SNORT原理简介与优化及GNORT初探

刘斐然

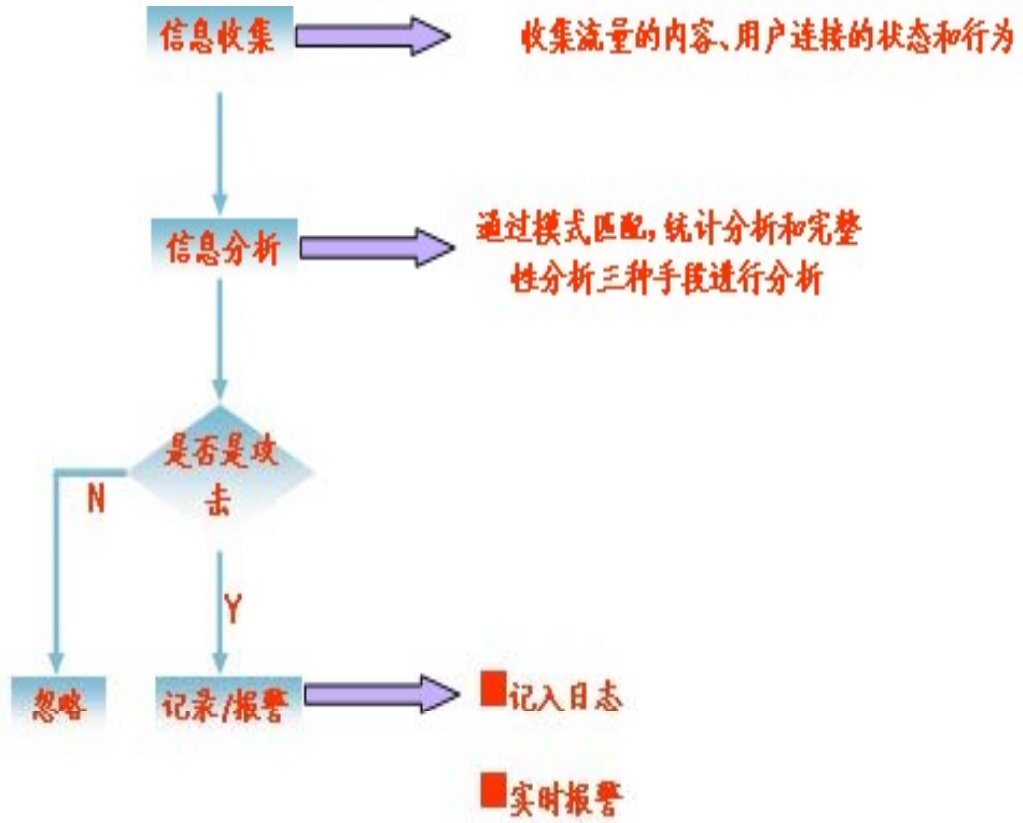
主要内容

- Snort原理是什么？
- Snort在实际应用中的缺陷有哪些？
- 如何对Snort进行优化？
- Gnort初探。

入侵检测系统的基本结构

入侵检测系统通常包括三功能部件：

- 信息收集，其来源如下：
  - 系统或网络的日志文件
  - 网络流量
  - 系统目录或文件的异常变
  - 程序执行中的异常行为
- 信息分析
  - 模式匹配
  - 统计分析
  - 完整性分析
- 结果处理
  - 对异常进行记录/报警等



入侵检测系统分类

入侵检测系统	分 类 依 据	入侵检测系统类别	备　注
	检测方法	基于行为的入侵检测系统	也称异常性检测
		基于入侵知识的检测系统	也称滥用检测
	被保护的目标系统	基于主机的入侵检测系统	主机环境适用
		基于网络的入侵检测系统	适用于网络环境
	分析数据源	主机系统的审计迹、系统日志等	根据分析数据源可分为针对不同分析数据源的入侵检测系统。
		网络数据报、网管信息	
		应用程序的日志	
		其它入侵检测系统的报警信息	
	响应方式	主动的入侵检测系统	对检测到的入侵进行主动响应、处理。
		被动的入侵检测系统	对检测到的入侵仅进行报警

Snort--基于特征检测的NIDS

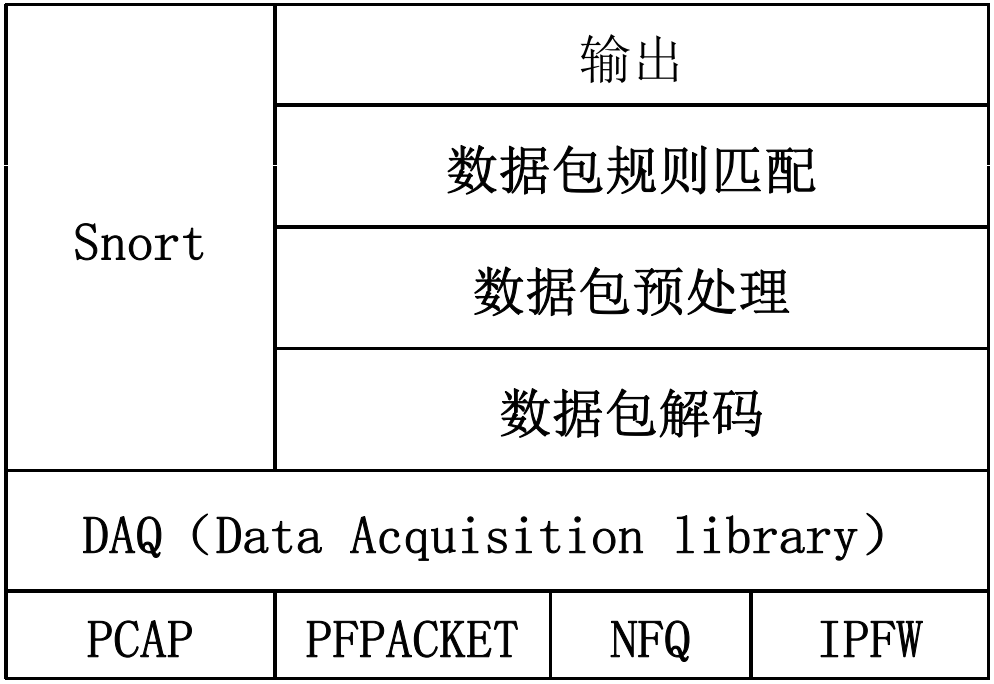
- 在1998年,Martin Roesch先生用C语言开发了开放源代码(Open Source)的入侵检测系统Snort.直至今天,Snort已发展成为一个多平台(Multi-Platform),实时(Real-Time)流量分析,网络IP数据包记录等特性的强大的网络入侵检测/防御系统(Network Intrusion Detection/Prevention System),即NIDS/NIPS.

