

《软件安全漏洞分析与发现》

网络协议逆向分析

闫佳

中国科学院软件研究所

2025年5月7日



一. 概述

网络协议逆向基础知识和原理

二. 协议消息格式逆向

协议字段划分、字段关系和字段语义恢复

三. 协议状态机恢复

协议消息类型的识别，状态机的恢复和化简

四. 密码运算逆向恢复

- 网络协议逆向目标
 - 基于网络协议的二进制实现代码，分析和提取消息格式和交互过程等网络协议规范
- 网络协议逆向应用
 - 网络协议测试
 - 通过逆向获取协议格式，辅助生成测试用例
 - 网络协议行为特征识别
 - 基于网络流量的深度数据包分析（DPI）、入侵检测
 - 兼容老旧软件系统

- 网络协议

- 定义计算机和电信网络中数据如何构造、传输和处理的一系列规范和准则
- 互联网发展不可或缺的重要组成部分

- 网络协议标准化

- 组织

- 国际标准化组织ISO，国际电信联盟电信标准化部门（ITU-T）
- 国际互联网工程任务组（IETF）、美国电气电子工程师学会（IEEE）、万维网联盟（W3C）等

- 模型

- ISO发布的OSI七层模型，TCP/IP协议模型



1、概述

- 网络协议 (物理层、数据链路层、网络层)

- 来自科来公司网站

http://www.colasoft.com.cn/download/protocols_map.php

第3层 网络层

控制子网的运行，如逻辑编址、分组传输、路由选择。

3



第2层 数据链路层

物理寻址，同时将原始比特流转变为逻辑传输线路。

2



第1层 物理层

机械、电子、定时接口通信信道上的原始比特流传输。

1

IEEE 802.2
Ethernet v.2
Internetwork

1、概述

• 网络协议（传输层、会话层、表示层）

第6层 表示层

信息的语法语义以及它们的关联，如加密解密、转换翻译、压缩解压缩。

6

第5层 会话层

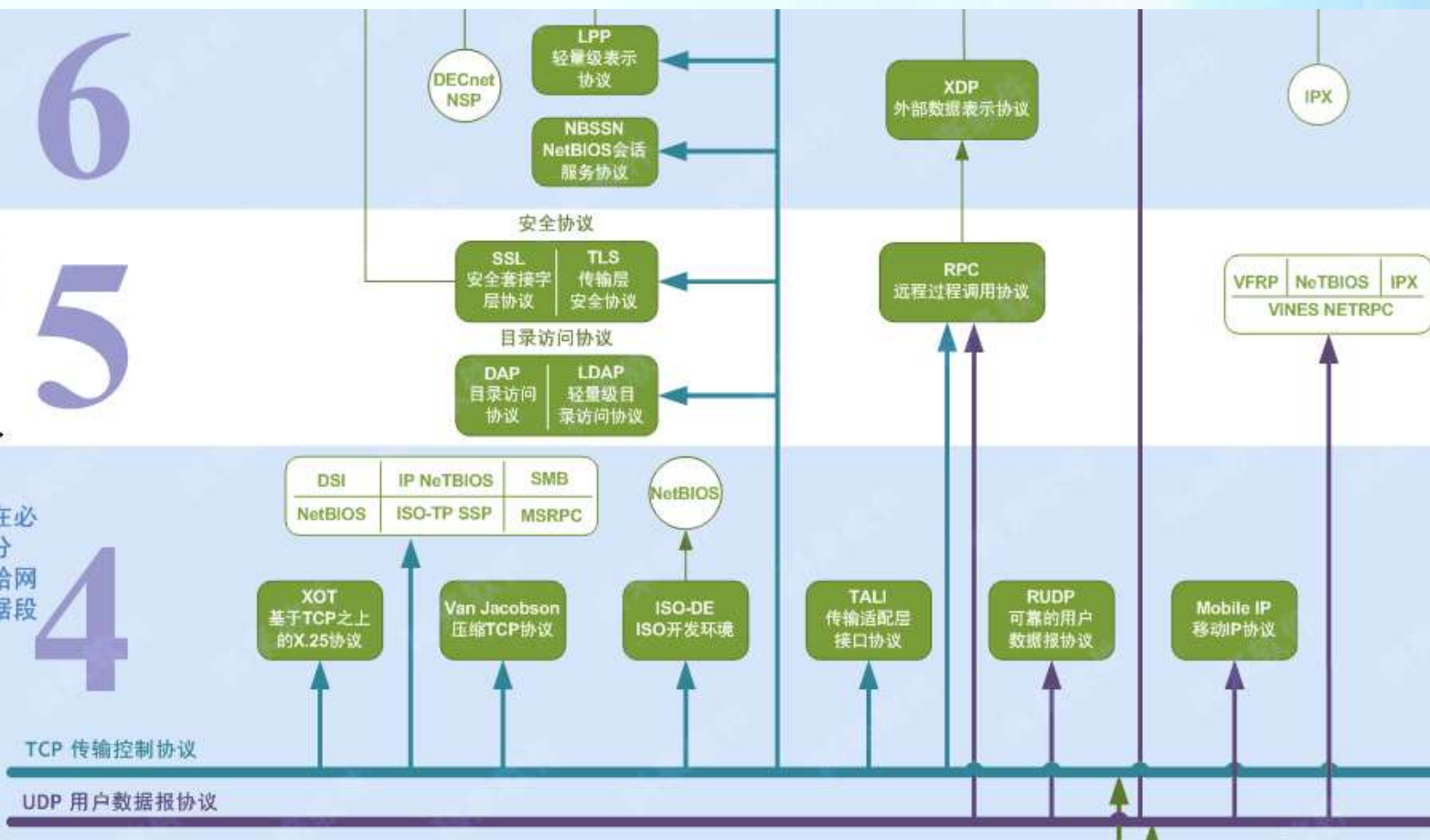
不同机器上的用户之间建立及管理会话。

5

第4层 传输层

接受上一层的数据，在必要的时候把数据进行分割，并将这些数据交给网络层，且保证这些数据段有效到达对端。

4

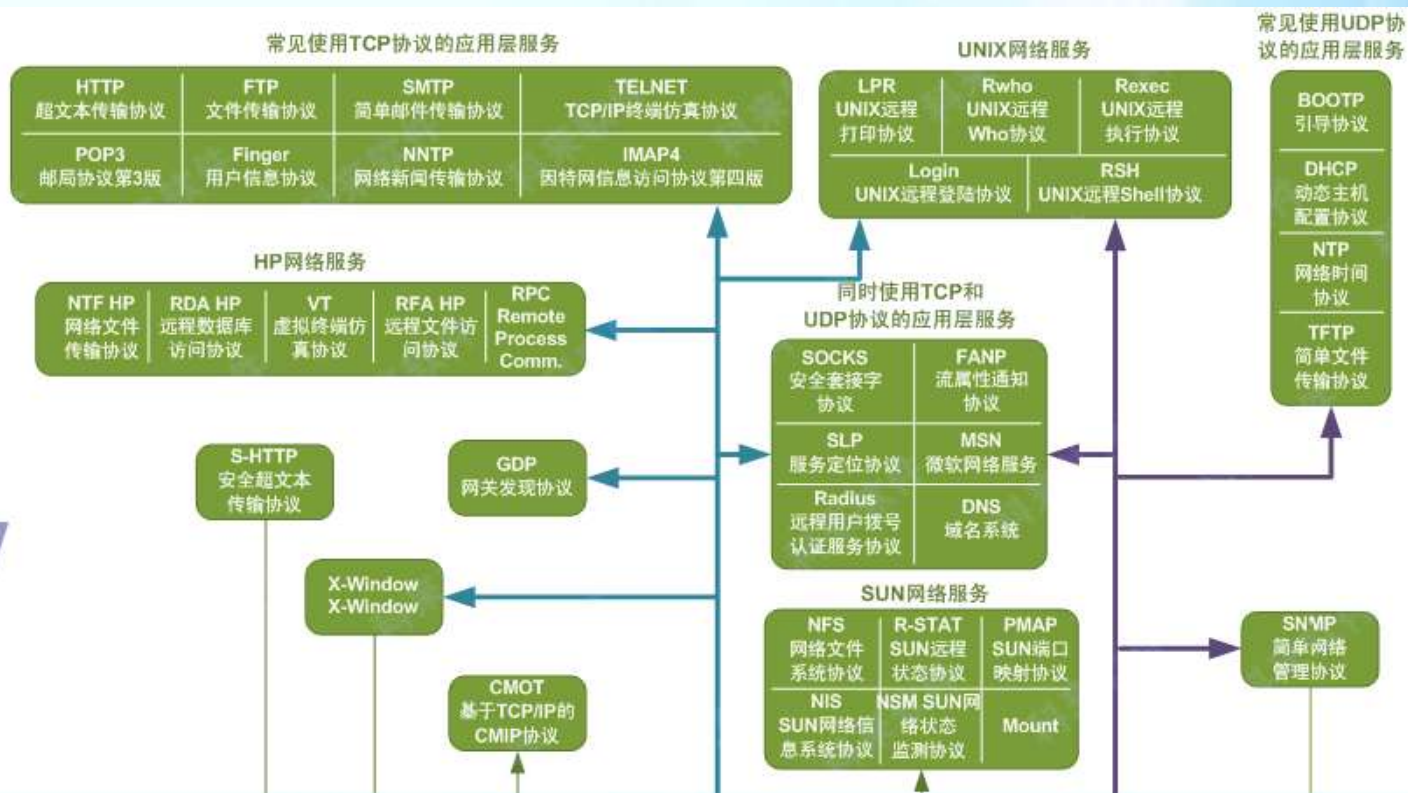


• 网络协议（应用层）

第7层 应用层

各种应用程序协议，如
HTTP、FTP、SMTP、
POP3。

7



- 网络协议的构成要素

1990年Gerard J. Holzmann 《Design and validation of computer protocols》

- 1.协议提供的服务

- 协议在应用中所提供的功能

- 2.协议依赖环境和条件等

- 协议部署应用所依赖的外部环境

- 3.协议的消息类型

- 协议中消息的种类

- 4.协议的消息格式和组成方法

- 协议中消息的基本元素、组成结构、编码方法等

- 5.协议状态机

- 即对协议中消息交换过程进行约束的一系列规则

- 网络协议的构成要素示例

- FTP协议

- 文件存储、查询和下载协议：， rfc959 (<https://www.ietf.org/rfc/rfc959.txt>)
 - 常用服务端：FileZilla Server, Serv-U, Xlight SFTP, VSFTPD, Crushftp
 - 常用客户端：Windows Explorer, 各类浏览器和下载工具

- FTP协议历史

- 1971年：ARPANET网络提供最基础的文件服务（RFC114）
 - 1973年，FTP协议首次正式通过RFC454 标准化
 - 1980年，FTP协议由NCP网络迁移至TCP网络（RFC765）
 - 1985年，最近更新的FTP协议标准规范（RFC959）
 - 2005年，基于TLS保护FTP协议，Securing FTP with TLS（RFC4127）

