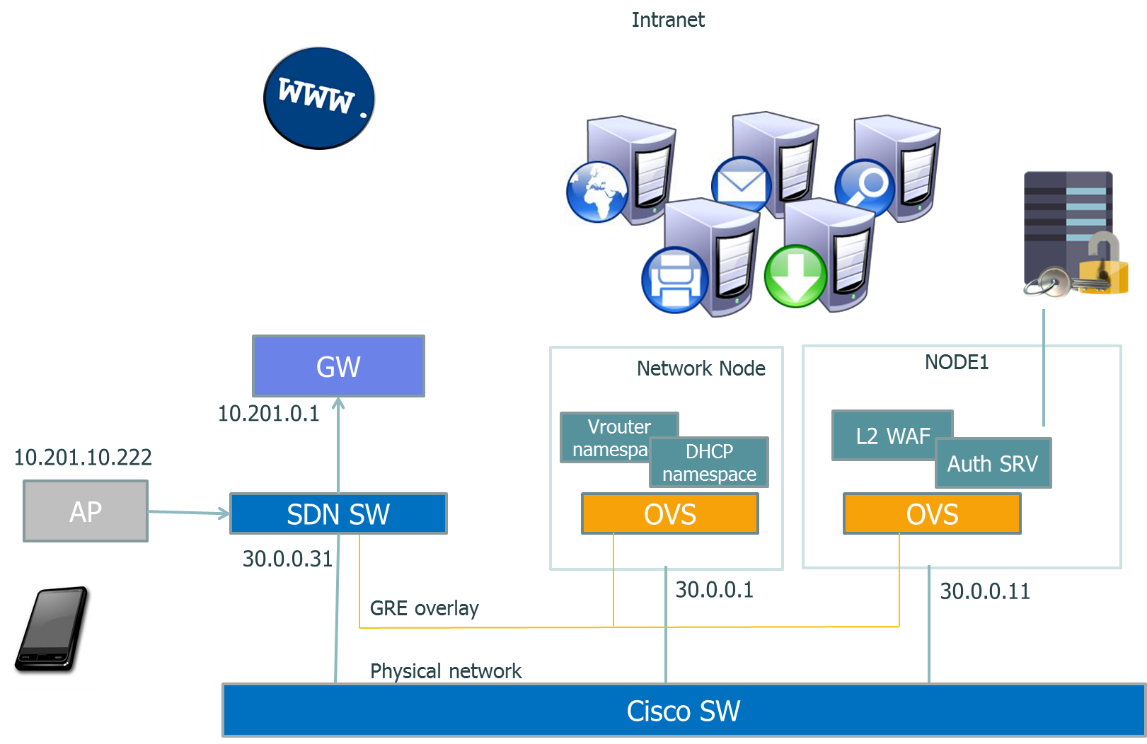


图13 设备驱动的攻击检测示意图

从而，北向的异常处理APP和日志分析APP下发的策略覆盖了历史上受攻击的主体、当前受到攻击的主体和未来可能会受攻击的主体，这些策略经过策略解析模块分解为若干基于流的命令，通过命令推送模块传输到网络控制器，最终下发到转发网络设备中，隔离、引导或进一步分析恶意流量。

### 企业网络中BYOD的访问控制



说明：在由SDN内网、外部网组成的网络环境中，BYOD的功能是使通过无线AP接入SDN内网的设备经过认证后，允许接入外部网。基本流程如下：

**应用初始化：**管理员下发初始化策略给SC，内容为：禁止所有从AP进入的流量，并且对于http流量，将其重定向到BYOD认证服务器上。

**登录：**只允许移动设备的客户端登录（通过检测HTTP头的User-Agent）。在登录页面，用户输入用户名、密码，BYOD Server通过DHCP服务查询接入设备的mac，然后用LDAP服务或用户数据库验证，通过后，向SC下发一条允许策略，。

**登出：**登出时，从DB查询当前用户下发的策略ID，生成撤销策略（内容包括ID）并下发。

BYOD Server(apache)