- Snort有三种工作模式：嗅探器、数据包记录器、网络入侵检测系统。
  - 嗅探器模式仅仅是从网络上读取数据包并作为连续不断的流显示在终端上。
  - 数据包记录器模式把数据包记录到硬盘上。
  - 网路入侵检测模式是最复杂的，而且是可配置的。我们可以让snort分析网络数据流以匹配用户定义的一些规则，并根据检测结果采取一定的动作。

Snort的结构组成

Snort主要包含以下模块：

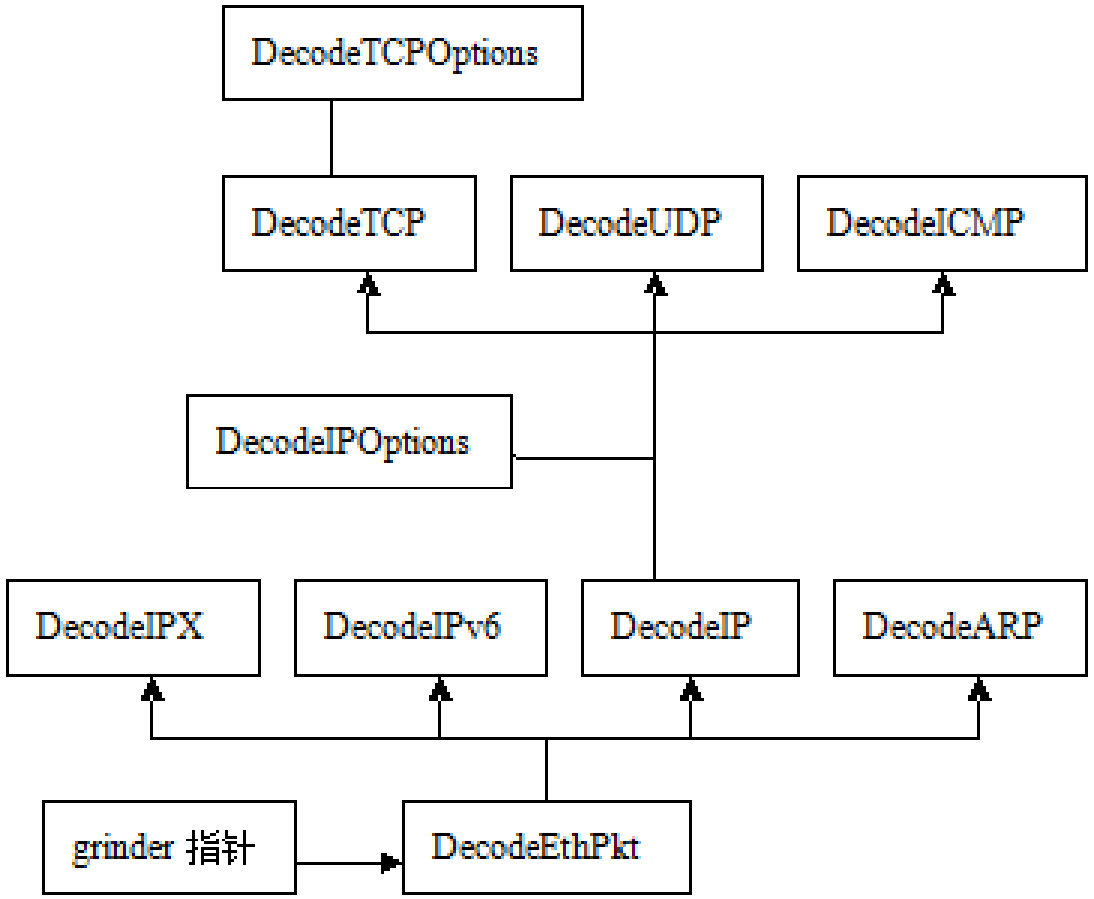
- 数据包捕获模块：
  - 通过DAQ调用底层函数库，捕获来自网络的数据包
- 解码模块
  - 对捕获的数据包进行协议解码
- 预处理模块
  - 以插件形式存在，对IP分片进行重组，防止ARP欺骗等等
- 规则匹配模块
  - 根据设置的规则对数据包进行匹配
- 输出模块
  - 以插件形式存在，当匹配成功时进行记录或报警



Snort原理分析--数据包解码

- 在Snort内部用Packet数据结构表示一个数据包。数据包解码模块负责根据捕获到的数据包初始化Packet数据结构。（该结构定义在decode.h中）

- 数据包解码函数均定义在decode.c中，由grinder指针指向其入口函数。解码流程如右图所示。最后生成完整的Packet结构。

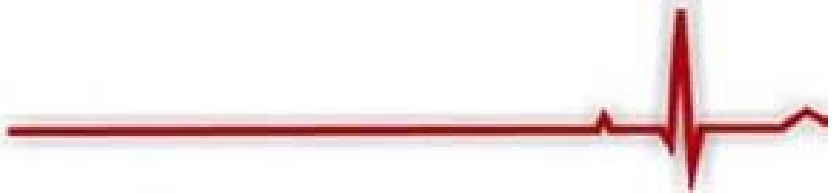


以太网解码流程

Snort原理分析--数据包预处理简介

- 解码后的数据包还需经过预处理才能被主探测引擎进行规则匹配。预处理器的主要用来应对一些IDS攻击手段。其作用包括：
  - 针对可以行为检查包或修改包，以便探测引擎能对其正确解释。
  - 负责对流量标准化，以便探测引擎能精确匹配特征。

- 目前已知的IDS逃避技术主要有：
  - 多态URL编码；
  - 多态shellcode；
  - 会话分割；
  - IP碎片；



Snort原理分析--预处理器简介

Snort主要包含以下预处理器：

- 包重组预处理器：
  - Frag3：IP分片重组和攻击监测。
  - Stream5：维持TCP流状态，进行会话重组。
- 协议规范化预处理器：
  - HttpInspect：规范HTTP流，如将Unicode或hex翻译成snort可以识别的字符集。
  - RpcDecode：规范RPC调用。
- 异常检测预处理器：
  - ARPspoof：检测ARP欺骗。
  - SfPortscan：检测端口扫描。

- 在Snort中，捕获的数据包要经过所有已经打开的预处理器。预处理器是由插件来实现的，这样的好处为：
  - 可以根据实际环境启动或停止一个预处理插件，提高Snort效率。
  - 可以灵活添加自己编写的预处理插件，预防新型的IDS逃避手段。（如何编写预处理插件在源码doc/README.PLUGINS中有初步介绍）