- 网络协议的构成要素示例（FTP协议）

- 1.协议提供的服务

- 具备用户身份认证功能的远程文件系统服务，支持文件和文件夹的上传、下载、查询、更新和删除

- 2.协议依赖环境和条件

- 基于提供可靠信息传输服务的TCP协议

- 3.协议的消息类型

- 包括请求和回复两大类型

- 请求类型包含用户登录、用户密码、下载文件、上传文件等几十个类型，回复类型包含失败、成功、错误等几十种具体类型

- 4.协议的消息格式和组成方法

- 由消息类型和消息内容组成，这两部分由空格字符分隔

- 5.协议状态机，即对协议中消息交换过程进行约束的一系列规则

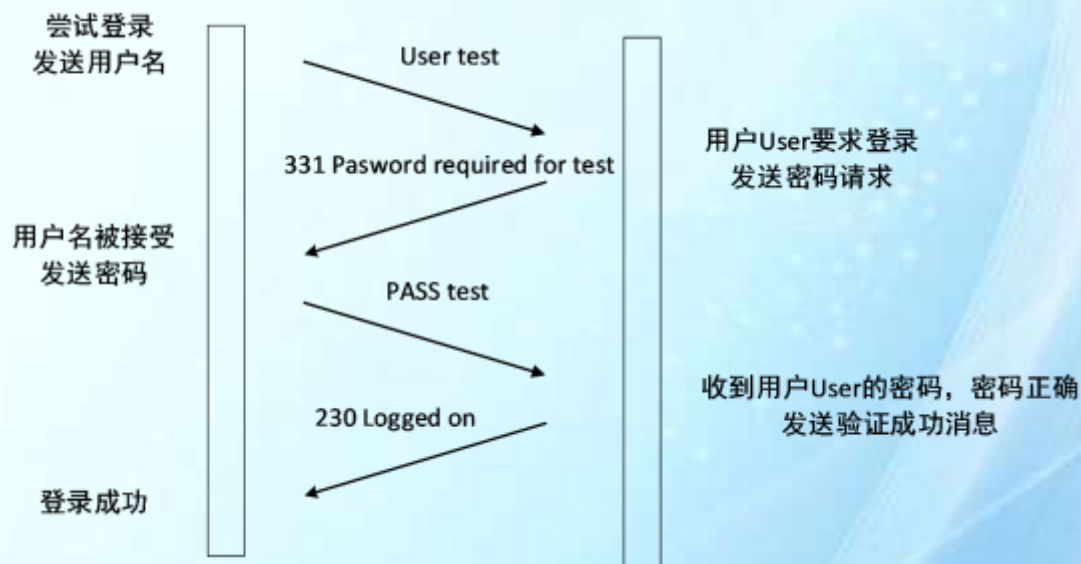
- 多用户登录机制、命令请求和回应等

- 典型网络协议交互过程示例

- FTP协议登录交互过程的网络流量

- Wireshark软件提取，FileZilla Server服务端，Internet Explorer、Edge

序号	源地址	目的地址	协议	长度	请求类型	请求内容	应答类型	应答内容
20	192.168.1.185	172.18.144.11	FTP	96			Service ready for new user	FileZilla Server version 0.9.41 beta
21	192.168.1.185	172.18.144.11	FTP	99			Service ready for new user	written by Tim Kosse (Tim.Kosse@gmx.de)
23	192.168.1.185	172.18.144.11	FTP	115			Service ready for new user	Please visit http://sourceforge.net/projects/filezilla/
24	172.18.144.11	192.168.1.185	FTP	65	USER	test		
26	192.168.1.185	172.18.144.11	FTP	86			User name okay, need password	Password required for test
27	172.18.144.11	192.168.1.185	FTP	65	PASS	test		
29	192.168.1.185	172.18.144.11	FTP	69			User logged in, proceed	Logged on



- 若去除Wireshark的内容解析，直接观察流量数据包内容

序号	时间	源地址	目的地址	协议	长度	内容
20	4.153493	192.168.1.185	172.18.144.11	FTP	96	0x3232302D46696C655A696C6C61205365727665722076657273696F6E20302E392E34312067
21	4.153607	192.168.1.185	172.18.144.11	FTP	99	0x3232302D7772697474656E2062792054696D20486F737365202854696D2E486F7373654067
23	4.153877	192.168.1.185	172.18.144.11	FTP	115	0x32323020506C6561736520766973697420687474703A2F2F736F75726365666F7267652E68
24	4.153961	172.18.144.11	192.168.1.185	FTP	65	0x5553455220746573740D0A
26	4.155993	192.168.1.185	172.18.144.11	FTP	86	0x3333312050617373776F726420726571756972656420666F7220746573740D0A
27	4.156063	172.18.144.11	192.168.1.185	FTP	65	0x5041535320746573740D0A
29	4.157621	192.168.1.185	172.18.144.11	FTP	69	0x323330204C6F67676564206F6E0D0A

```
0x3232302D46696C655A696C6C61205365727665722076657273696F6E20302E392E3431206  
0x3232302D7772697474656E2062792054696D204B6F737365202854696D2E4B6F737365406  
0x32323020506C6561736520766973697420687474703A2F2F736F75726365666F7267652E6  
0x5553455220746573740D0A  
0x3333312050617373776F726420726571756972656420666F7220746573740D0A  
0x5041535320746573740D0A  
0x323330204C6F67676564206F6E0D0A
```

- 网络协议逆向
 - 协议消息格式逆向
 - 字段划分
 - 字段关系提取
 - 字段间从属、位置型关系
 - 字段语义分析
 - 关键字
 - 其他：
 - 加密算法识别
 - 协议状态机逆向
 - 协议消息类型提取
 - 消息聚类
 - 协议状态机推断和化简
 - 有限状态机

• 网络协议逆向典型步骤

原始消息

GET /news.html HTTP/1.0\r\n

HTTP/1.1 200 OK\r\n

字段划分

GET

/news.html

HTTP

/

1.1

\r

\n

字段关系
字段语义

GET /news.html HTTP/1.1\r\n

GET /news.html

HTTP/1.1

/r/n

GET

/news.html

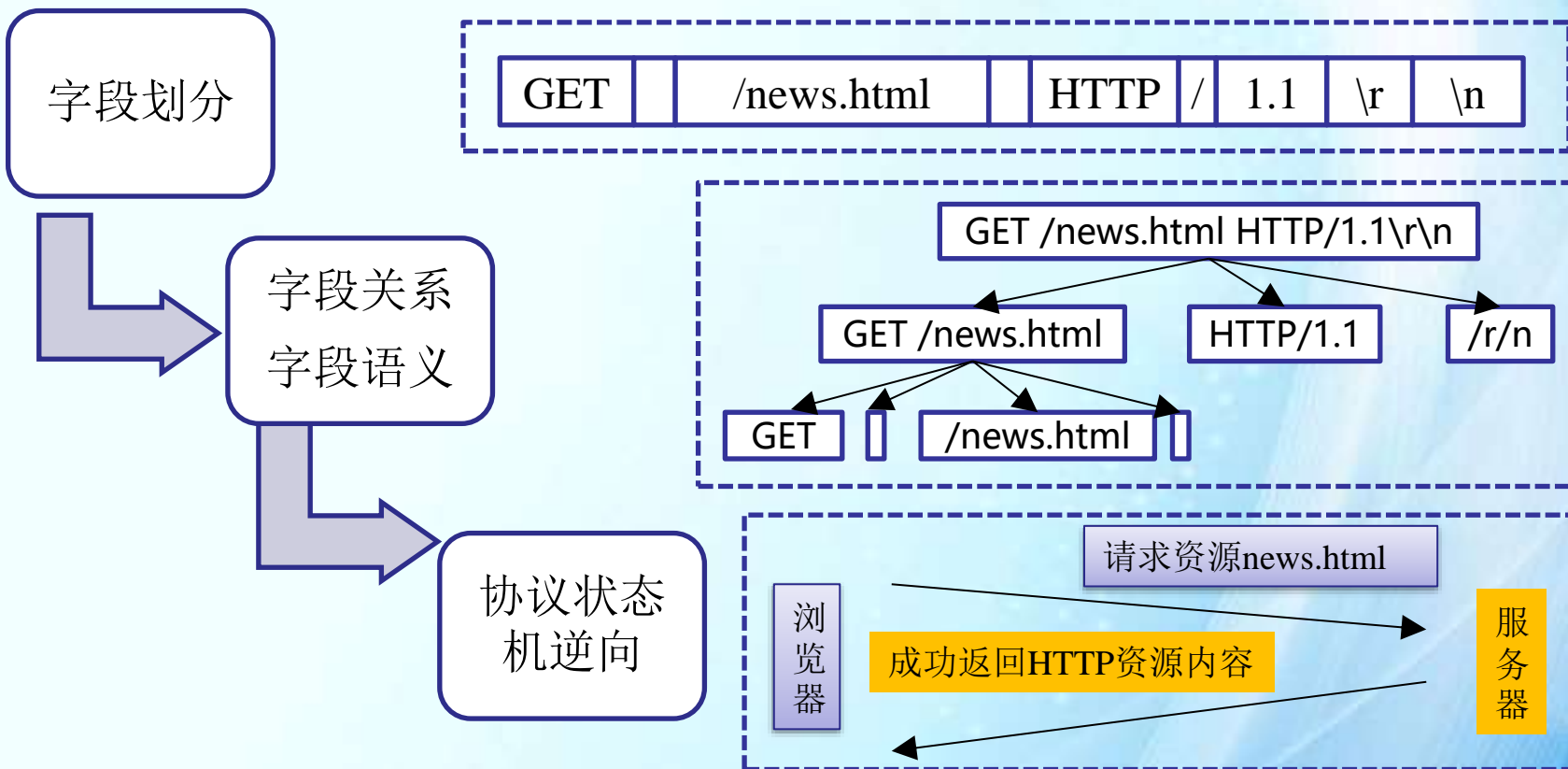
协议状态
机逆向

请求资源news.html

浏览器

成功返回HTTP资源内容

服务器



- 网络协议逆向主要方法

- 基于网络流量分析的逆向方法

- 思路：流量特征匹配、序列相似性分析
 - 优点：只需要获取目标网络协议运行时产生的流量，但是逆向分析准确性依赖于大量流量数据
 - 缺点：无法针对加密网络协议进行逆向

- 基于可执行程序分析的逆向方法

- 动态监控程序执行过程，依据程序对网络消息的处理过程恢复消息格式和状态转换过程
 - 同样需要合规的网络流量

- 国内外研究现状

- 网络流量

- 2007年，微软研究院学者W. Cui提出使用聚类等数据分析方法应用于网络协议的自动化逆向

- 网络协议逆向

- 2007年，卡内基梅隆大学D.Song首次提出基于污点传播的动态程序分析方法
 - 2008年，普渡大学Z.Lin提出依据消息处理代码执行的上下文语义的分析方法
 - 2009年，加州伯克利D.Song提出基于后向程序切片的语义回溯分析方法

- 综述

- A Survey of Automatic Protocol Reverse Engineering Tools. ACM Computing Surveys 2015

- 经典的网络协议逆向案例
 - SMB协议：文件和打印共享服务
 - Samba项目：支持类Unix系统的 SMB/CIFS 网络协议实现
 - 软件更新协议
 - 路由器等网络设备固件等
 - SCADA协议等工控服务相关协议
 - 各类蠕虫、木马和僵尸网络等恶意代码
 - Zeus, SpyEye, Storm, ZeusGameover

- 网络协议逆向面临的挑战

- 协议通信数据：

- 非可读地“二进制”协议数据，混乱高熵数据

- 协议数据内部各个字段之间的复杂依赖关系

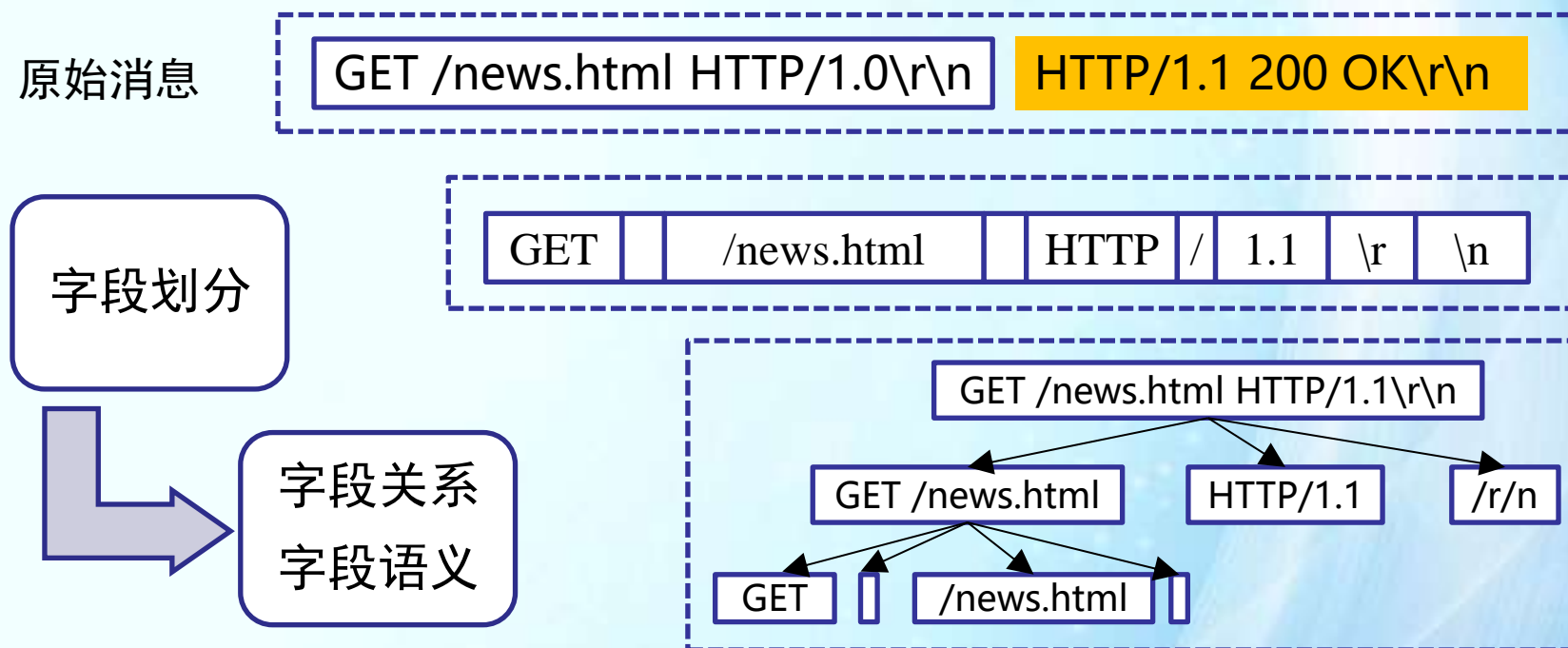
- 可变长度的协议字段
 - 基于上下文的消息字段依赖
 - 基于交互协商形成的数据