网络控制器

登出页面

[根据用户名查询策略ID]

登录页面

[检测客户端类型]

[根据IP查mac]

策略REST API

[生成策略，返回策略ID]

策略解析

[冲突检测等]

下发

允许策略

下发

撤销策略

读取

策略

LDAP Server(正式员工)

登录：

用户名，密码

新建命令

用户

命令下发

临时用户数据库(实习生)

DHCP

Server

登录记录DB：

用户-设备mac-策略ID

认证用户名密码

策略DB

登出：

撤销命令

管理页面

下发初始化策略

管理员

#### 4.3.1实现准入的相关初始化流配置

因为无线设备已经配置为桥接模式，相当于一个无线的交换机，所以在实体OF交换机V350上可以获得每一个接入的移动设备的MAC信息，这保证了可以实现基于MAC的接入认证。

首先在AP与v350连接的eth-0-3端口上配置一条优先级低于Floodlight默认流优先级(10)的drop all流（比如priority=1）：

**priority=1,in\_port=3 actions=drop**

当一个新的设备出现时，因为尚没有为其建立其它流，所以它会匹配这一条，数据包被丢弃。

因为要实现Http请求的自动重定向，所以我们首先必须让设备能够获使用DHCP和DNS以及ARP功能，因此添加以下对应的允许流：

**priority=2,udp,in\_port=3,tp\_dst=67 actions=CONTROLLER:65535**

**priority=2,udp,in\_port=3,tp\_dst=53 actions=CONTROLLER:65535**

**priority=2,arp,in\_port=3 actions=CONTROLLER:65535**

最后，需要将http请求重定向到认证服务器，这个过程通过网络控制器中的Agent来实现，添加流：

**priority=2,tcp,in\_port=3,tp\_dst=80 actions=CONTROLLER:65535**

当Agent收到packet in时，会首先查看匹配规则是不是为空，如果空则不对packet in进行处理。当SC向NC发出BYOD\_INIT指令时，除了下发以上初始化流之外，也会在Agent中加入一条匹配规则，指定交换机、交换机端口、网段、协议、传输层端口和认证服务器IP和MAC地址等信息，当有匹配的Packet in进入时，Agent会为其建立一条到达认证服务器，并返回原移动设备的流。因为认证服务器是一个三层设备，所以我们需要对数据包头进行修改。重定向到认证服务器时，需要修改包头的目的MAC和IP指向认证服务器，以使认证服务器能够接收来自设备的http请求。从认证服务器返回的流，需要将源IP修改为移动设备http请求的原始目的IP，以使设备能够接收来自认证服务器的数据。

当认证完成 ，会添加一条允许该MAC的包被发送到控制器的流，其优先级大于之前DROP的流：

**priority=2,in\_port=3,dl\_src=24:69:a5:5e:ae:52 actions=CONTROLLER:65535**

当这个设备再次发送数据时，数据包将首先被以Packet In的方式发送给Controller，然后Controller会自动为其建立相应的流：

**priority=10,tcp,in\_port=3,vlan\_tci=0x0000,dl\_src=24:69:a5:5e:ae:52,dl\_dst=00:11:22:03:04:01,nw\_src=111.0.0.232,nw\_dst=180.149.153.216,tp\_src=43595,tp\_dst=80 actions=output:9**

由于Controller下发的流优先级是10，所以此条流的后续数据包都不会被发送给控制器，直到设备发起新的连接。

#### 4.3.2 相关的接口

将所有初始化的操作都封装成一个INIT命令

##### BYOD\_INIT

{

"type": "BYOD\_INIT",

"commandlist": [

{

"id": "p3",

"commandName": "command3",

"dpid": 128983852086,

"inPort": 3,

"serverIp": "100.0.0.14",

"serverMac": "fa:16:3e:8f:fc:73",

"network": "111.0.0.0",

"mask": 24,

"commandPriority": 2,

"idleTimeout": 0,

"hardTimeout": 0

}

]