Snort原理分析--预处理器初始化

Snort预处理器初始化流程如下：

- 首先调用RegisterPreprocessors函数，将所有预处理插件注册进全局preproc_config_funcs链表中，内容包括：关键字及相关预处理插件的初始化函数等。
- 调用ConfigurePreprocessors函数，根据配置文件和命令行参数遍历preproc_config_funcs链表，比对关键字。如相同则运行预处理插件的初始化函数。
- 预处理插件的初始化函数通过调用AddFuncToPreprocList将其实际的预处理函数添加到preproc_eval_funcs链表中，插入时会根据预处理插件的priority进行排序。

- 当程序捕获到数据包时，首先通过grinder指向的函数进行解码。然后执调用preproc_eval_funcs链中的预处理程序进行预处理。

Snort原理分析--规则简介

- Snort规则分为Rules Headers和Rules Options，格式为：
  - *rule headers （ rule options ）*

- 规则头(Rules Headers)
  - 包含结果处理模式，协议，来源与目地的IP地址，来源与目地的端口等。

- 规则选项(Rules Options)
  - 包含匹配内容，匹配位置，报警信息，引用说明等等。

例：alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-PHP Opt-X header.php remote file include attempt"; flow:to_server,established; content:"/header.php"; nocase; http_uri; content:"systempath=";.....)

Snort原理分析--规则表现形式

- 在Snort中，规则头由RuleTreeNode(RTN)结构表示，规则选项由OptTreeNode(OTN)表示。

- RTN结构中主要包括：
  - IpAddrSet *sip，*dip：源IP地址，目的IP地址。
  - PortObject *src_portobject，*dst_portobject：源端口，目的端口。
  - OptTreeNode *down：指向相关的OTN。
  - RuleFpList *rule_func：规则匹配函数。

- OTN结构中主要包括：
  - OptFpList *opt_func：规则匹配函数。
  - void *ds_list[PLUGIN_MAX]：插件需要的数据结构指针。
  - RuleTreeNode **proto_nodes：指向其关联的RTN。

Snort原理分析--RTN初始化

- RTN及OTN均由ParseRule负责初始化。

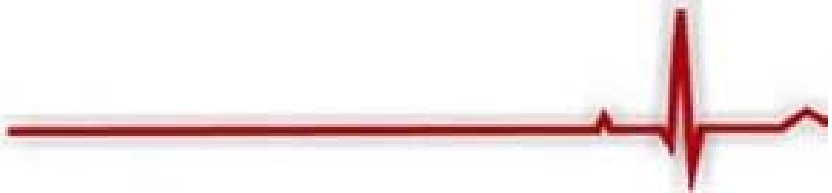
- 首先读入一行规则，传给ParseRule函数进行分析。ParseRule函数会将规则头取出并进行拆分，然后执行如下函数对RTN进行初始化：
  - ProcessIP函数：将源IP地址，目的IP地址写入RTN的*sip，*dip中。
  - ParsePortList函数：将源端口，目的端口写入RTN的*src_portobject，*dst_portobject中。
  - ProcessHeadNode函数：调用SetupRTNFuncList函数，将具体的规则匹配函数插入rule_func链表中，其函数根据标志位不同可选用如下几个：
    - CheckSrcPortNotEq/CheckSrcPortEq/CheckDstPortNotEq/CheckDstPortEqual
    - CheckSrcIP/CheckDstIP
    - RuleListEnd：rule_func的链结尾函数

- 匹配RTN时会依次执行rule_func链表中的函数进行匹配，如匹配不成功则返回0。最终如果执行到RuleListEnd函数则表明匹配成功。

Snort原理分析--OTN初始化

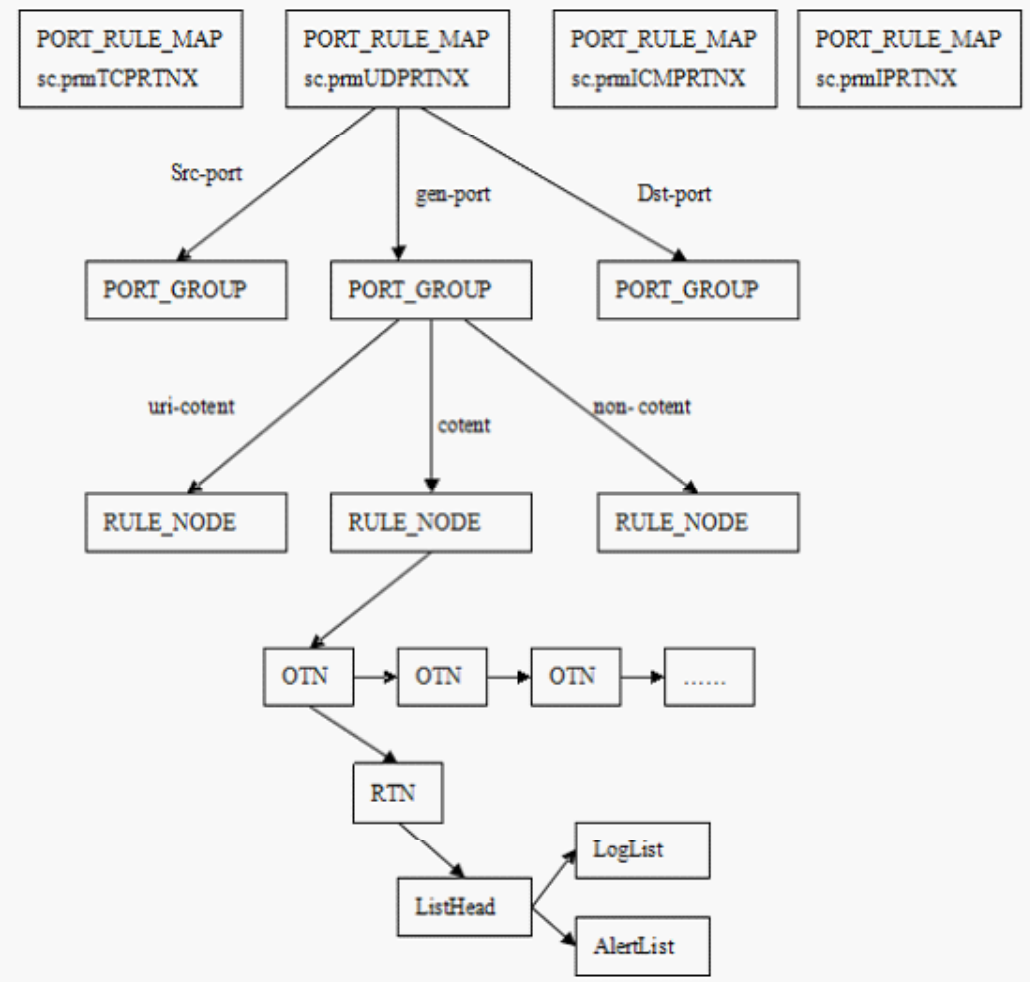
- 与检测RTN不同，系统检测OTN时会用到规则检测插件（例如针对TCP flags的匹配的检测是由TCPFlagCheck插件中的CheckTcpFlags函数来完成的），所以先介绍一下规则检测插件的初始化：
  - 通过调用RegisterRuleOptions函数，将所有的规则检测插件添加到rule_opt_config_funcs链表中，包括关键字与具体的检测初始化函数。