- 流量数据加密导致流量分析方法失效

- TLS协议、IPSec协议
 - 其他自定义加密协议

2、协议消息格式逆向

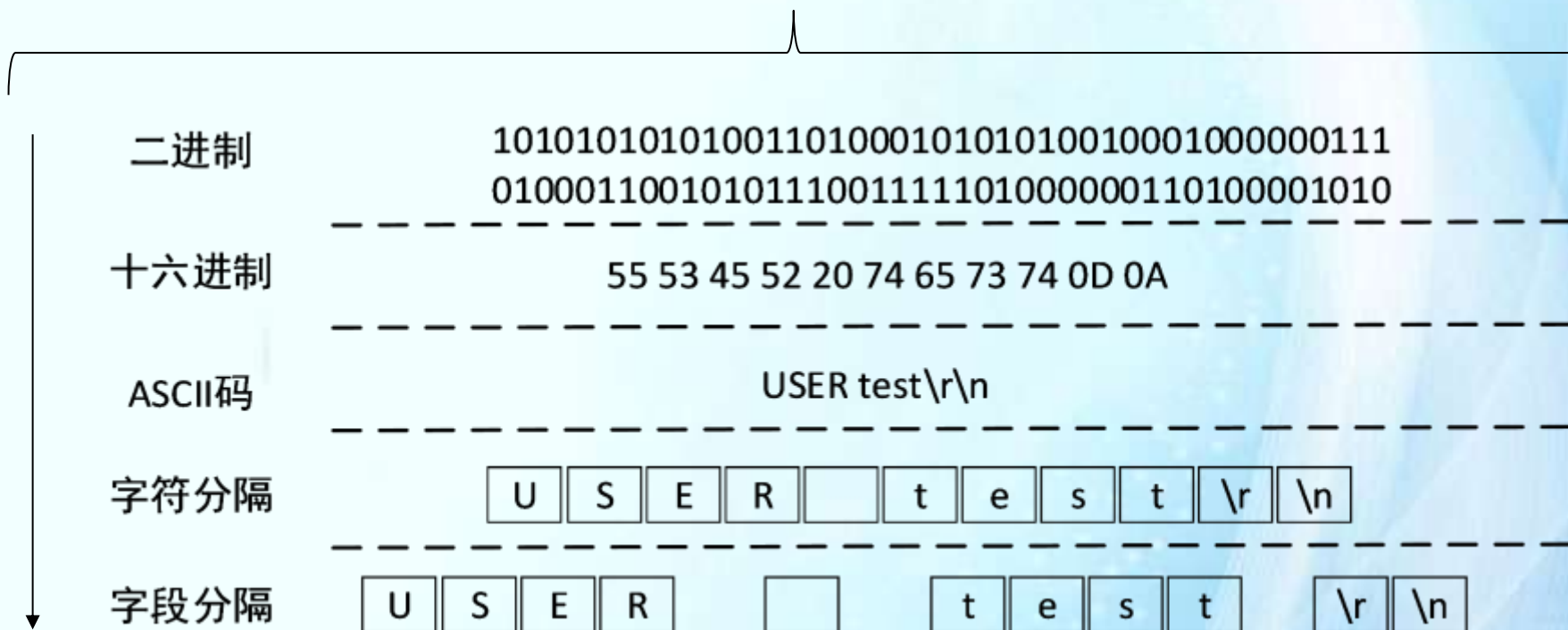
- 通过逆向还原出协议消息的格式
 - 字段划分 → 字段间关系恢复 → 字段关联语义恢复



2、协议消息格式逆向 – 字段划分

- 字段划分 将某一类型的消息整体按照一定规则切分为不同的区域（字段）

举例：FTP协议的USER消息



- 数据表现形式的改良可以改善分析人员对数据的理解
- 协议应用背景知识有助于协议逆向

- 字段划分方法

思路一：单个消息划分为若干个部分

思路二：单个消息的各个字节按照功能等分类标准进行聚合分类

- 基于分隔符的划分方法

- 利用分隔符切分消息, Polyglot (CCS'07)

- 基于消息处理指令上下文差异的划分方法

- 利用处理指令上下文对字节进行聚类, Autoformat (NDSS'08)

- 基于字段来源回溯的的划分方法

- 基于字段来源对字节进行聚类, Dispatcher (CCS'09)

2、协议消息格式逆向 – 字段划分

- 基于分隔符的字段划分

- 源代码示例 - FTP协议解析消息的过程（摘自FileZilla Server 0.9.43）

摘自 [Controlsocket.cpp](#)

```
246: //Split command and arguments
247: int pos = str2.Find(_T(" "));
248: if (pos != -1)
249: {
250:     command = str2.Left(pos);
251:     if (pos == str2.GetLength() - 1)
252:         args = _T("");
253:     else
254:     {
255:         args = str2.Mid(pos + 1);
256:         if (args == _T(""))
257:         {
258:             Send(_T("501 Syntax error, failed to decode
string"));
259:             return FALSE;
260:         }
261:     }
262: }
```

2、协议消息格式逆向 – 字段划分

- 基于分隔符的字段划分
 - 汇编代码示例 - FTP协议解析消息的过程
 - Find函数对应的汇编函数名称 sub_403040

```
,
v16 = sub_403040((int)&v34, (int)&unk_47C160,
if ( v16 == -1 )
{
    sub_404030(a3, &unk_48232C);
    v25 = *(_DWORD *)(a2 + 20);
    if ( v25 < 8 )
        v22 = (int *)a2;
    else
        v22 = *(int **)a2;
    v26 = &v34;
    if ( v36 >= 8 )
        v26 = (int *)v34;
    if ( v22 != v26 )
    {
        *(_DWORD *)(a2 + 16) = 0;
        if ( v25 < 8 )
            v27 = a2;
        else
            v27 = *(_DWORD *)a2;
        *(_WORD *)v27 = 0;
        loc_402AB0(a2, &v34, 0, -1);
    }
    goto LABEL_60;
}
```

```
push    1
push    0
push    offset unk_47C160
lea     ecx, [esp+80h+var_40]
call    sub_403040
mov     edi, eax
or      eax, 0FFFFFFFh
cmp     edi, eax
cmovz   edi, eax
cmp     edi, eax
jz      loc_446B3F
,
loc_446B3F:                                ; CODE
mov     edx, offset unk_48232C
mov     ecx, ebp
call    sub_404030
mov     edx, [esi+14h]
cmp     edx, 8
jb      short loc_446B57
mov     ecx, [esi]
jmp     short loc_446B59
```


2、协议消息格式逆向 – 字段划分

- 基于分隔符的字段划分
 - Find函数的源代码 和 汇编代码

```
if ( a3 > *(_DWORD *)(this + 16) )
{
    v5 = *(_DWORD *)(this + 16);
    if ( (unsigned int)a3 < v5 )
    {
        v6 = v5 - a3;
        if ( HIWORD(a3) <= v6 )
        {
            v7 = 1 - HIWORD(a3) + v6;
            v16 = *(_DWORD *)(this + 20);
            if ( v16 < 8 )
                v8 = this;
            else
                v8 = *(_DWORD *)this;
            for ( i = v8 + 2 * v3; ; i += 2 )
            {
                v17 = i;
                if ( v7 )
                {
                    v10 = v7;
                    while ( *(_WORD *)i != *(_WORD *)a2 )
                    {
                        i += 2;
                        --v10;
                        if ( !v10 )
                            goto LABEL_13;
                    }
                }
            }
        }
    }
```

```
lea     eax, [eax+edx*2]

; CODE XREF: sub_4
mov     dword ptr [esp+18h+arg_4+4], eax
test    ebx, ebx
jz      short loc_4030BB
mov     edx, [esp+18h+arg_0]
mov     ecx, ebx
movzx   edx, word ptr [edx]
lea     esp, [esp+0]

; CODE XREF: sub_4
cmp     [eax], dx
jz      short loc_4030BD
add     eax, 2
dec     ecx
jnz     short loc_4030B0
```

2、协议消息格式逆向 – 字段划分

- 字段划分

- Find函数（汇编代码）的关键比较指令

`cmp [eax], dx`

eax : 查找目标缓冲区

dx : 空格分隔符

```
lea     eax, [eax+edx*2]

; CODE XREF: sub_4
mov     dword ptr [esp+18h+arg_4+4], eax
test    ebx, ebx
jz      short loc_4030BB
mov     edx, [esp+18h+arg_0]
mov     ecx, ebx
movzx   edx, word ptr [edx]
lea     esp, [esp+0]

; CODE XREF: sub_4
cmp     [eax], dx
jz      short loc_4030BD
add     eax, 2
dec     ecx
jnz     short loc_4030B0
```

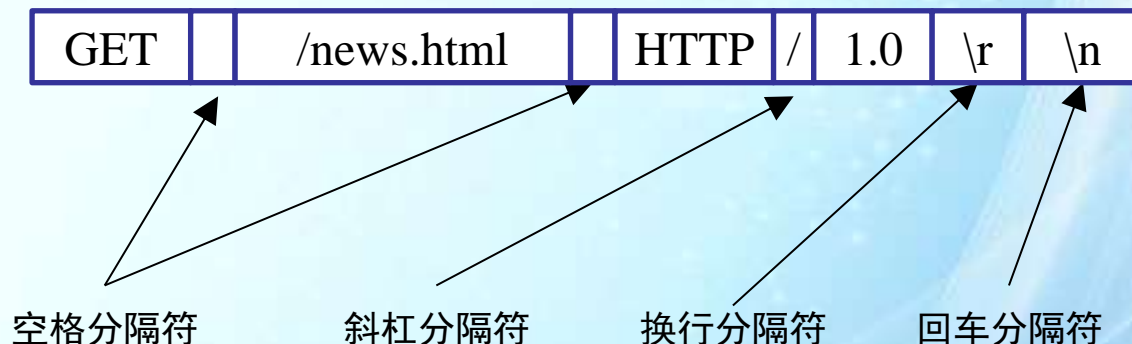
- 基于分隔符识别的字段划分

- 观察到现象：

- 分隔消息的字符会作为常量，与从网络接收到的数据依次进行比较

- 基本思想

- 与网络数据的多个位置进行连续比较的字符视作分隔符



2、协议消息格式逆向 – 字段划分

• 基于分隔符识别的字段划分



程序动态执行

网络数据作为污点，
通过污点传播获得
污点相关的比较指
令的信息



提取单字节分隔符

- 1 与连续的污点内存区域进行比较的单字节字符
- 2 该字符也出现在网络消息



生成多字节分隔符 划分字段

依据相邻出现频次
合并单字节分隔符

网络消息

G	E	T	/	\r	\n	
0	1	2	3	4	5	6

从比较指令中抽取的常量字符

0x0d (\r)	X	X	X	X	X	X
0x0a (\n)	X	X	X	X	X	X
0x47 (G)	X					
0x45 (E)		X				
0x54 (T)			X			
0x2f (/)				X	X	X
0x20 ()			X	X		



0x0d (\r)
0x0a (\n)
0x2f (/)
0x20 ()



0x0d0a (\r\n)
0x2f (/)
0x20 ()

GET / \r\n

2、协议消息格式逆向 – 字段划分

算法 基于单字符分隔符的字段划分

输入：

污点来源标签集合 T ，分别对应接收消息的各个字节

常数字符串 C (通常定义为为8bit单字节的取值域，即0-255)

污点传播记录序列（以接受消息为污点源，传播规则为数据依赖关系）Tlist

输出：疑似分隔符集合 S

Begin

Define $TaintMap = \{\}$ ，，单字节字符到污点来源标签集合的子集的映射

For t in Tlist:

 If Type(t) is 比较指令：

 提取操作数 op_a ，操作数 op_b

 If $op_a \in T \wedge op_b \in C$

$TaintMap[op_b] = TaintMap[op_b] \cup \{op_a\}$

 Else if $op_b \in T \wedge op_a \in C$

$TaintMap[op_a] = TaintMap[op_a] \cup \{op_b\}$

End For

For c , c_list in $TaintMap$:

 If $c_list.size() < 2$:

 continue

 if ($c_list ?$)

$S = S \cup \{c\}$

End For

- 基于分隔符识别的字段划分
 - 若分隔符为多个字节，考虑如下两种情况
 - 若两个分隔符出现的次数相同，且两个分隔符总是在消息中相邻的位置出现
 - 则将这两个分隔符合并为一个分隔符
 - 举例： `cmp [al], 0xA, cmp [al], 0xd` 相继出现
 - 若分隔符是基于双字节比较操作符的
 - 则还还需要使用前述的方法以两字节为粒度进行一次查找，从而能够直接定位双字节分隔符
 - 举例： `cmp [edx], eax`

2、协议消息格式逆向 – 字段划分

• 基于分隔符识别的字段划分方法小结

– 方法:

- 基于被污点数据影响的比较指令提取字段分隔符
- 主要针对具有字段分隔符的网络协议进行分析

– 多数基于文本的网络协议: HTTP、FTP、SMTP、POP3, IRC等

– 缺陷:

- 无法针对没有分隔符的二进制网络协议进行分析
- NTP, DNS, TLS

216.27.185.42 → 192.168.50.50 NTP NTP Version 3, symmetric passive

00	D0	59	6C	40	4E	00	0C	41	82	B2	53	08	00	45	00
00	4C	A2	22	40	00	32	11	22	5E	D8	1B	B9	2A	C0	A8
32	32	00	7B	00	7B	00	38	70	E1	1A	02	0A	EE	00	00
07	A4	00	00	0B	A3	A4	43	3E	C2	C5	02	01	81	E5	79
18	19	C5	02	04	EC	EC	42	EE	92	C5	02	04	EB	D9	5E
8D	54	C5	02	04	EB	D9	69	B1	74						

4 protocols in packet:

Ethernet	IPv4	UDP	NTP
----------	------	-----	-----

• Network Time Protocol (NTP Version 3)

- **Flags:** 0x1a, **Leap Indicator:** no warning, **Version number:** 3
- Peer Clock Stratum: secondary reference (2)
- Peer Polling Interval: 10 (1024 seconds)
- **Peer Clock Precision:** 0.000004 seconds
- Root Delay: 0.029846 seconds
- Root Dispersion: 0.045456 seconds
- Reference ID: 164.67.62.194
- Reference Timestamp: Sep 27, 2004 03:03:29.896378999 UTC
- Origin Timestamp: Sep 27, 2004 03:18:04.922896299 UTC
- Receive Timestamp: Sep 27, 2004 03:18:03.849098999 UTC
- Transmit Timestamp: Sep 27, 2004 03:18:03.849268999 UTC

- 基于指令执行上下文的字段划分

- 指令执行上下文


- 指令执行上下文是指执行过程中除指令本身外，周围执行环境的状态，包括目标软件、操作系统甚至系统硬件（CPU各个寄存器、内存等）的状态

系统状态信息项	描述	区分度
指令EIP	指令执行时在内存中的位置，指令本身	高
各寄存器的值	EAX,ECX,EDX等寄存器的值	低
内存值	系统内存	低
最后一次Call/JMP的地址	当前指令执行前最后一次控制转移时的地址	中

2、协议消息格式逆向 – 字段划分

- 基于指令执行上下文的字段划分
 - 指令执行调用栈
 - 指执行过程中未返回的Call调用的返回地址形成的序列
 - FileZilla Server接受并处理客户端消息时的执行调用栈

调用堆栈

名称
 FileZilla Server.exe!CControlSocket::ParseCommand() 行 531
FileZilla Server.exe!CControlSocket::OnReceive(int nErrorCode) 行 216
FileZilla Server.exe!CAsyncSocketExHelperWindow::WindowProc(HWND_ * hWnd, unsigned int message, unsigned int wParam, long lParam) 行 357
user32.dll!_InternalCallWinProc@20()
user32.dll!_UserCallWinProcCheckWow@36()
user32.dll!_DispatchMessageWorker@8()
user32.dll!_DispatchMessageW@4()
FileZilla Server.exe!CThread::Run() 行 94
FileZilla Server.exe!CThread::ThreadProc(void * lpParameter) 行 81
kernel32.dll!@BaseThreadInitThunk@12()
ntdll.dll!__RtlUserThreadStart@8()
ntdll.dll!_RtlUserThreadStart@8()

- 基于指令执行上下文的字段划分
 - 同一函数内 - 字段的使用处理
 - 摘自OpenSSL 函数库：一个TLS传输层安全套接字的实现库

摘自 `bss_dgram.c`:

```
1085:  n = recvmsg(b->num, &msg, 0);
1086:  if (msg.msg_controllen > 0){
1087:      for (cmsg = CMSG_FIRSTHDR(&msg); cmsg; cmsg =
CMSG_NXTHDR(&msg, cmsg)) {
1088:          if (cmsg->cmsg_level != IPPROTO_SCTP)
1089:              continue;
1090:      if (cmsg->cmsg_type == SCTP_RCVINFO)
1091:      {
1092:          struct sctp_rcvinfo *rcvinfo;
1093:          rcvinfo = (struct sctp_rcvinfo *)CMSG_DATA(cmsg);
1094:          data->rcvinfo.rcv_sid = rcvinfo->rcv_sid;
```

- 基于处理指令上下文差异的字段划分

- 观察到现象：

- 程序对网络协议消息的不同字段进行处理的指令上下文存在显著差异

- 基本思想

- 提取消息中每个字节的处理指令及运行时调用栈，并用其进行聚类实现字段划分

摘自 bss_dgram.c:

```
1085:  n = recvmsg(b->num, &msg, 0);
```

```
1086:  if (msg.msg_controllen > 0){
```

```
1087:      for (cmsg = CMSG_FIRSTHDR(&msg); cmsg; cmsg = CMSG_NXTHDR(&msg, cmsg)) {
```

```
1088:          if (cmsg->cmsg_level != IPPROTO_SCTP)
```

```
1089:              continue;
```

```
1090:          if (cmsg->cmsg_type == SCTP_RCVINFO)
```

```
1091:              {
```

```
1092:                  struct sctp_rcvinfo *rcvinfo;
```

```
1093:                  rcvinfo = (struct sctp_rcvinfo *)CMSG_DATA(cmsg);
```

```
1094:                  data->rcvinfo.rcv_sid = rcvinfo->rcv_sid;
```

```
.....
```


• 基于处理指令上下文差异的字段划分



程序动态执行

1. 网络数据作为污点源进行污点传播
2. 提取访问污点数据的所有指令以及指令运行时的调用栈信息，形成动态运行日志



建立协议字段树

依据动态运行日志进行离线分析，通过合并相似字段构造协议字段树



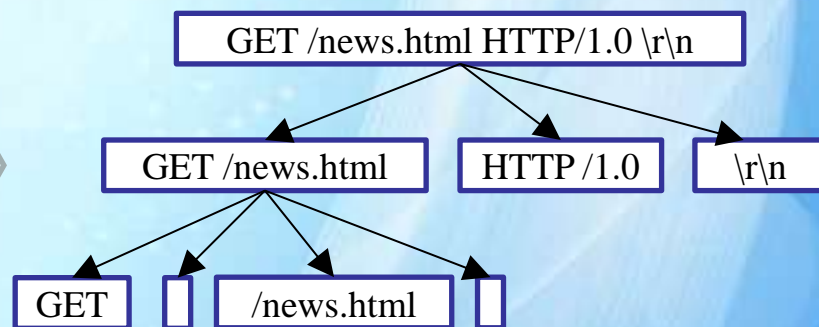
字段树优化

划分字段

字段符号化

冗余节点删除

输入	调用栈	指令地址
0: 'G'	main->run->0x15C57->...	0x4BA56A2
1: 'E'	main->run->0x15C57->...	0x4BA56A2
2: 'T'	main->run->0x15C57->...	0x4BA56A2
...
24: '\n'	main->run->0x15C57->...	0x1F7F3



算法 基于调用栈的字段划分

输入：

污点来源标签集合 T，分别对应接收消息的各个字节

污点传播记录序列（以接受消息为污点源，传播规则为数据依赖关系） **TList**

每一个记录 $t = (o, s, pc)$ （其中指令所处理消息的字节偏移 o，指令的调用栈 s，指令序号 pc）

输出：字段切分结果 **TaintMap**

Define **TaintMap** = { }，，污点来源标签集合 T 到分组标识集合的映射

TList = sort_by_o(**TList**)

group = 0

TaintMap[**TList**[0].o] = 0

For (i = 1; i < **TList**.size() ; i++){

 t_a = **TList**[i-1]

 t_b = **TList**[i]

 if(t_a.s != t_b.s)

 group++;

 EndIf

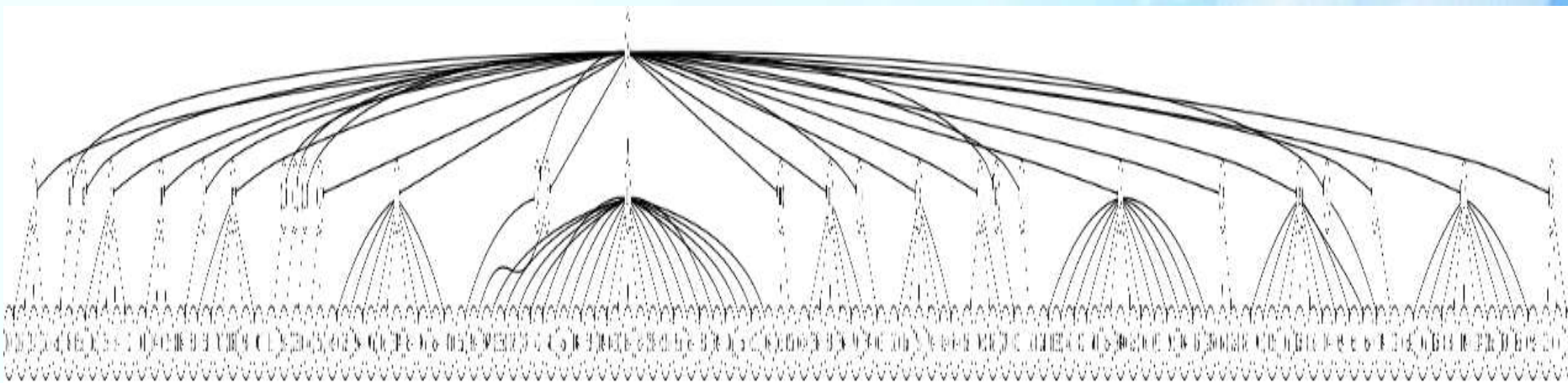
TaintMap[t_b.o]=group

EndFor

- 基于处理指令上下文差异的字段划分
 - 针对Nginx Web服务器的HTTP协议消息字段划分
 - 三层结构，根节点为整个消息，叶子节点为单个字节，中间节点为字段分组信息

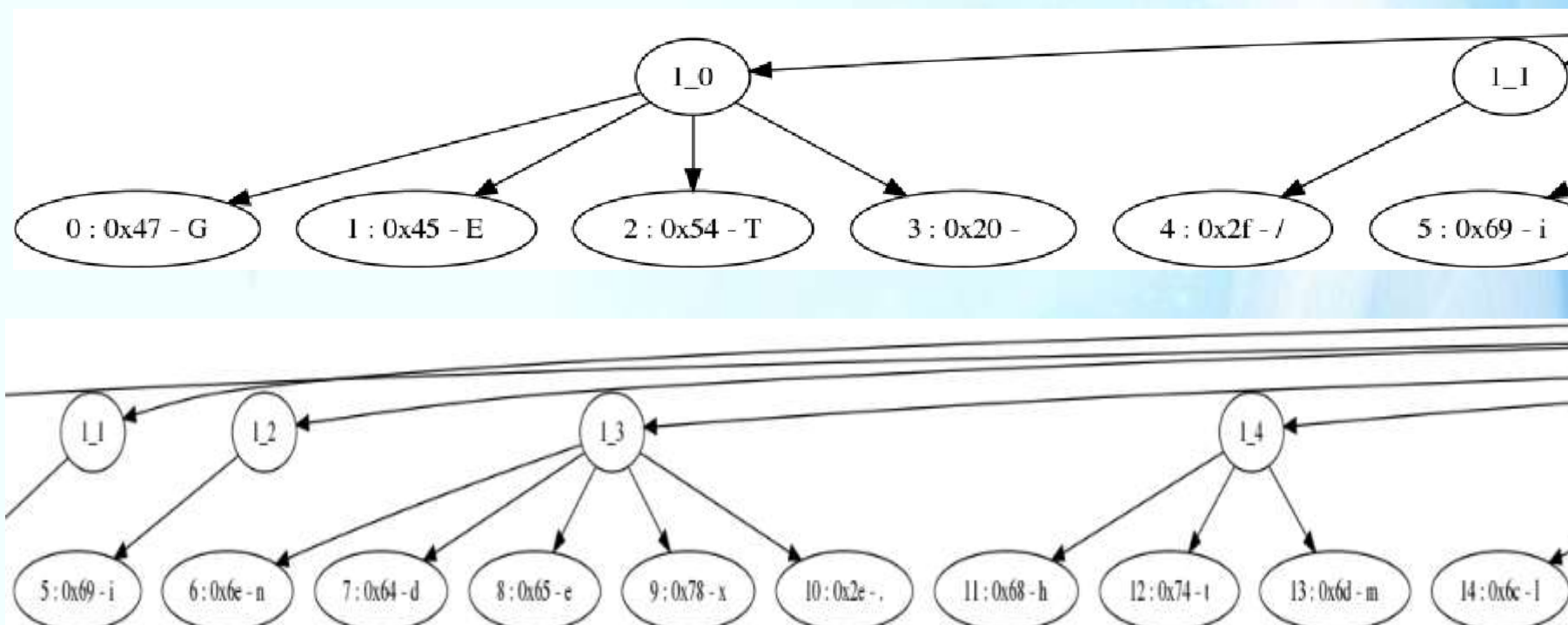
发送消息

```
GET / HTTP/1.1^M
User-Agent: Wget/1.15 (linux-gnu)^M
Accept: /*.*^M
Host: 127.0.0.1:5555^M
Connection: Keep-Alive
```