}

##### BYOD\_ALLOW

{

"id": "1f0271de-e123",

"type": "BYOD\_ALLOW",

"commandlist": [

{

"id": "62c27921d44c351005366188bfbbb9bb",

"commandName": null,

"commandPriority": 5,

"matchArguments": {

"wildcards": 0,

"inputPort": 0,

"dataLayerSource": "24:69:a5:5e:ae:52",

"dataLayerDestination": "00:00:00:00:00:00",

"dataLayerVirtualLan": 0,

"dataLayerVirtualLanPriorityCodePoint": 0,

"dataLayerType": 0,

"networkTypeOfService": 0,

"networkProtocol": 0,

"networkSource": 0,

"networkDestination": 0,

"transportSource": 0,

"transportDestination": 0

},

"idleTimeout": 0,

"hardTimeout": 0,

"dpid": 128983852086,

"inPort": 3,

"flowName": null

}

]

}

##### RESTORE\_BYOD\_ALLOW

{

"id": "eb9b5340-9157-4fe5-9d02-86850b4c9502",

"type": "RESTORE\_BYOD\_ALLOW",

"commandlist": [

{

"id": "d6187ba60044b0657014f150d9620092"

}

]

}

## 六、实验结果

我们在SDN环境中设计了安全控制器抵御DDoS攻击的实验，在大量的实验中整个过程中系统的正常运作，验证从检测DDOS攻击到策略生成和下发过程，记录了相关的实验数据，并针对实验结果对相应模块作出调整和改进，以达到最优效果。

### 6.1 实验设置

如下图所示，整个实验是在设施虚拟化平台Openstack上进行，环境中存在两台安装64位Ubuntu 12.04.2 LTS系统的服务器（node1、node2），分别作为控制节点和计算节点，主要硬件均为Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz x 24 + 132GB RAM。实验所有虚拟机（VM、安全设备adsdevice）均运行于Openstack之上，底层openflow-enable的交换机为openvswitch（图中br代表虚拟交换机），SDN网络控制器为Floodlight，两台物理主机的虚拟网络通过GRE隧道连接。adsapp为检测DDoS的安全应用。

实验开始时，由某租户的一台虚拟机利用DDoS生成工具PenTBox向另外一台虚拟机VM的80端口进行DDoS攻击，安全控制器检测到可疑流（即符合adsapp订阅条件的流）之后会将其发送至adsapp；adsapp如果确定存在DDoS攻击，将生成相应的策略发送给安全控制器，安全控制器收到策略后将其解析为网络控制器Floodlight能识别的格式，交由Floodlight下发实施。策略实施后将重定向被攻击对象的流量到安全设备adsdevice进行清洗，最后将清洗后的流量返回原对象。

node2

node1

br-tun

br-out

br-in

br-tun

br-int

GRE tunnel

VM

adsdevice

Floodlight

……

Security Controller

adsapp

图14 实验环境设置示意图

下图为3个物理节点组成的Openstack的虚拟网络连接情况。

br-tun

br-tun

br-tun

gre

gre

gre

br-int

br-int

br-int

patch

patch

patch

连接虚拟机、网关等

图15 Openstack节点部署示意图

安全设备可能同时部署于多个物理节点之上，但是它们始终都于连通的虚拟（二层）网络之上。

### 6.2 抵御DDoS攻击性能

在功能方面，经过多次测试，系统整体能够顺利完成对DDoS攻击的检测、策略生成和策略实施等所有预期功能，确实起到了抵御DDoS攻击的效果。

在性能方面，实验主要对时间系统进行了统计，包括整体（从获取攻击流到策略实施完成）和各模块的处理时间，稍后给出的数据为最终修改版本的测试数据，数值是多次测试后所取平均值。

我们对两组设定的DDoS攻击进行了测试，攻击工具为PentBox。第一组，使用真实源地址，流量检测间隔3秒，平均流数量为18544条，总用时为3.9秒；第二组，使用伪造的源地址，流量检测间隔5秒，平均流数量为30868条，总用时为8.0秒。