- OTN初始化是由ParseRuleOptions函数完成的。其执行如下操作：
  - 调用addRtnToOtn，将关联的RTN指针添加到OTN的proto_nodes中
  - 遍历rule_opt_config_funcs规则链，进行关键字匹配，如匹配成功则执行其对应的检测初始化函数
  - 检测初始化函数负责将需要的数据指针添加到OTN的ds_list中，并将检测函数插入OTN的opt_func链表中
  - 最后将OptListEnd插入到opt_func链的末尾



Snort原理分析--快速规则匹配初始化

- fpCreateFastPacketDetection：构建快速规则匹配结构：
  - 第一层是由四个PORT_RULE_MAP构成，分别对应TCP，UDP，ICMP及其他IP协议。
  - 第二层由PORT_GROUP结构构成，分别对应源端口、目的端口及通用端口三种
  - 第三层由RULE_NODE结构构成，分别对应以下三类规则：
    - uri-content:URI中的包含规则；
    - Content:内容包含规则；
    - non-content：非content规则。



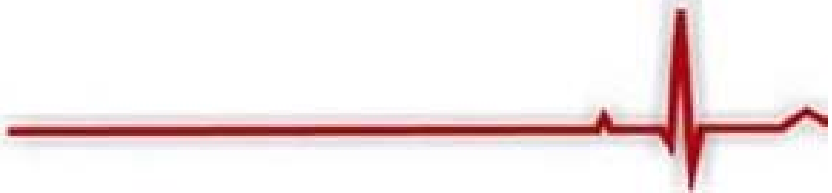
Snort原理分析--规则匹配流程

- 调用Detect(Packet * p)对数据包进行规则检测，首先判断是TCP，UDP或者ICMP协议，然后调用相关函数：fpEvalHeaderTcp，fpEvalHeaderUdp，fpEvalHeaderIcmp进行处理。如均不是则调用fpEvalHeaderIp进行处理。

- fpEvalHeaderXXX的处理过程为首先通过对应协议的PORT_RULE_MAP，根据源端口与目的端口号，找到相关的PORT_GROUP。

- 执行fpEvalHeaderSW()，通过规则类型从PORT_GROUP中找到RULE_NODE，然后对相应规则进行匹配。具体如下：
  - 对于url-content或content，执行多模式匹配函数mpseSearch()进行匹配；
  - 对于非content规则，则逐一运行OTN中opt_func链中的所有规则匹配函数。如均执行成功，则继续运行其指向RTN中的rule_func链中的匹配函数。

- 如均匹配成功，则最后会调用输出插件进行输出。



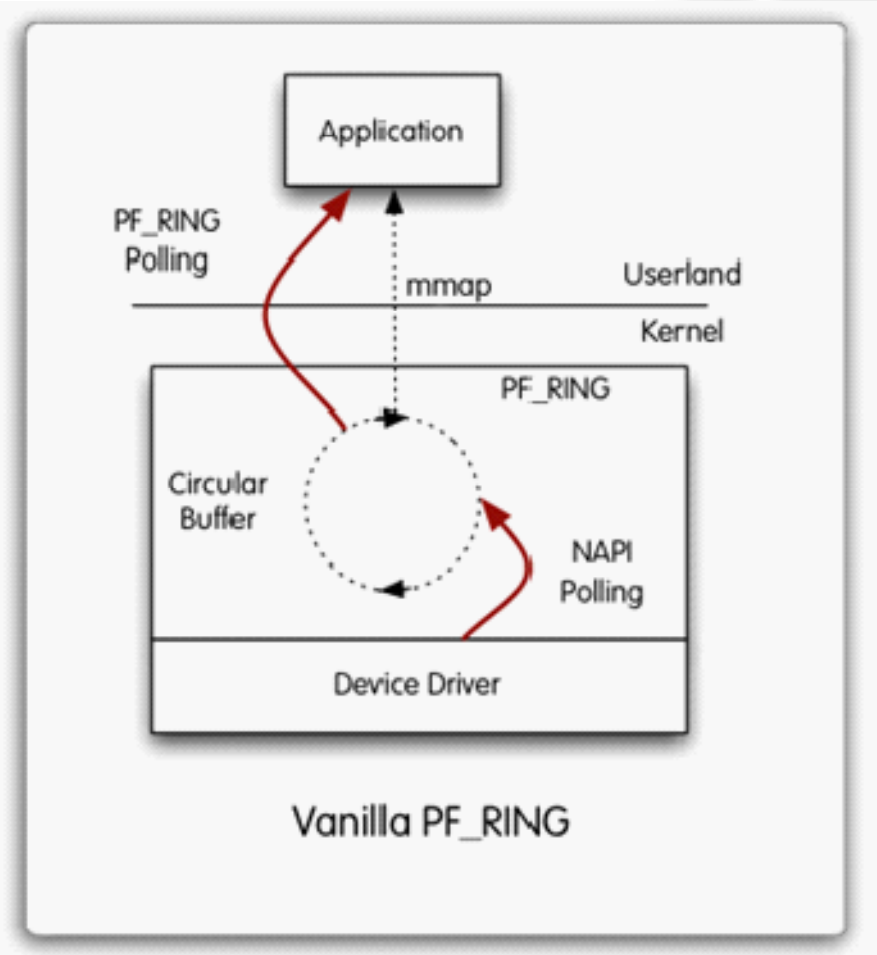
Snort目前存在的问题

- snort是一款轻量级的网络入侵检测系统。如果将其应用在大规模、大流量的网络中，snort的报警性能会非常低下。可见snort以下两方面的效率还无法满足我们的需求：
  - 数据包捕获效率
  - 数据包分析处理效率

- 改进思路为：
  - 首先需保证snort可以捕获到网络中的所有数据包。
  - 然后需要保证这些数据包可以及时得到处理。

高效的数据包捕获接口--PF_RING

- PF_RING是一个第三方的内核数据包捕获接口，类似于libpcap。它提供一种PF_RING类型的套接字，大大增加了数据包捕获的效率。
- 它包括三方面的内容：
  - 网卡驱动程序
    - PF_RING-aware drivers
    - User-space DNA (Direct NIC Access) drivers
  - PF_RING内核模块
  - 用户态函数库
    - Libpcap-ring
    - pfring-daq-module