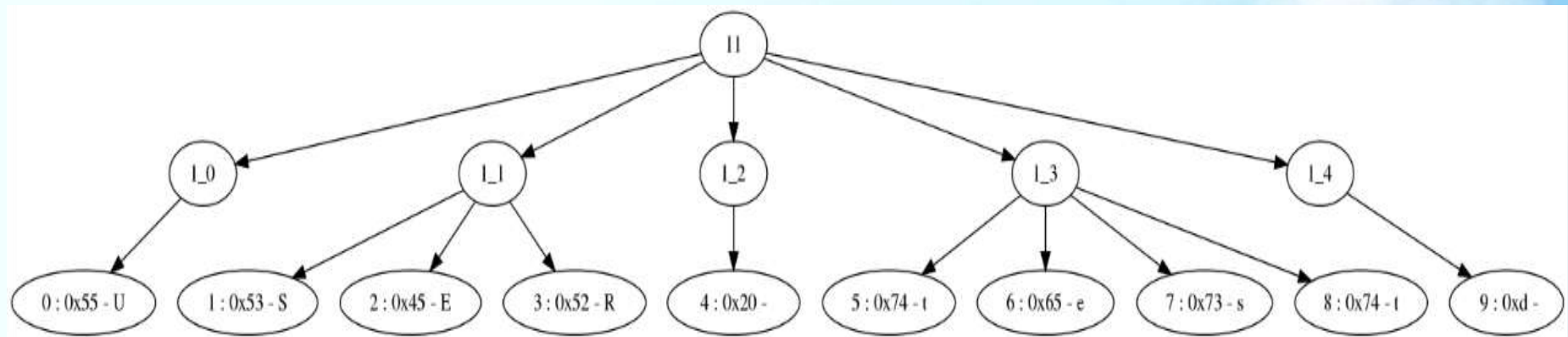


2、协议消息格式逆向 – 字段划分

- 基于处理指令上下文差异的字段划分
 - 针对Nginx Web服务器的HTTP协议消息字段划分 (局部放大)



- 基于处理指令上下文差异的字段划分
 - 针对FileZilla 服务器的FTP协议消息字段划分
 - 三层结构，根节点为整个消息，叶子节点为单个字节，中间节点为字段分组信息
 - 请求（Request）消息内容：USER TEST



- 基于处理指令上下文差异的字段划分小结
 - 基于程序在接受并处理输入消息时不同字段的实时调用栈差异来划分字段
 - 针对服务端进行程序监控
 - 需要利用污点传播筛选指令，同时实时跟踪调用栈变化
 - 随着程序结构的日益复杂，调用栈的字段区分准确度难以保证
 - 宏观结构：程序功能日益复杂，规模更为庞大
 - 微观结构：程序优化、安全防护

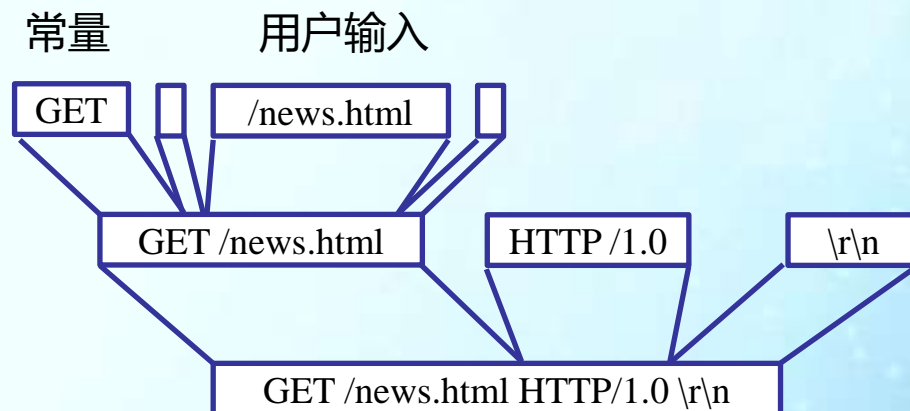
- 基于字段来源回溯的字段划分

- 观察现象：

- 程序构造发送数据包时，需要把分散在内存中的字段数据信息进行封装整合，生成存储在整块缓冲区中的消息数据
 - 从发送数据包构造过程可以逆向推导出发送消息的格式

- 基本思想

- 从程序执行记录进行回溯得到消息中不同字段的来源，并依据其进行字段划分



2、协议消息格式逆向 – 字段划分

- 基于字段来源回溯的字段划分

示例：基于FileZilla Client 源代码的用户登录消息的来源追溯

```
Server.cpp: CServer::GetUser() const
```

```
⑤257: return m_user;
```

```
Ftpcontrolsocket.cpp: CFTPControlSocket::LogonSend
```

```
④1064: res = SendCommand(_T( "USER " )+m_pCurrentServer->GetUser());
```

```
调用 Ftpcontrolsocket.cpp: CFTPControlSocket::SendCommand
```

```
③1183: wxCharBuffer buffer = ConvToServer(str + _T( "\r\n" ));
```

```
②1190: bool res = CRealControlSocket::Send(buffer, len);
```

```
调用 ControlSocket.cpp: CRealControlSocket::Send
```

```
①875: int written = m_pBackend->Write(buffer, len, error);
```

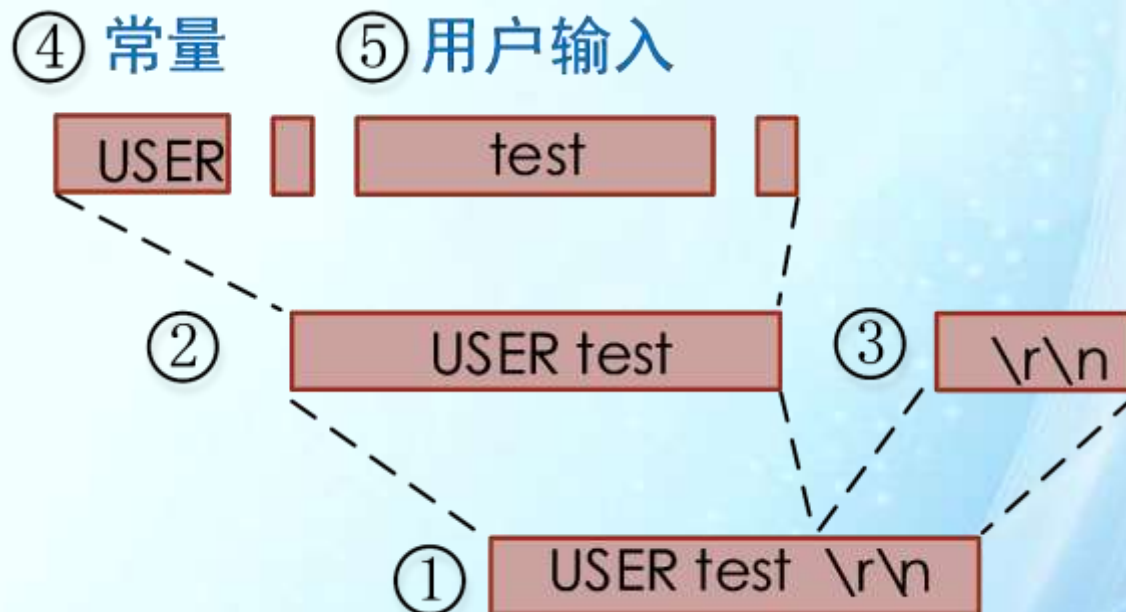
2、协议消息格式逆向 – 字段划分

- 基于字段来源回溯的字段划分

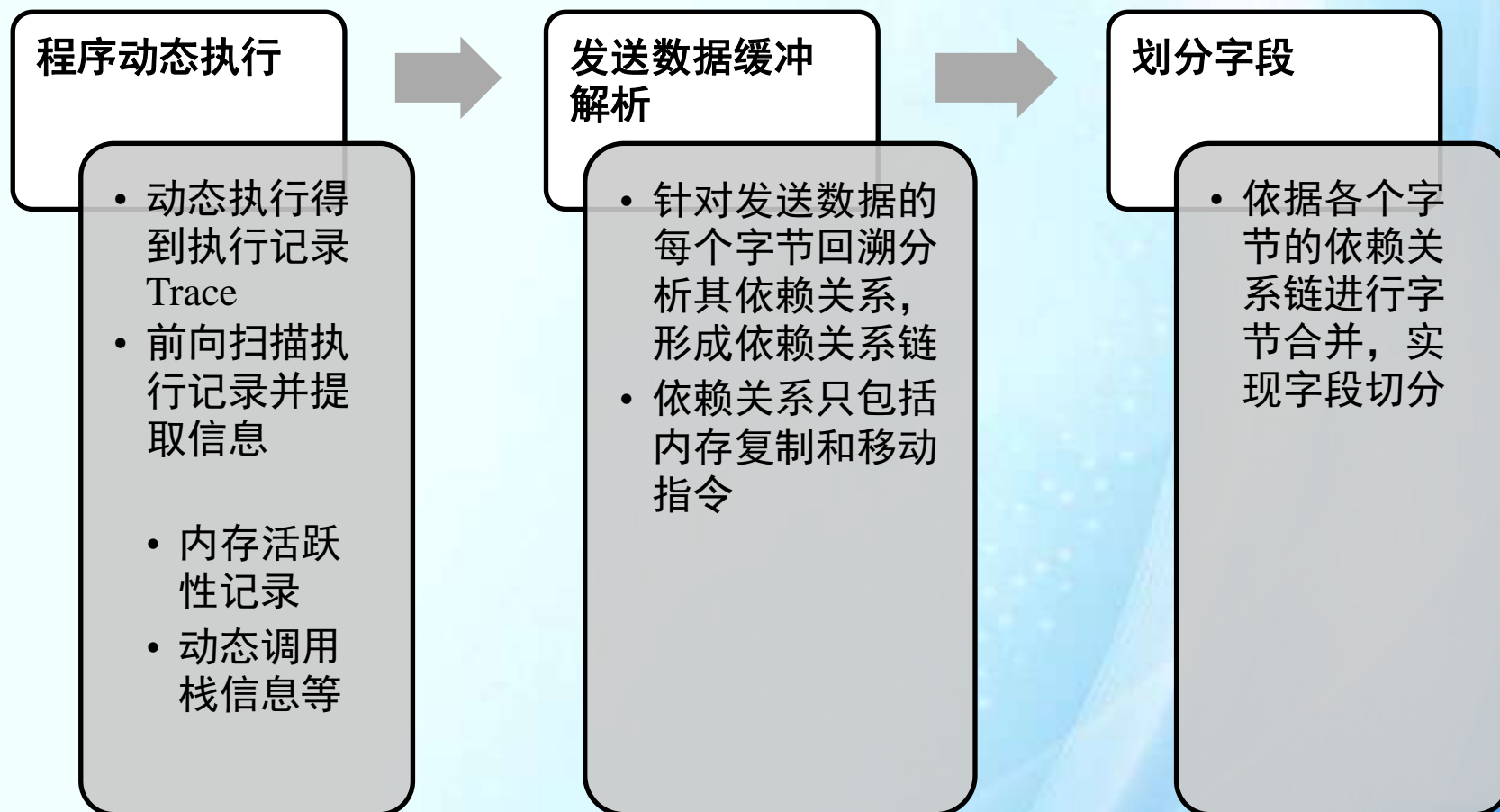
示例：基于FileZilla Client 源代码的用户登录消息的来源追溯

发送消息的构成来源为：

“USER” 来自常量，“test”来自用户输入，“\r\n” 来自常量



- 基于字段来源回溯的字段划分



2、协议消息格式逆向 – 字段划分

算法 基于发送消息来源回溯的字段划分

输入：

程序启动开始到发送消息m之间所有的指令记录T

发送消息中每个字节到其后向数据切片的映射 m_sliceMap

输出：字段划分结果 S

Begin

For m_byte in m:

 m_sliceMap[m_byte] = backward_dataslice(m_byte, T)

group = 0

S[m[0]]=group

For (i=1;i<m.size();i++){

 slice_prev = m[i-1]

 slice_cur = m[i]

 if (not TypeEqual(slice_prev.last, slice_cur.last))

 group++;