表1和图15列出了两组实验的详细数据，其中FlowPolling是将相邻两次的获取的流进行比对的，FlowMonitor模块检测流是否符合adsapp订阅条件，adsapp对数据进行处理并生成相应的策略的所需时间，Transport代表订阅的流由安全控制器发送至adsapp的传输时间（第一组中该值已合并到adsapp，未单独统计），最后Push flow表示策略下发所需时间。

表1 抵御DdoS攻击安全方案的模块耗时表

(a) 使用真实源地址的模块耗时(s) (b) 使用伪造源地址的模块耗时(s)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Total** | **00:03.9** | | FlowPolling | 00:00.3 | | FlowMonitor | 00:01.3 | | adsapp | 00:01.8 | | Push Flow | 00:00.1 | | | |  |  | | --- | --- | | **Total** | **00:08.0** | | FlowPolling | 00:00.6 | | FlowMonitor | 00:02.7 | | Transport | 00:01.5 | | adsapp | 00:03.0 | | Push flow | 00:00.2 | |
|  | |  |
|  |  | |

(a) 使用真实源地址的模块耗时(s) (b) 使用伪造源地址的模块耗时(s)

图15抵御DdoS攻击安全方案的模块耗时对比图

相对其它类型攻击，DDoS攻击会产生大量流，所以传输和处理全部流信息比较费时，系统能分别在3.9秒和8.0秒的时间内完成对攻击的检测与阻止，整体结果可以接受。而且一旦策略下发以后，交换机就不再会出现大量流，安全系统的性能得到保证。

就模块耗时而言，除传输时间，主要开销集中在FlowPolling、FlowMonitor和adsapp三个处理模块，这三个模块共同的特点是需要对大量的流进行处理，时延较大。就两个分组的实验结果而言，两者的差异也主要是由于流数量的不同而导致的，证明系统的瓶颈在于对大量流的处理效率，系统架构的设计是合理的。

当安全决策由安全控制器的FlowMonitor模块决策，可进一步减少防护时间，图4描绘了整个防护工作流，其中*t*0为从流获取完毕到命令下发的时间间隔，即检测时间开销。[5]已研究了流传输开销影响，故此处略去，仅考察安全控制器的性能。首先计算*t*0的累积概率，所得的防护延迟如图5所示。攻击者分别以每台VM约4000新流/秒和8000新流/秒的速度发动两次SYN flooding攻击。