传统数据包捕获流程

- 传统的数据包捕获流程如下(NAPI polling)：
  - 网卡中断处理程序
  - 网卡接收程序
  - 分配skb内存
  - 将其放入softnet_data队列中
  - 置软中断位
  - do_softirq
  - net_rx_action
  - iprecv
  - IP层检查数据有效性
  - TCP/UDP协议处理
  - 唤醒用户层进程

PF_RING数据包捕获流程

- PF_RING有三种数据包捕获模式，由插入模块时的参数transparent_mode控制：

Mode	Standard driver	PF_RING-aware driver	Packet Capture Acceleration
0	Packets are received through Linux NAPI		Same as Vanilla Linux
1	Packets are received through Linux NAPI	Packets are passed to NAPI (for sending them to PF_RING-unaware applications) and copied directly to PF_RING for PF_RING-aware applications (i.e. PF_RING does not need NAPI for receiving packets)	Limited
2	The driver sends packets only to PF_RING so PF_RING-unaware applications do not see any packet	The driver copies packets directly to PF_RING only (i.e. NAPI does not receive any packet)	Extreme

用PF_RING优化SNORT

- 网卡要求：
  - 1 Gigabit/sec：Intel 82575/82576/82580/I350-based（Linux driver igb ）
  - 10 Gigabit/sec：Intel 82598/82599based（Linux driver ixgbe）

- 优化步骤：
  - 安装PF_RING的kernel模块
  - 安装PF_RING的用户态库
  - 安装Snort的DAQ
  - 安装PF_RING的pfring-daq-module
  - 安装snort
  - 安装PF_RING-aware网卡驱动

数据包捕获优化性能对比

- 首先进行千兆网络中小数据包高频率的测试，单数据包大小选取6字节，分别测试发送1000到20000个数据包，测试结果如下：

发包数量	发包频率(万次/秒)	实际带宽(MB/s)	未优化前snort报警数	PF_RING优化后snort报警数
1000	19.6	8.6	1000	1000
1500	20	9	1316	1500
2000	21	9.4	1316	2000
4000	25	11	1317	4000
8000	31	13.6	1316	8000
10000	33	14	1316	10000
20000	35.4	15	1316	20000

数据包捕获优化性能对比

- 然后进行千兆网络中大数据包高带宽的测试，数据包发送个数1000，分别测试包大小从6B至1K时的报警数量，测试结果如下：

数据包大小(Byte)	发包频率(万次/秒)	实际带宽(MB/s)	未优化前snort报警数	PF_RING优化后snort报警数
10	20.4	9.76	1000	1000
20	20.2	11.55	983	1000
40	20.48	15.63	795	1000
80	20.2	23.12	582	1000
100	20.17	26.93	507	1000
200	20.58	47.11	313	1000
400	20.07	84.24	182	1000
800	19.5	106.76	130	1000
1000	18.65	114.93	125	1000

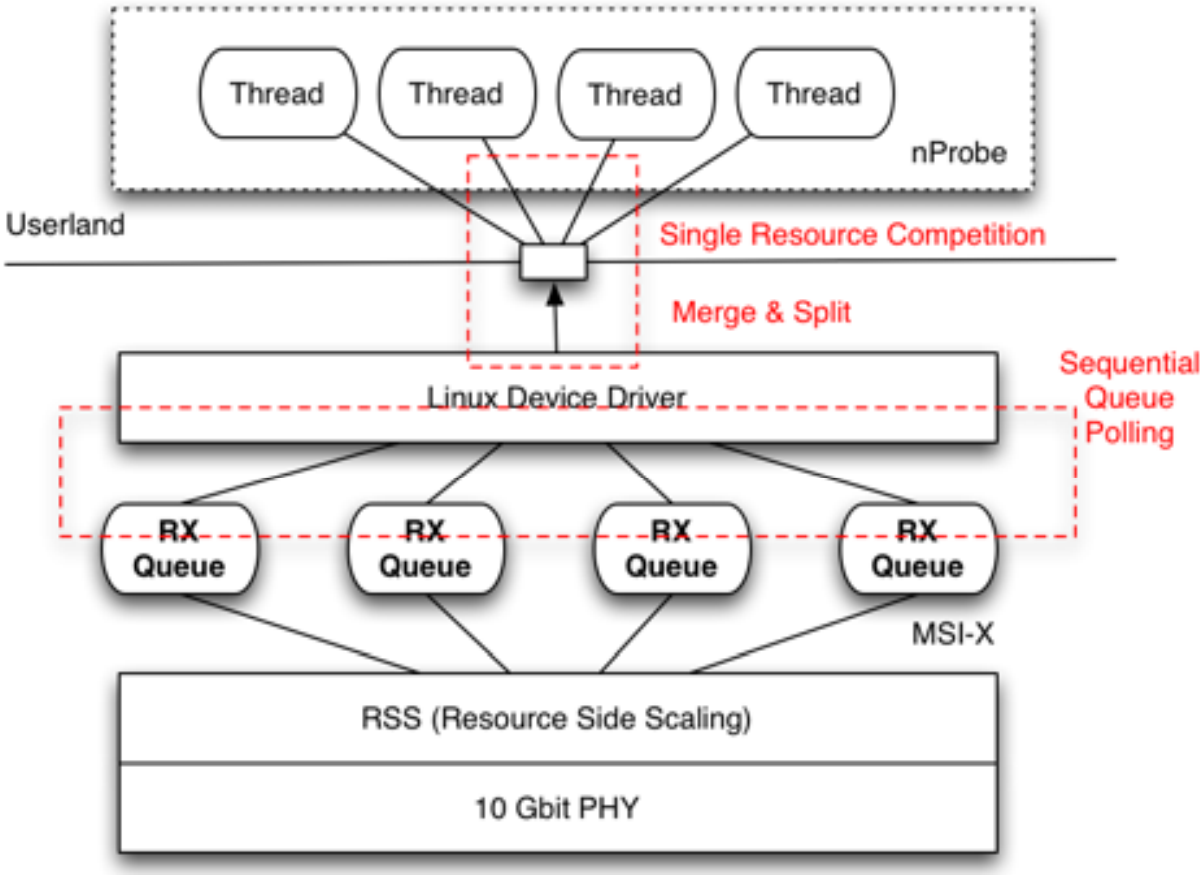
迈向万兆网络

- 用PF_RING优化过的snort，已经完全满足了千兆网络的要求。但是对于每秒几十Gbit的网络流量来说，千兆网络还远远不够。我们需要一种可以工作在万兆网络中的IDS系统。

- 首先需解决数据包捕获问题。目前支持万兆线速包捕获的方式有：
  - TNAPI（Threaded NAPI）
  - NIC（Direct NIC Access）