 S[slice_cur] = group;

- 基于切片的字段来源回溯

- 切片

- 仅考虑数据依赖关系

- 重点追踪的x86汇编指令

- » mov, movs, movz, push, pop, stos, bswap, shr

- 后向动态切片

- 基于程序执行动态记录的指令记录进行离线分析

- 从网络发送函数所在位置N开始，针对发送函数的发送缓冲区参数buf进行切片

- 切片准则 $\langle N, \{buf\} \rangle$

- 切片终止条件

- 跟踪指令的源操作数为如下类型时停止切片

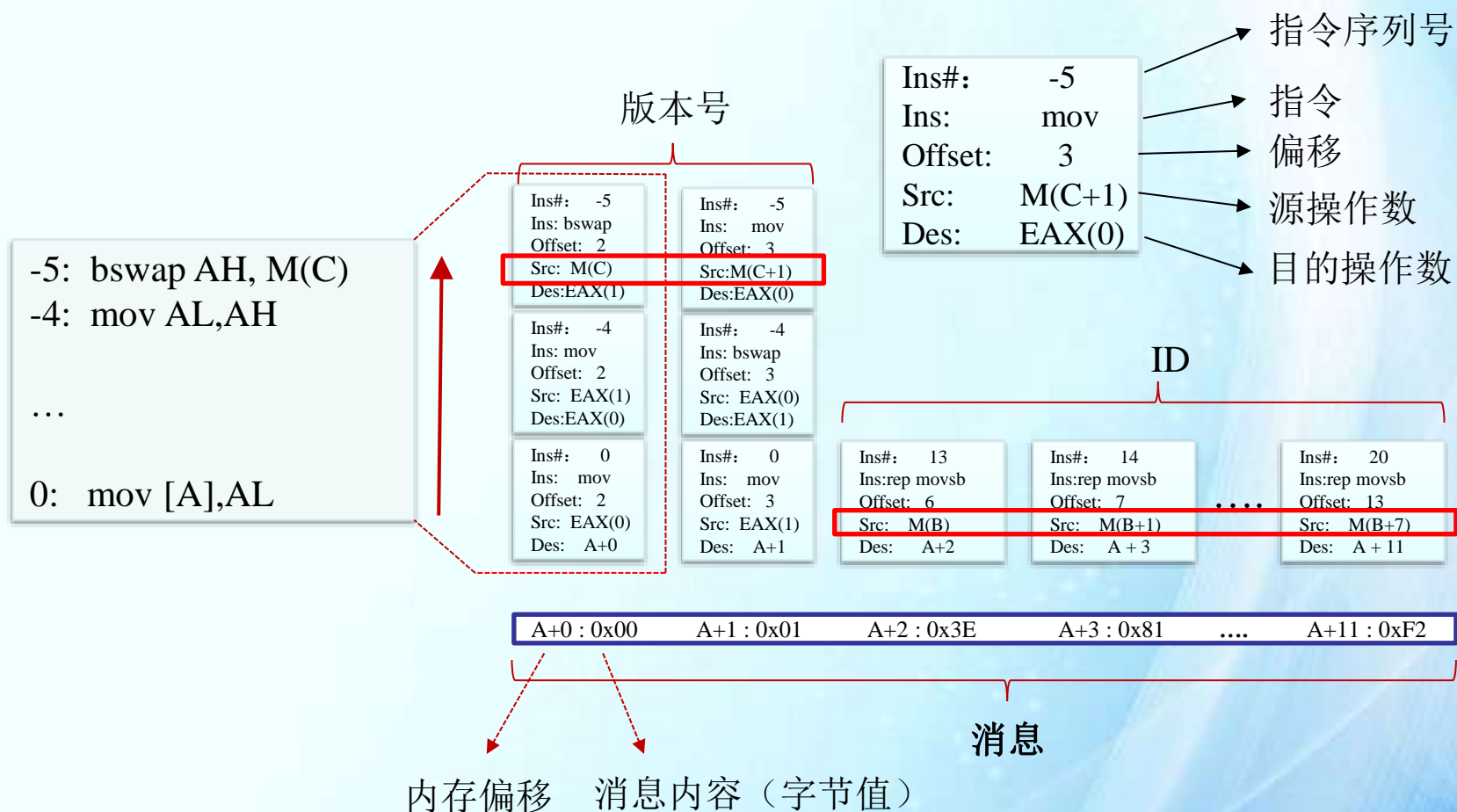
- 立即数

- 存储某个常数的内存区域

- 未知操作数

2、协议消息格式逆向 – 字段划分

- 基于字段来源回溯的字段划分（动态后向数据切片）



- 基于字段来源回溯的字段划分小结
 - 基于程序在发送消息前消息中不同字节的构成来源来划分字段
 - 针对消息发送端（一般为客户端）进行程序监控
 - 需要利用程序执行监控技术提取程序执行指令
 - 基于程序执行指令进行动态后向数据切片

- 小结

- 基于分隔符的字段切分方法适用于部分明文协议（HTTP）,过于依赖分隔符
- 基于执行上下文的分析方法能够分析加密协议，但需要依据实际情况调整调用栈深度，不能完全自动化
- 基于字段来源回溯的分析方法适用性广，准确性较高，同时还能辅助用于语义分析

- 语法分析

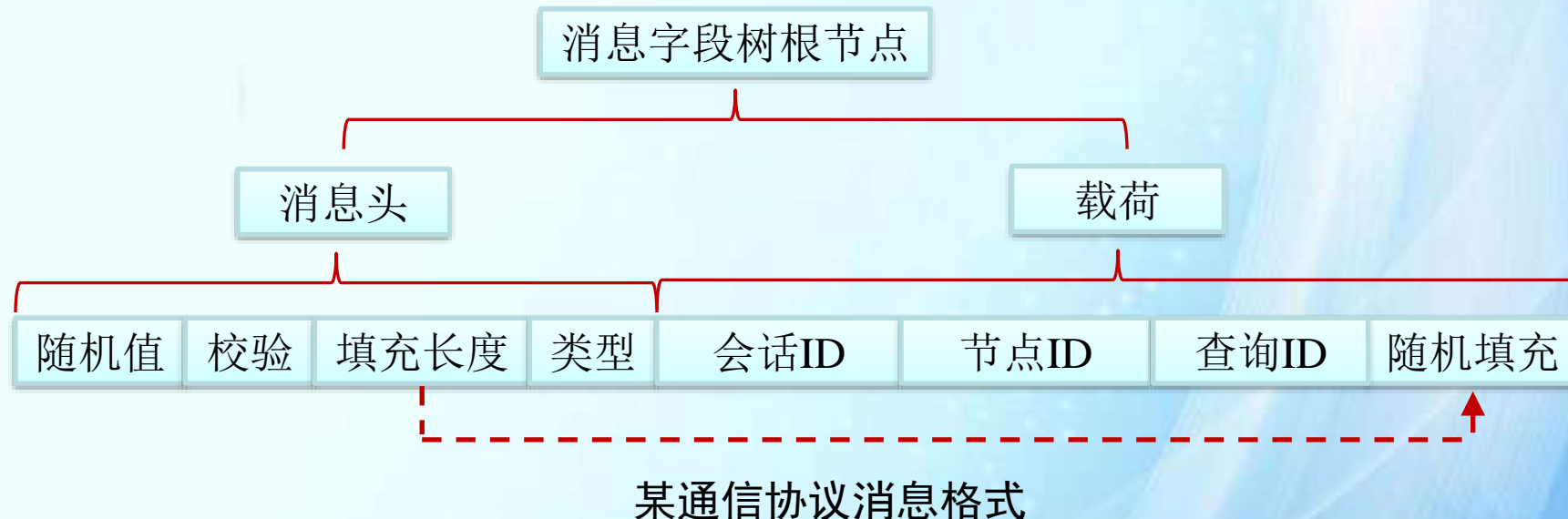
- 字段间关系识别

- 位置型字段

- 长度字段、可变长字段、指针型字段

- 字段层次结构的识别

- 字段从属、字段并行和顺序关系



- 网络协议语法

- 网络协议的RFC通常包含协议消息的语法规范

- 一般用巴克斯 (BNF) 范式描述（上下文无关文法）

```
HTTP-message = Request | Response
```

```
generic-message = start-line
```

```
    *(message-header CRLF)
```

```
    CRLF
```

```
    [ message-body ]
```

```
start-line = Request-Line | Status-Line
```

```
message-header = field-name ":" [ field-value ]
```

```
message-body = entity-body | <entity-body encoded as per Transfer-Encoding>
```

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

```
Method = "OPTIONS" | "GET" | "HEAD" | "POST" | "PUT" | "DELETE" | "TRACE" |  
"CONNECT" | extension-method
```

```
Request-URI = "*" | absoluteURI | abs_path | authority
```

```
....
```

- 网络协议语法结构示例
 - HTTP协议消息语法结构



2、协议消息格式逆向 – 字段间关系的识别

- 网络协议语法结构示例
 - Wireshark截获的HTTP GET请求数据包

0000	52 54 00 12 34 56 52 55	0a 00 02 02 08 00 45 00	RT..4VRUE.
0010	00 a4 01 09 00 00 40 06	61 3b 0a 00 02 02 0a 00@. a;.....
0020	02 0f e7 4b 00 50 00 58	de 02 43 b0 8b 21 50 18	...K.P.X ..C..!P.
0030	22 38 ca 52 00 00 47 45	54 20 2f 69 6e 64 65 78	"8.R..GE T /index
0040	2e 68 74 6d 6c 20 48 54	54 50 2f 31 2e 31 0d 0a	.html HT TP/1.1..
0050	55 73 65 72 2d 41 67 65	6e 74 3a 20 57 67 65 74	User-Age nt: wget
0060	2f 31 2e 31 33 2e 34 20	28 6c 69 6e 75 78 2d 67	/1.13.4 (linux-g
0070	6e 75 29 0d 0a 41 63 63	65 70 74 3a 20 2a 2f 2a	nu)..Acc ept: */*
0080	0d 0a 48 6f 73 74 3a 20	31 32 37 2e 30 2e 30 2e	..Host: 127.0.0.
0090	31 3a 35 35 35 35 0d 0a	43 6f 6e 6e 65 63 74 69	1:5555.. Connecti
00a0	6f 6e 3a 20 4b 65 65 70	2d 41 6c 69 76 65 0d 0a	on: Keep -Alive..
00b0	0d 0a		..

```
⊞ Frame 4: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits)
⊞ Ethernet II, Src: 52:55:0a:00:02:02 (52:55:0a:00:02:02), Dst: RealtekU_12:34:56 (52:54:00:12:34:56)
⊞ Internet Protocol Version 4, Src: 10.0.2.2 (10.0.2.2), Dst: 10.0.2.15 (10.0.2.15)
⊞ Transmission Control Protocol, Src Port: 59211 (59211), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 124
⊞ Hypertext Transfer Protocol
  ⊞ GET /index.html HTTP/1.1\r\n
    User-Agent: wget/1.13.4 (linux-gnu)\r\n
    Accept: */*\r\n
    Host: 127.0.0.1:5555\r\n
    Connection: Keep-Alive\r\n
    \r\n
    [Full request URI: http://127.0.0.1:5555/index.html]
    [HTTP request 1/1]
    [Response in frame: 5]
```

- 字段间关系识别的三种主要方法
 - 分隔符作用域
 - 调用栈
 - 字段来源回溯

- 分隔符作用域

- 定义

- 分隔符与输入消息（污点标签）的字节序列做连续比较的范围
 - 定义分隔符为s，污点标签为 $t_i (i=1,2,\dots,n)$ ，谓词 $\text{CmpTaint}(s, t_i)$ 表示分隔符s与污点标签 t_i 作过比较，则分隔符作用域 $\text{Range}(s)$ 的形式化定义