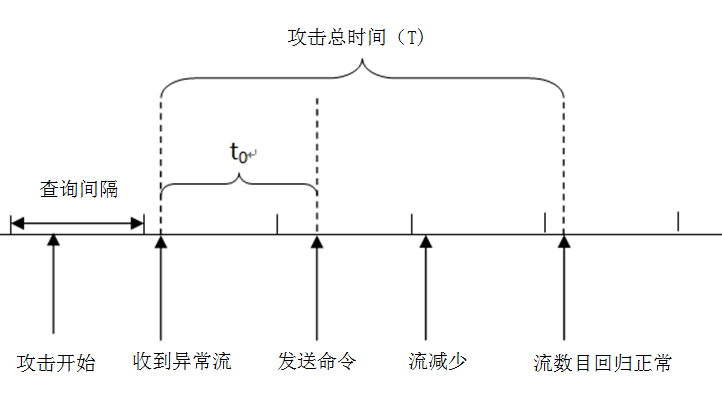


图4 流检测防护工作流程

实验表明当流建立速度为4000新流/秒时，安全控制器的处理时延为1-1.5秒，当流建立速度为8000新流/秒时，时延增加为1-2.2秒。而流检测总体相对较快。

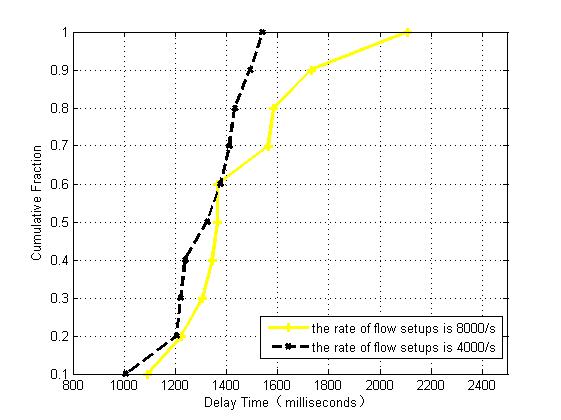


图5 处理延迟的累计概率函数

### 6.3 端口扫描检测

端口扫描是一种重要的事先攻击手段，通常有两个特征：

1) 流数量剧增，多数有相同的源地址

2) 大量失败响应，多数扫描请求会失败，导致大量如TCP RST、ICMP不可达等响应。

传统IDS根据固定模式匹配数据包字段，有较高误报率，很多同样使用TRW算法分析连接、端口和成功失败流。本文的架构可使用全局视图和流特征分析可不依赖局部安全设备而得到高准确率。

当收到全局流集合{*f*}后，流检测模块在式(1)中计算流的存活目的端口数*A*，在式(2)中计算未建立流比例*U*：

*A* = count({*obj.dst\_port*!=*ref.dst\_port* AND *obj.dst\_addr =ref.network\_addr*}) (1)

*U* = count(*obj.dst\_addr*=*ref.dst\_addr* AND (( *object.pkg\_count*<3 AND *obj.protocol*=TCP) or (*object.pkg\_count*<2 AND *obj.protcol*=UDP)))/count(*obj.dst\_addr*=*ref.network\_addr*) (2)

其中*ref*为{*f*}中的任意一条参考流，*obj*是{*f*}中其他任意一条流。

实验中使用Nmap 5.21发起了TCP扫描和TCP SYN扫描，速率分别为246.76 端口/秒和313.33端口/秒，并与正常场景比较，结果如表1所示。实验中有3000条流的背景流量，流获取与流计算间隔为3秒（与OpenFlow的idle\_timeout项一致）。两种扫描场景中，活跃端口和未建立流比例远比正常场景多，因而可设阈值*A*max=500 和 *U*max =90%，安全控制器可快速检测出扫描行为。

表1 常规端口扫描结果

|  |  |  |  |
| --- | --- | --- | --- |
| 扫描类型 | 存活端口*A* | 未建立流比例 *U* | *I* |
| TCP 扫描 | 735 | 98.28% | 4.573 |
| TCP SYN 扫描 | 970 | 98.22% | 6.023 |
| 正常场景 | 3 | 19.15% | 0.026 |

然而如果攻击者使用表2中的慢扫描，如2端口/秒，活跃端口和未建立流比例分别降到208和11.16%，特征不太明显。但由于安全控制器的知识库有全局和历史流视图，那么在较长时间（如180秒）后计算，该特征还较为明显。记*I*为式(3)中的累积结果，其中*Ai*和*Ui*为扫描期间每次计算的活动端口数和未建立流比例，*C*和*R*为其阈值，*n*为计算次数， 为*Ui*权重，*α*为归一化系数，实验中*C*=150，*R*=0.9，*α*=0.3。在两次常规扫描和慢扫描中，可观察到*I*值均远大于正常流量。

(3)

表2 慢扫描结果

|  |  |  |  |
| --- | --- | --- | --- |
| 扫描类型 | 存活端口 *A* | 未建立流比例 *U* | *I* |
| 慢速SYN扫描 | 208 | 11.16% | 1.384 |
| 正常场景 | 3 | 1.6% | 0.02 |

接着比较安全控制器与Snort的扫描检测效率，尽管Snort有若干条针对扫描的规则，但实验中发现它检查Nmap扫描时仅产生了DoS误报，故又选择Snort的sfportscan预处理插件，结果如表4所示，Snort的对每个包的检查开销变化不大（约2μs/pkt），作为对比，安全控制器的流监控模块大约平均节省了95%的时间，因其只检查流而非包细节，一条流覆盖越多数据包，则总体检测开销就越小。