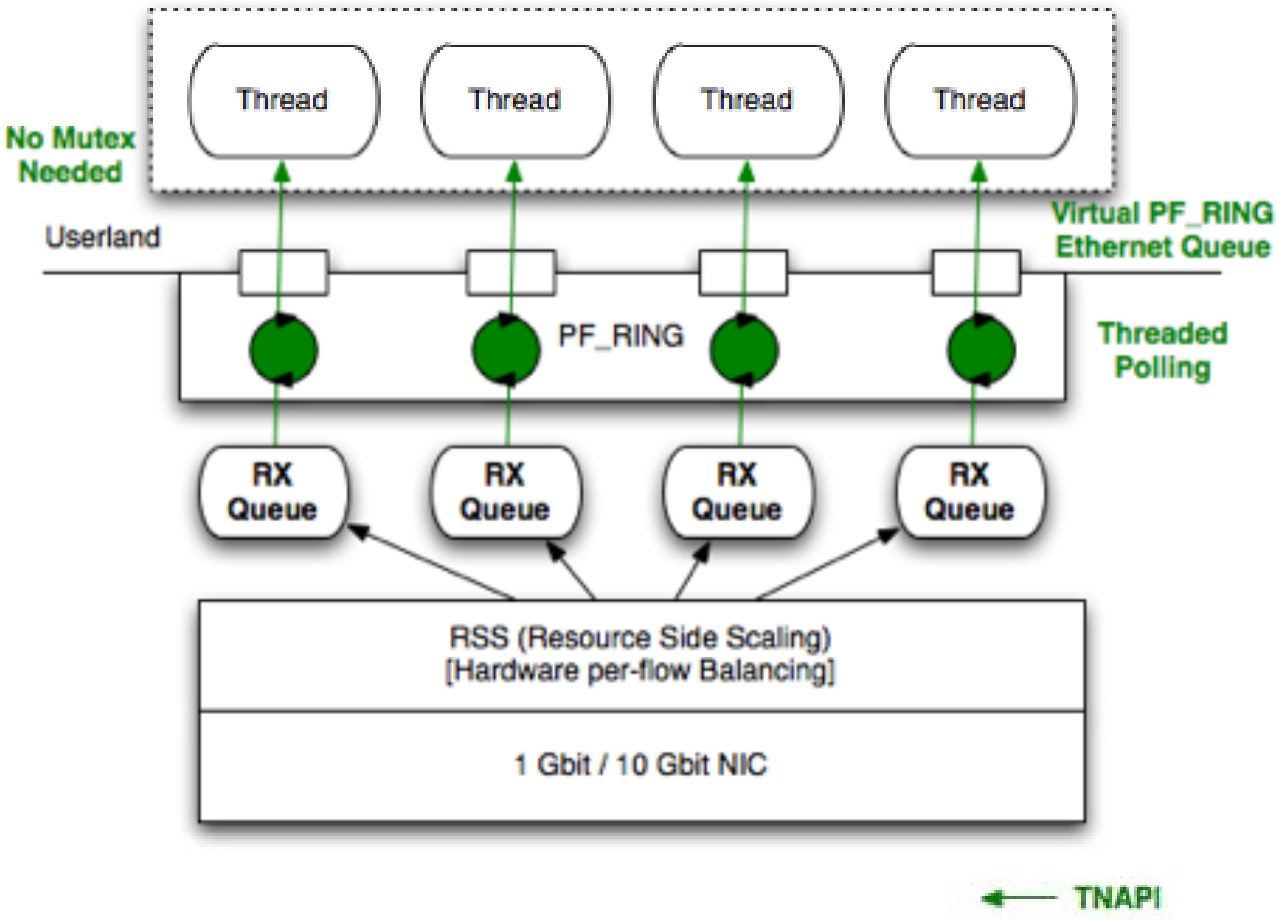
万兆网络数据包捕获--TNAPI

- 支持MSI-X 中断（Message Signaled Interrupts）的网卡可以将RX队列分解为多个，每个对应一个核。MSI-X使得设备可以最多分配2048个中断，而且每个都可以有独立的地址。
- 但是常规内核无法发挥其优势，原因为：
  - 操作系统合并多个RX到单个ethX接口，对于万兆网络是不可接受的
  - 多个线程将竞争同一个ethX



万兆网络数据包捕获--TNAPI

- TNAPI通过将内核RX队列直接暴露给用户态程序。使得：
  - 流量真正分布到多个核
  - 同时从每个RX队列轮询包
  - 通过PF_RING，应用层可以针对每个RX队列分配一个线程
- 系统结构如右图所示。



Snort multiprocessing with PF_RING+TNAPI

- 有了以上技术作为前提，我们设计出第一套针对万兆网络的IDS系统（处理性能将达到5Gbps以上）：

  Snort multiprocessing with PF_RING+TNAPI

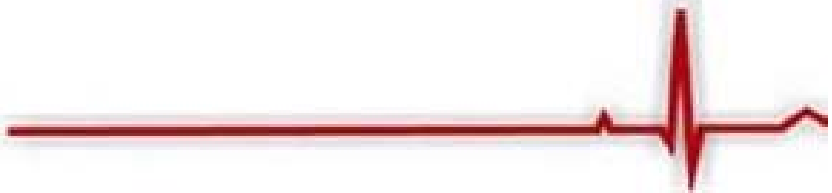
  系统采用PF_RING+TNAPI的抓包方式线速获取万兆网络的数据包，然后送给多个Snort进程进行处理。

- 该系统的优点为：不但解决了抓包效率的问题，还通过开启多个Snort进程解决了数据包处理效率的问题。

Snort multiprocessing with PF_RING+TNAPI

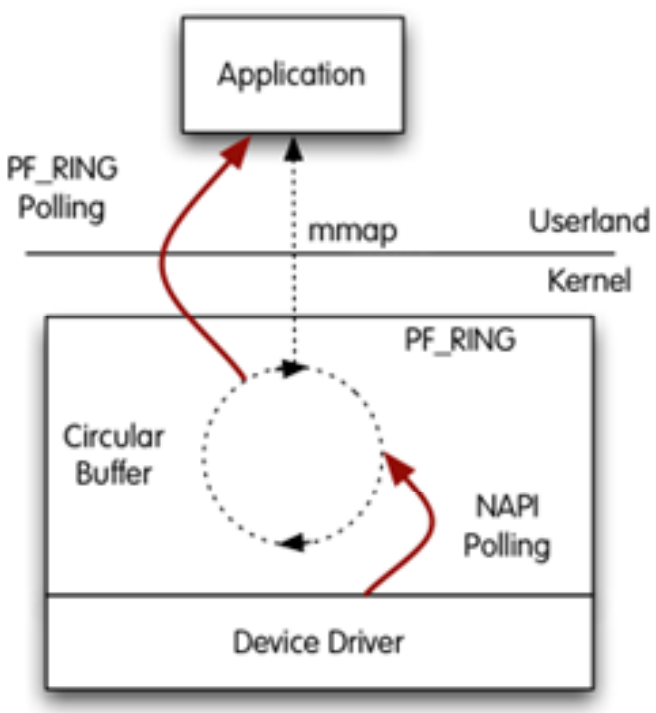
- 系统硬件推荐如下：（双路服务器）
  - 网卡：Intel 82598/82599万兆网卡 x2
  - CPU：Intel Xeon X5675 （6核心，12线程）x 2
  - 内存：24G