- $\text{Range}(s) = \{ j, j+1, \dots, j+k \mid \text{CmpTaint}(s, t_{j+i}), i=0,1,\dots,k \}$

- 示例

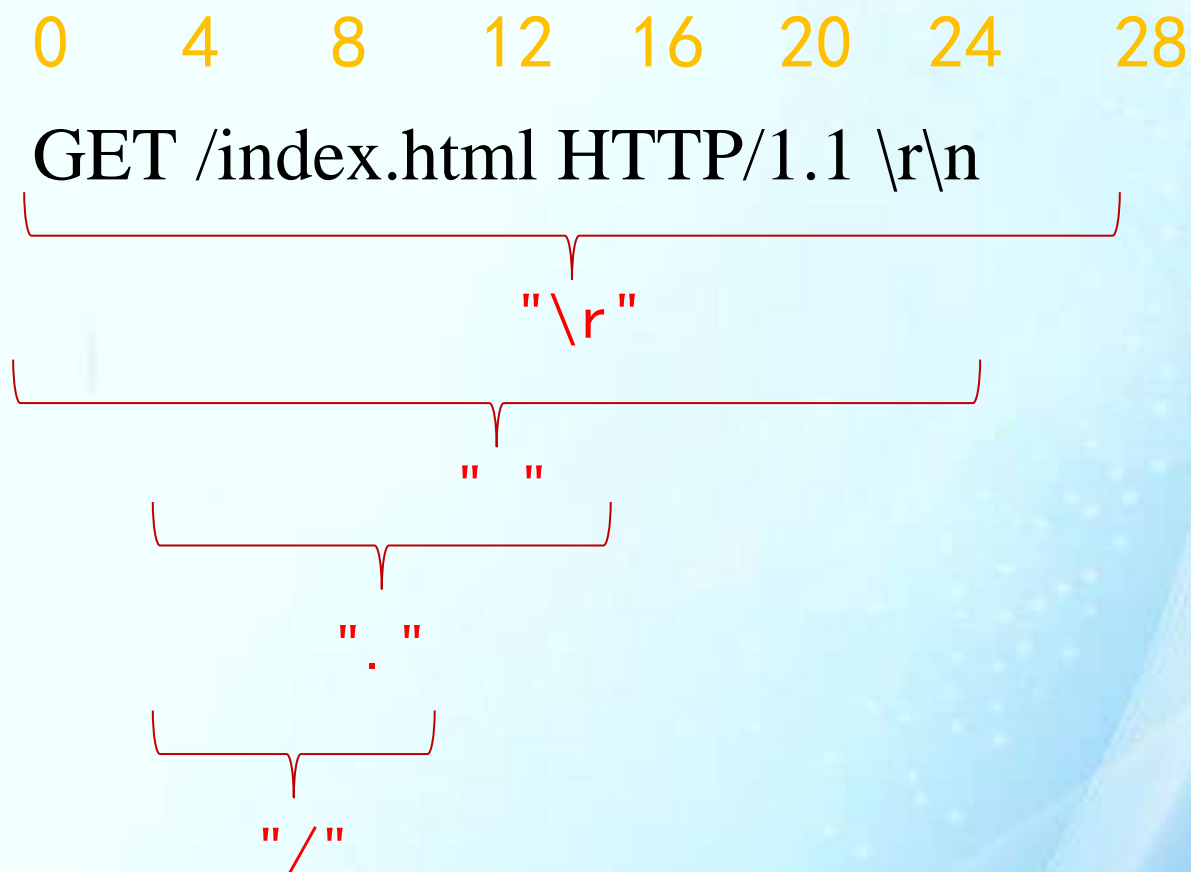
- Nginx服务器处理HTTP GET 消息时的分隔符作用域

分隔常数	作用域
“\r”:0xA	[0,25]
“ ”:0x20	[0,23]
“.”:	[4,15]
“/”	[4,9]

2、协议消息格式逆向 – 字段间关系的识别

- 分隔符作用域

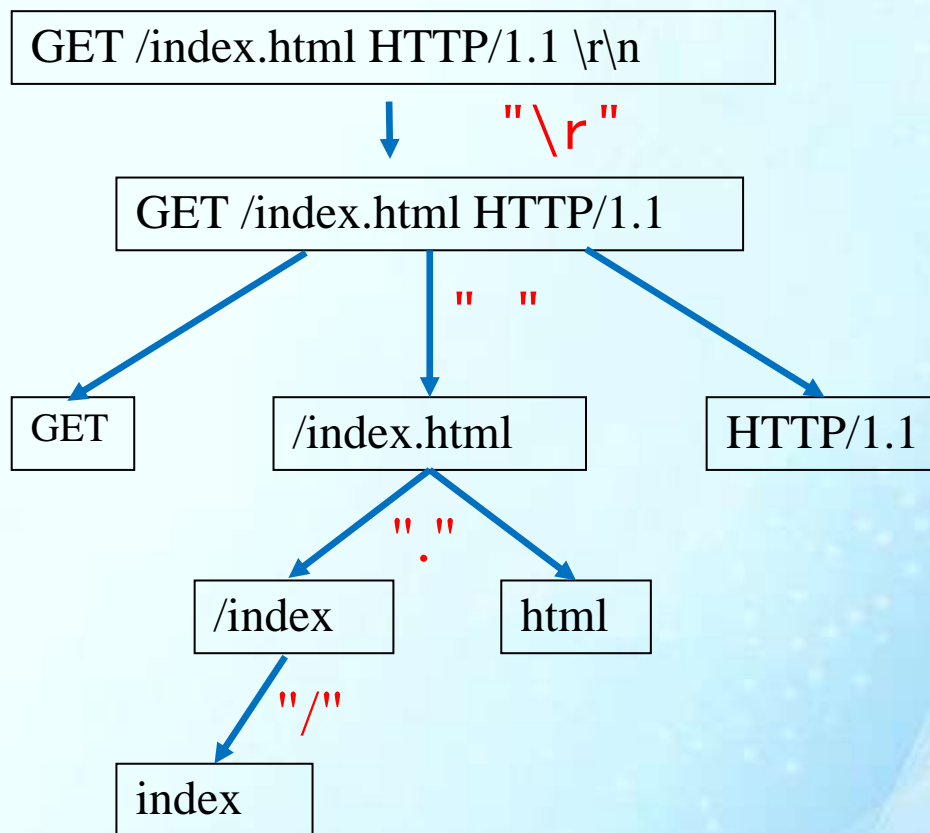
- 示例



2、协议消息格式逆向 – 字段间关系的识别

- 分隔符作用域

- 示例



2、协议消息格式逆向 – 字段间关系的识别

算法 基于分隔符的包含从属关系识别

输入：

分隔符集合 S , 污点来源标签集合 T , 分别对应接收消息的各个字节

$\text{SepRangeMap } S \rightarrow \{ \text{Range}(s_i) \mid s_i \in S \}$ // 字节分隔符到其作用域的映射

输出：字段从属关系映射 $R : \{ \}$

Begin

For s_i , $\text{Range}(s_i)$ in SepRangeMap :

For s_j , $\text{Range}(s_j)$ in SepRangeMap :

If $\text{Range}(s_i) \subset \text{Range}(s_j)$: //真包含于

$R(s_i, s_j) = 1$ // s_i 从属于 s_j

Else if $\text{Range}(s_j) \subset \text{Range}(s_i)$:

$R(s_j, s_i) = 1$ // s_j 从属于 s_i

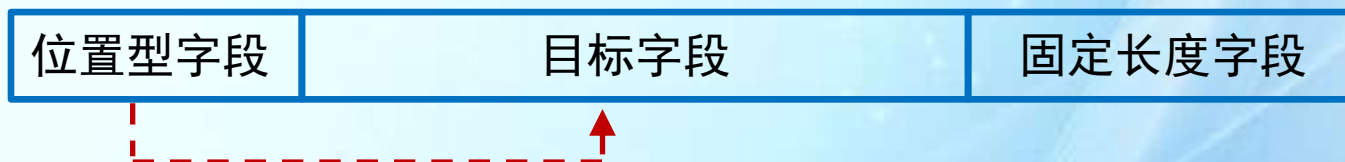
Else if $\text{Range}(s_j) \wedge \text{Range}(s_i) \neq \phi$:

$R(s_j, s_i) = -1$ // s_j 与 s_i 存在交叉

Else if $\text{Range}(s_j) \wedge \text{Range}(s_i) == \phi$:

$R(s_j, s_i) = 0$ // s_j 与 s_i 为并列字段

- 位置型字段
 - 定义：该字段包含了指示同一个消息中其他字段位置的信息
 - 目标字段：位置型字段所指示位置信息的字段
- 常见位置型字段
 - 长度字段
 - 字段值是另外一个字段的长度（以字节计数）
 - 指针型字段
 - 字段值是另外一个字段开始位置的相对或绝对偏移
 - DNS 应答(Response)中资源记录的域名指针
 - 计数字段
 - 字段值是另外一组相同类型字段的数量
 - DNS协议中权威服务器记录的数量



- 位置型字段识别的两个基本思路

- 间接寻址：程序在读取消息中目标字段值时，在获取（或计算）目标字段的访问地址时引用了位置型字段



$$A = \text{Sizeof}(B)$$

$$\text{Offset}(C) = \text{Base} + \text{Sizeof}(A) + A$$

- 循环：程序循环读取消息中目标字段值，其中循环结束条件的计算引用了位置型字段

```
for ( int I = Base + Sizeof(A) ; I < Base + sizeof(A) + A; I++)  
    *I += 1;
```

- 基于间接寻址的位置型长度字段识别

- 间接寻址的指令特点

- 输入消息标记为污点，则如下指令将被污点标记

- ① $\text{Offset}(C) = \text{Base} + \text{Sizeof}(A) + A$

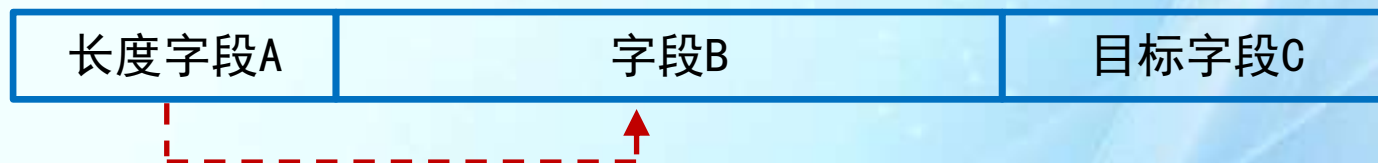
- ② $\text{Buffer}[\text{Offset}(C)] += 1$

A 为污点， $\text{Offset}(C)$ 、 $\text{Buffer}[\text{Offset}(C)]$ 也为污点

示例汇编指令：mov edx, [ebx + eax + 4]