表3 检测开销对比

|  |  |  |  |
| --- | --- | --- | --- |
| 扫描开销（μs/pkt） | TCP SYN扫描 | TCP扫描 | 慢扫描 |
| 流检测模块 | 0.1356 | 0.1620 | 0.0013 |
| Snort插件 | 2.00 | 2.02 | 2.34 |

安全控制器需要同时处理所有交换机的当前所有流，因而可能使整个网络过载，本文架构使用了分布式实时处理库Storm，如图6所示，其中流监控模块部署在多个bolt节点上，每个节点处理同一交换机上同样目的端口的流。实验中该机群处理了9个交换机的流，我们在图7记录了分别部署1-6个bolt节点的处理时间，每次实验对比了约1400条和4000条流规模的场景，主要研究流数量与并行处理效率的关系。当计算集群规模变大时，总体的处理时间显著下降。但在一个规模确定的网络中，机群规模进一步扩展不能导致整体处理时间持续降低。图中可见机群规模大于3后并不线性减少，原因是交换机负载不均和任务不可分的特点：扫描期间一个bolt节点经常处理4000多条流数据，而大多数bolt节点仅处理不足100条流，所以总时长取决于负载最重的节点，即便增加集群规模也不会进一步减少整体时间开销。总体而言，在大规模网络中，分布式处理的优点是显著的，检测流程架构上也是可扩展的。



图6 分布式流处理结构

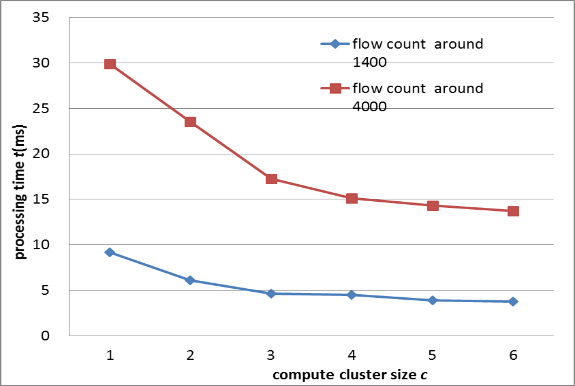
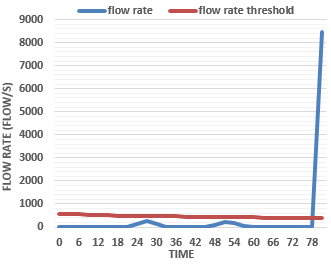
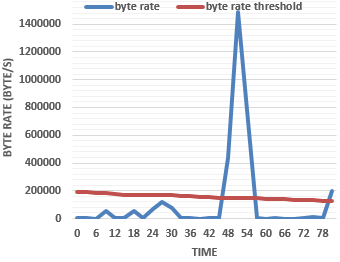
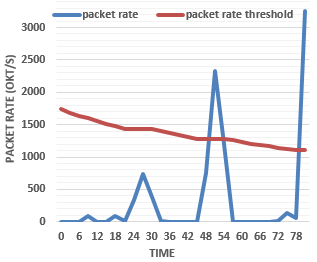


图7 机群处理流时间对比

### 6.4 大流量预警

近期很多针对社交网站的“拖库”攻击中，攻击者窃取大量敏感数据，此外攻击者除了用僵尸网络等传统手段外，也开始租赁云服务对云环境内的资源进行DDoS攻击，这些攻击或绕过常规检查，或隐藏于虚拟网络中，传统安全检测很难及时发现并阻断。本实验目的是对可以对物理和虚拟网络设备上的大数据流产生告警，如此可进而经由安全控制器的决策模块进行检查、隔离或阻断。当获取到流统计信息后，流监控模块计算流数量*FC*、包数量*PC*和总字节数*BC*，并更新这些变量的历史平均值，当超过阈值则触发大流量告警。