- 该系统软件构建过程如下：
  - 插入网卡驱动模块时，设置IntMode参数为3:
    insmod ./<driver name>.ko IntMode=3 (enables MSI-X)
  - 用ICC编译snort程序（ICC会对将程序针对intel平台进行优化）
  - 针对每块万兆网卡，启动12个snort进程:
    snort --daq-dir=/usr/local/lib/daq --daq pfring --daq-mode passive -i eth0@0 -c /etc/snort/snort.conf
    .....

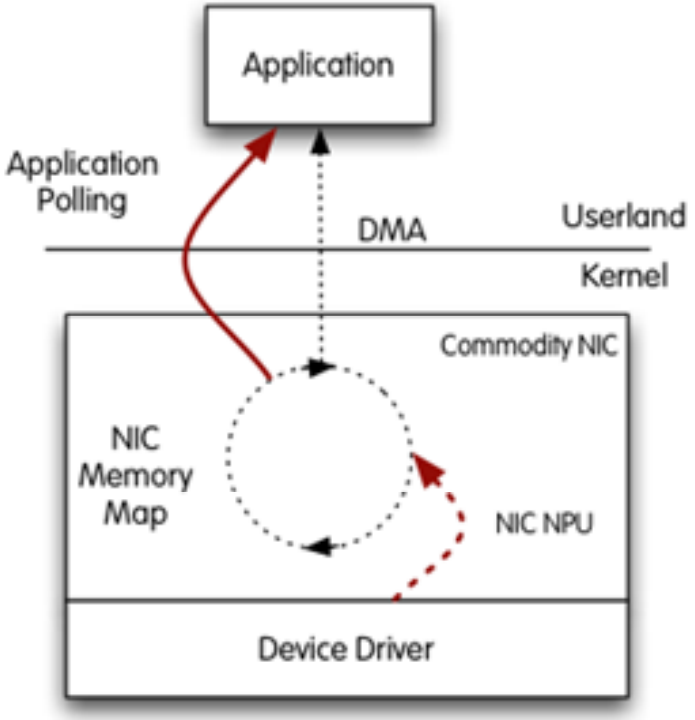


万兆网络数据包捕获--DNA

- DNA (Direct NIC Access)是一种将NIC的内存直接映射到用户空间的技术。由NPU (Network Process Unit) 负责将数据包拷贝到ring中，大大节省了CPU资源。



Vanilla PF_RING



PF_RING with DNA
(Direct NIC Access) driver

Snort multiprocessing with PF_RING+DNA

- DNA 也可以将网卡RX队列分为多个，但是每个队列只能由一个用户态进程访问。其RX队列上限会被具体硬件所限制。（现代的NIC一般可分16个以上的RX队列）

- 安装方法如下：
  - 编译并插入网卡的DNA驱动。
  - 针对每块万兆网卡，启动12个snort进程:

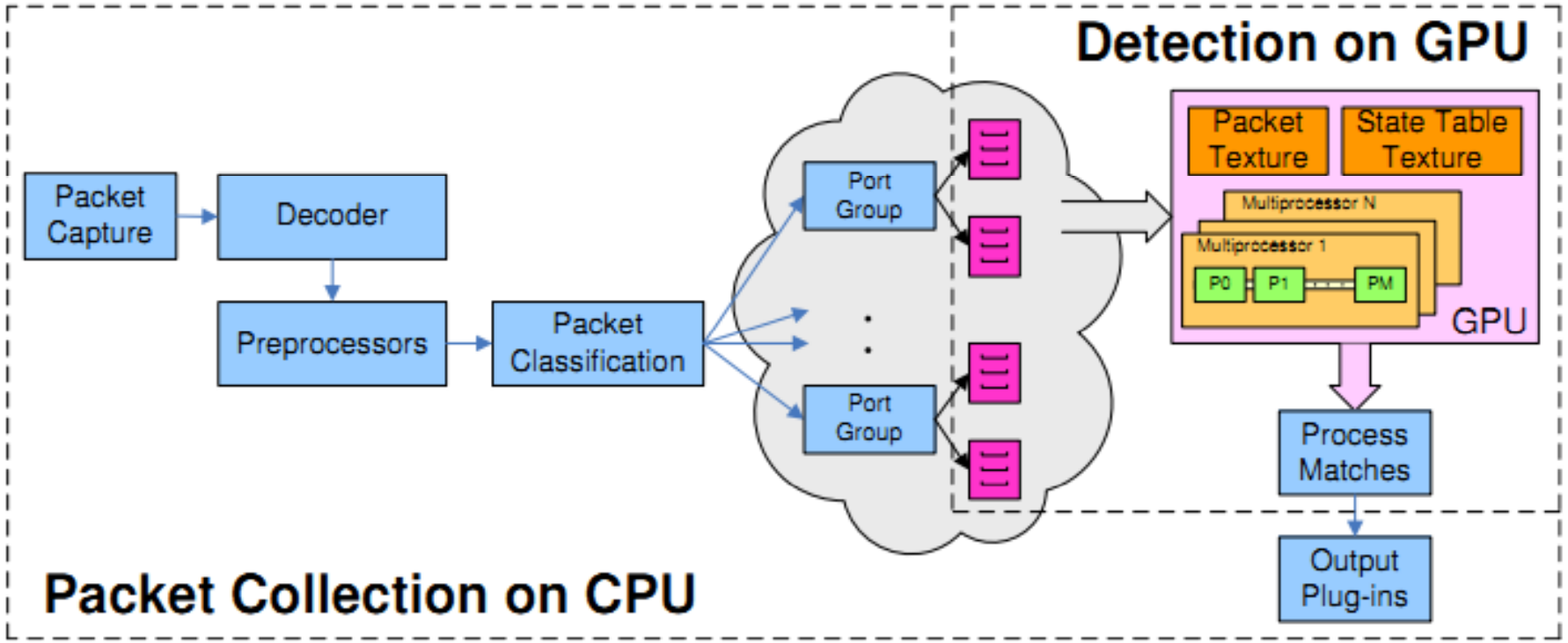
    snort --daq-dir=/usr/local/lib/daq --daq pfring --daq-mode passive -i dna0@0 -c /etc/snort/snort.conf