在图7中依次生成三种流：(1) 大量正常的HTTP请求；(2) 大量高并发的HTTP请求（由Apache Bench产生，包括500个总请求和100个并发请求，48~54秒)；(3) TCP SYN floodling流量(由PentBox生成，7000 pkts/s, 40 bytes/pkt, 75~81秒)。



(a) flow rate variation (b) packet rate variation (c) byte rate variation

图7 处理延迟的累计概率函数**.**

当在正常场景中，所有三个指标均缓慢增长，没有超过阈值；当产生高并发的HTTP请求时，除*FC*外，*PC*和*BC*均剧增超过阈值；在TCP SYN攻击中，*FC*和*PC*剧增，但*BC*没有过大变化。在两个异常场景汇中，安全控制器均观察到了流统计信息的异常变化，从而产生告警，在大型的网络环境中，只需设定三个指标的阈值，就可以实时发现全局范围内的窃取数据或内部网络资源滥用的可疑行为。

## 七、结论

SDN环境中的安全挑战有很多，建立安全控制器、安全设备和安全应用联动的系统是一种有前景的安全解决方案。本文介绍了一种基于安全控制器的安全防护体系，松耦合模块和开放的应用接口使得安全控制器可以适应多种场景，抵御不同类型的攻击。实验结果显示这种分布式的安全控制器-安全应用联动的方案具有较好的性能和保护能力。

## 参考文献

[2] Garnter, The Impact of Software-Defined Data Centers on Information Security

Published: 16 October 2012

[1] 左青云,陈鸣,赵广松,邢长友,张国敏,蒋培成.基于OpenFlow的SDN技术研究.软件学报,2013,24(5):1078-1097

[3] Qazi Zafar Ayyub, Tu, Cheng-Chun. Chiang, Luis and Miao, Rui, Sekar, Vyas, Yu, Minlan, SIMPLE-fying Middlebox Policy Enforcement Using SDN, sigcom 2013

[3] Radware Unveils Comprehensive SDN Strategy and Introduces New SDN Application: DefenseFlow, April 2013

[4] Syed Akbar Mehdi,Junaid Khalid,Syed Ali Khayam, Revisiting traffic anomaly detection using software defined networking, Proceeding RAID'11 Proceedings of the 14th international conference on Recent Advances in Intrusion Detection, 161-180,2011

## 附录1 模块服务接口

安全控制器和安全应用均提供了基于资源的REST服务，调用方只需对资源采用GET、POST、PUT和DELETE四种操作，即可完成各种复杂的业务功能。附录中提供了各个模块的接口说明。

1) 设备管理模块DeviceManager

|  |  |  |  |
| --- | --- | --- | --- |
| URI | Description | Method | Arguments |
| /sc/device/ | 注册新设备 | POST | name，mac，ipaddr，tap |
| /sc/device/{id} | 获得设备信息 | GET |  |
| /sc/device/{id}/ | 更新设备 | PUT | name，mac，ipaddr，tap |
| /sc/device/{id}/ | 删除设备 | DELETE |  |

2) 应用管理模块AppManager

|  |  |  |  |
| --- | --- | --- | --- |
| URI | Description | Method | Arguments |
| /sc/app/ | 注册新应用 | POST | name，category，type |
| /sc/app/{id} | 获得应用信息 | GET |  |
| /sc/app/{id} | 更新应用 | PUT | name，category，type |
| /sc/app/{id} | 删除应用 | DELETE |  |
| /sc/app/{id}/subscription | SAPP向SC提交/更新可疑数据订阅，详见3.4.1节 | POST/PUT | Id：订阅消息ID  Module：监听模块  Eventtype：触发检查事件类型  Eventsubscription：匹配模式  appid：订阅app的id  url：获得数据后推送的url |
| /sc/app/{id}/command | 安全应用向SC下发安全指令，详见3.4.3节 | POST | Appid：安全应用ID  Policy：下发策略 |

3) 日志管理模块LogManager

|  |  |  |  |
| --- | --- | --- | --- |
| URI | Description | Method | Arguments |
| /sc/device/{id}/suspiciousdata | 设备向SC推送检测到的可疑数据 | POST | date，sd\_id，sd\_ip，target\_id，target\_ip，attacker\_id，attacker\_ip, attacker\_details, target\_bytes, attacker\_bytes, SuspiciousDataType |

4) 云平台信息管理模块CloudAgent

|  |  |  |  |
| --- | --- | --- | --- |
| URI | Description | Method | Arguments |
| /sc/cloudagent/list/{item}/json |  | GET |  |
| /sc/cloudagent/{id} |  | GET |  |

5) 策略管理模块PolicyResolver

|  |  |  |  |
| --- | --- | --- | --- |
| URI | Description | Method | Arguments |
| /sc/app/{id}  /command | 安全应用向SC推送安全策略，详见3.4.3节 | POST | Appid：安全应用ID  Policy：下发策略 |

安全应用SAPP也可以提供自己的服务接口，以Anti-DDoS服务为例，可有如下接口：

|  |  |  |  |
| --- | --- | --- | --- |
| URI | Description | Method | Arguments |
| /app/ads/{op}/{obj}/json | 应用状态查询接口 | GET | op：status/enable/disable |
| /app/ads/securitydata | 接收SC推送的安全数据，详见3.4.2节 | POST | 流数据 |
| /app/ads/suspiciousdata | 接收SC推送的可疑日志 | POST | 日志信息，详见附录2 |

## 附录2 可疑日志格式

LogGettingService模块通过API /sc/device/{id}/suspiciousdata和底层设备进行交互，底层设备通过POST一定格式的可疑数据供该模块生成安全日志。同时也定义了内部方法insertLogtoDB供其他类/模块进行调用。其接口格式定义如下：

底层安全设备（SD）向安全控制器（SC）发送可疑数据消息格式，如下所示，其中security device（sd）为生成日志来源的安全设备，source（s）为可疑数据访问的目的

服务器/主机，c为访问者：

req.json

{

"date":"Wed Jul 31 09:27:50 CST 2013",

"sd\_id":"",

"sd\_ip":"",

"s\_sitename":"", //客体名称

"s\_ip":"",

"s\_port":"",

"c\_name":"", //主体

"c\_ip":"",

"cs\_method":"", //数据提交方式

"cs\_uri":"", //访问地址

"sc\_status":"", //响应状态

"sc\_bytes":"", //服务端发送到客户端数据字节大小

"cs\_bytes":"", //客户端发送到服务端数据字节大小

"sdtype":"DDOS(ping\_of\_death, teardrop, UDP\_flood, SYN\_flood, Land, Smurf, Fraggle)

|USINGATTACK(password\_guessing\_attack, Trojan, buffer\_overflow)

|INFORMATIONCOLLECTIONATTACK(scan, using\_information\_service, Finger, LDAP, Sniffer)

|FAKEINFORMATIONATTACK(spoofed\_email, DNS\_cache\_pollution)" //攻击方法

}

SC的日志模块接收到数据后向事件队列中添加RECEIVED\_LOG事件，将该事件参数（即可疑数据日志）发送到监听该事件类型的LogMonitor模块进行处理。

## 附录3 安全策略模型定义

SAPP(security app)向SC(security controller)发送策略消息，格式如下。策略中“subject”表示主体，“actiontype”表示谓词，“objectlist”表示客体。Objectlist由多个Object组成，代表着多个需要处理的流对象。每个Object通过name，aimtype，id标识了其维度，流向以及流的唯一标识。因此在安全控制器的策略转义模块中需要将每个Object根据OpenstackClient生产的不同维度的映射表定位到某一条具体的流。

req.json {

"appid":"xxxx",

"url":"http:securitycontroller.expr.nsfocus:8888/flow",

policy:{

"actiontype":"CLEAN\_FLOW",

"subject":"SECURITYCONTROLLER",

"negated":"",

'objectlist':

[

{

'name':'tenant|user',

'objectId':'XXXXX',

'valuetype':'String'

},

{...},...]

}

}