Gnort--GPU简介

- 目前，主流GPU的单精度浮点处理能力已经达到了同时期的CPU的10倍左右，而其外部的存储器带宽则是CPU的5倍左右。
- 近几年来GPU的性能每一年都可以翻一倍，大大超过了CPU遵循的摩尔定律。
- GPU在逻辑性处理上有很大不足，导致一般程序都需要CPU+GPU的异构并行方式在GPU上执行。
- 2007年6月，NVIDIA推出了CUDA(Compute Unified Device Architecture,统一计算设备架构)。CUDA采用类c语言进行开发。

Gnort--NIDS using GPU

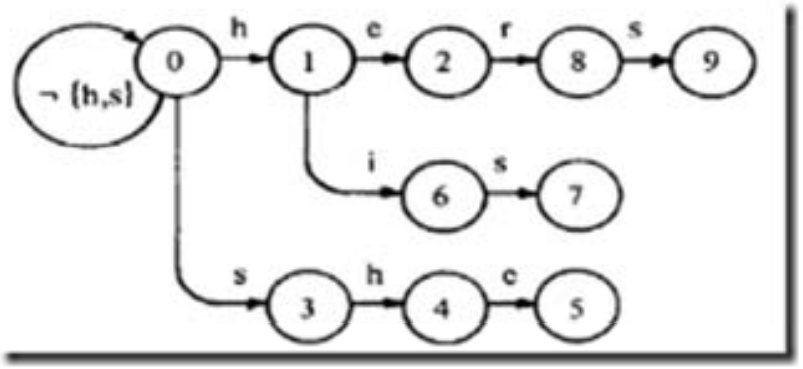
- Gnort是一套采用CPU+GPU异构并行的NIDS系统。
- 其中CPU负责捕获数据包，并进行解码与预处理等工作。而GPU负责具体的规则检测。最终GPU将检测结果发回CPU做进一步处理。（输出）
- Gnort通过将数据包的模式匹配放到GPU上并行执行，大大缩短匹配时间。实际测试中，其效率较CPU至少提升了一倍。



AC模式匹配算法简介

- AC算法思想：用多模式串建立一个确定性的树形有限状态机，以主串作为该有限状态机的输入，使状态机进行状态的转换，当到达某些特定的状态时，说明发生模式匹配。
- AC模式匹配算法分为预处理阶段与匹配阶段。在预处理阶段，AC自动机算法建立了三个函数，转向函数goto，失效函数failure和输出函数output，由此构造了一个树型有限自动机。
- **AC模式匹配的时间复杂度为O(n)。**
- 右图为模式串为：he/ she/ hers/ his 时的三类函数。

转向函数：



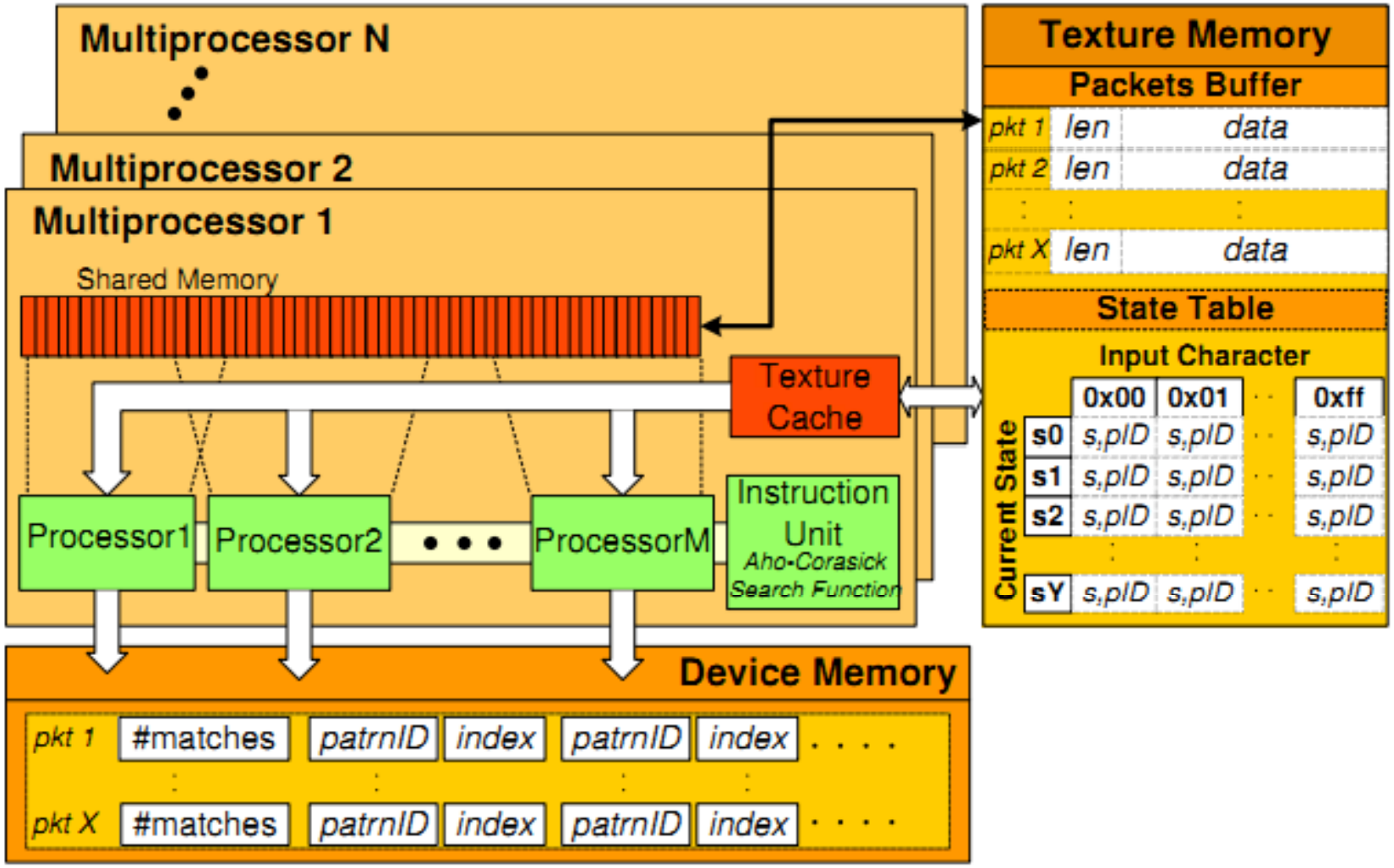
失效函数：

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

输出函数：

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

AC模式匹配算法在GPU上的实现



- 谢谢！