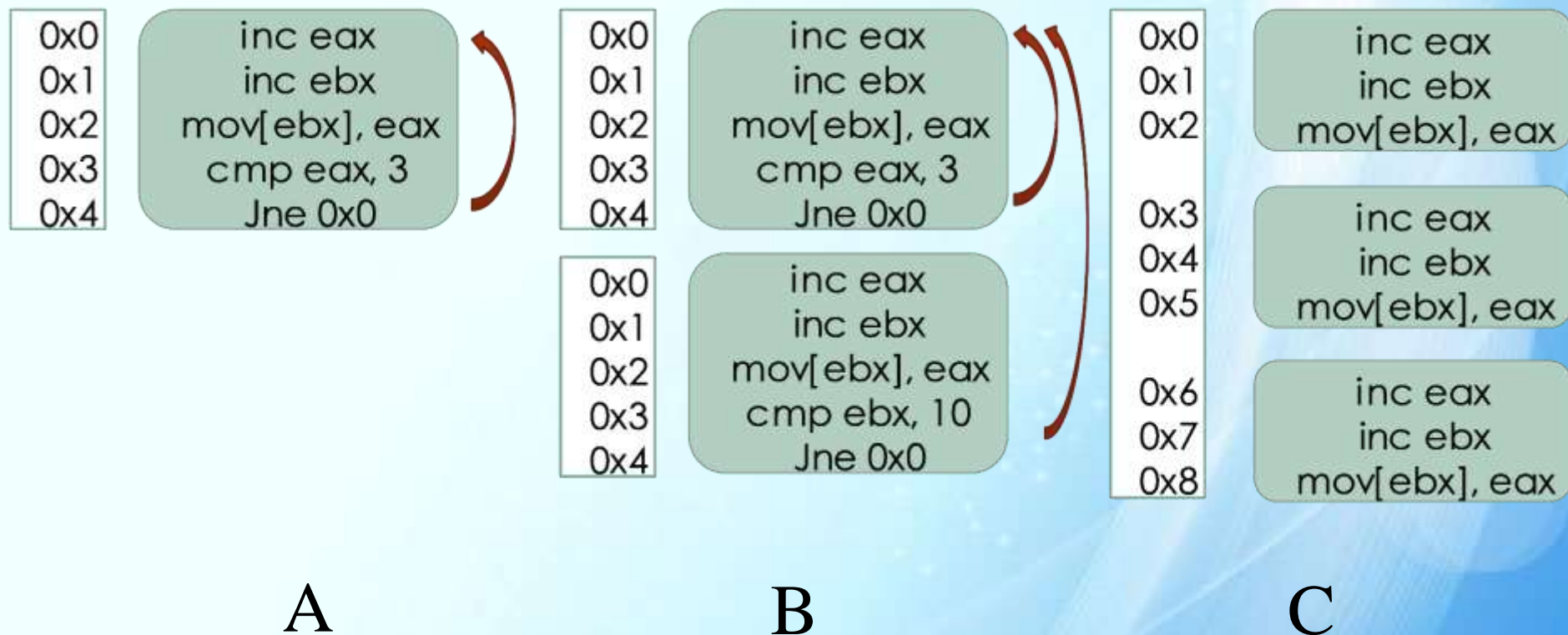
- 间接寻址的位置型字段识别

- 访问一个污点标记的区域
 - 访问使用的地址是由污点标记的数据计算得到的



2、协议消息格式逆向 – 字段间关系的识别

- 基于循环的位置型长度字段识别
 - 基于汇编指令的循环识别
 - A. 基于回边跳转的循环识别
 - B. 循环嵌套
 - C. 基于重复执行指令块的循环识别



- 基于循环的位置型长度字段识别

```
for ( int I = Base + Sizeof(A) ; I < Base + sizeof(A) + A; I++)  
    *I += 1;
```

- 方法

- 静态循环识别：基于跳转回边的循环识别

- 获取每个循环的结束条件对应的比较指令
- 获取每条指令所属循环的集合

- 动态执行指令分析

- 基于输入消息进行污点传播
- 若污点相关的指令属于某个循环结束条件（即该循环结束条件被污点控制），则该循环条件相关的污点标记字段是潜在长度字段
- 若每一次循环执行时，结束条件都引用了相同的污点字段，则该字段判定为长度字段

2、协议消息格式逆向 – 字段间关系的识别

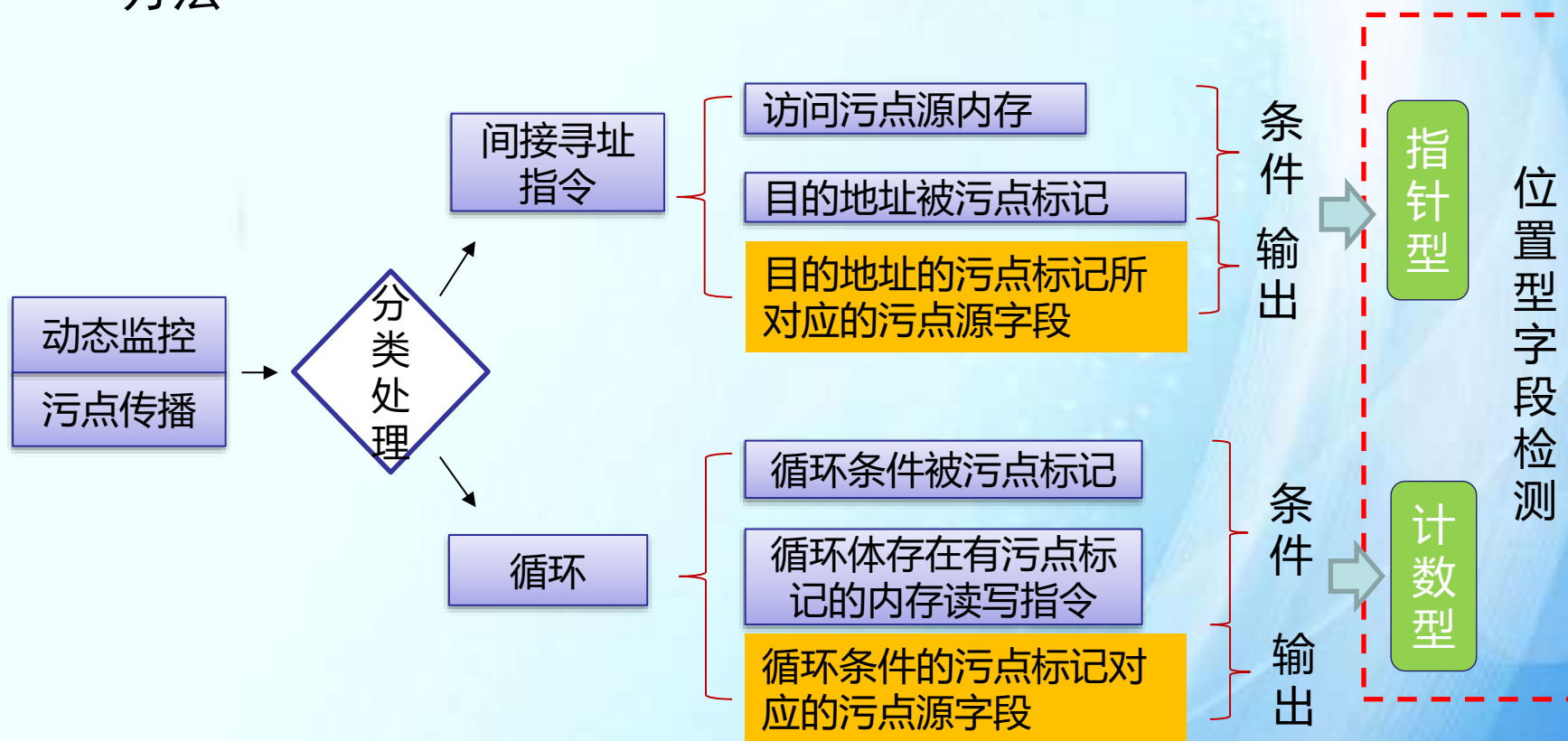
- 位置型字段的识别 (Polyglot)

- 基本思想:

- 将消息中某字段的读写操作所依赖的字段视为位置型字段

- 如指令 `mov edx, [eax + ecx]` 中的 `ecx`

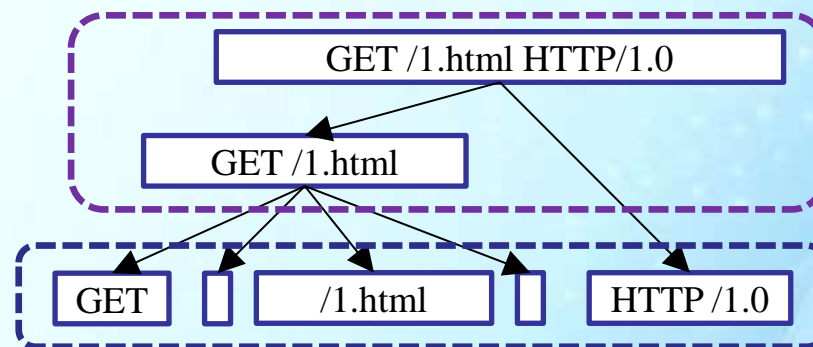
- 方法



2、协议消息格式逆向 – 字段间关系的识别

- 字段层次结构的识别（Autoformat）
 - 依据消息处理指令上下文的差异对字段进行分类重组

输入	调用栈	指令地址
0: 'G'	main->run->0x15C57->...->0x2D667->apr_brigade_split_line->memchr	0x4BA56A2
1: 'E'	main->run->0x15C57->...->0x2D667->apr_brigade_split_line->memchr	0x4BA56A2
2: 'T'	main->run->0x15C57->...->0x2D667->apr_brigade_split_line->memchr	0x4BA56A2
3: ' '	main->run->0x15C57->...->0x2D667->apr_brigade_split_line->strtok	0x4BA51F3
4: '/'	main->run->0x15C57->...->0x2D667->ap_read_request->ap_rgetline_core	0x26AFB
5: '1'	main->run->0x15C57->...->0x2D667->ap_read_request->ap_rgetline_core	0x26AFB
6: '.'	main->run->0x15C57->...->0x2D667->ap_read_request->ap_rgetline_core	0x26AFB
.....		
12: 'H'	main->run->0x15C57->... 0xF5A8->ap_read_request->ap_getword_white	0x1F7F3
13: 'T'	main->run->0x15C57->... 0xF5A8->ap_read_request->ap_getword_white	0x1F7F3



- 字段语义恢复
 - 概述
 - 关键字识别
 - 比较指令分析
 - 污点传播
 - 基于运行环境语义的字段语义恢复
 - 污点传播和程序切片

- 字段语义恢复

- 目标:

- 通过软件逆向分析恢复出目标网络协议中各个消息每个字段的功能、用途等语义信息

- 可提取的语义信息主要来自:

- 协议实现自身
 - 程序附带的调试信息，以及程序使用手册等其他语义信息
 - 运行环境
 - 程序依赖的运行环境，如C语言运行时，Java虚拟机等
 - 底层操作系统以及底层硬件
 - Windows操作系统，CPU各寄存器信息等

2、协议消息格式逆向 – 字段功能语义恢复



- 字段语义恢复
 - 可获取的语义信息示例

寄存器	Intel手册提供的原始语义
EAX	累加器寄存器 ret指令返回值
ECX	loop 指令循环长度
EDX	除法指令的余数
EBX	内存寻址的基地址
ESI	rep mov 指令源地址
EDI	rep mov 指令目的地址
EBP	框架指针
ESP	栈指针

调用约定 寄存器	Windows MSVC cdecl	Borland fastcall	Linux 内核 系统调用
调用参数	[ebp+8+4*参数索引]	第一参数: EAX 第二参数: EDX 第三参数: ECX 其余参数入栈	第一参数: EAX 第二参数: EBX 第三参数: ECX 第四参数: EDX ...
返回值	EAX	EAX	EAX

2、协议消息格式逆向 – 字段功能语义恢复



- 字段语义恢复
 - 二进制程序反汇编

```
void __thiscall CControlSocket::OnReceive(CControlSocket *this, int nErrorCode)
{
    CControlSocket *v2; // esi@1
    __int64 v3; // qax@4
    int v4; // edi@4
    void *v5; // eax@9
    void *v6; // ebx@9
    int v7; // eax@9
    unsigned __int8 v8; // cf@12
    int v9; // edi@13
    signed int v10; // eax@14
    char v11; // cl@18
    int v12; // [sp+Ch] [bp-34h]@1
    int v13; // [sp+10h] [bp-30h]@4
    CStdStr<char> thisa; // [sp+14h] [bp-2Ch]@26
    unsigned int v15; // [sp+2Ch] [bp-14h]@1
    int v16; // [sp+3Ch] [bp-4h]@26

    v15 = (unsigned int)&v12 ^ __security_cookie;
    v2 = this;
    if ( !this->n_antiHammeringWaitTime )
    {
        v12 = 500;
        v3 = CControlSocket::GetSpeedLimit(this, upload);
        v13 = HIWORD(v3);
        v4 = v3;
        if ( !v3 )
        {
            CControlSocket::ParseCommand(v2);
            return;
        }
        if ( v3 < 500 && (HIWORD(v3) & (unsigned int)v3) != -1 )
            v12 = v3;
        v5 = operator new__(0x1F4u);
```

```
int __thiscall sub_446C20(void *this, int a2)
{
    int result; // eax@1
    int v3; // esi@1
    signed __int64 v4; // qax@4
    int v5; // edi@4
    int v6; // eax@9
    void *v7; // ebx@9
    int v8; // eax@9
    unsigned __int8 v9; // cf@12
    int v10; // edi@13
    signed int v11; // eax@14
    char v12; // cl@18
    int v13; // [sp+10h] [bp-34h]@1
    int v14; // [sp+14h] [bp-30h]@4
    int v15; // [sp+18h] [bp-2Ch]@26
    int v16; // [sp+28h] [bp-1Ch]@28
    unsigned int v17; // [sp+2Ch] [bp-18h]@26
    unsigned int v18; // [sp+30h] [bp-14h]@1
    int v19; // [sp+38h] [bp-Ch]@1
    int v20; // [sp+40h] [bp-4h]@26

    v18 = (unsigned int)&v13 ^ __security_cookie;
    result = (int)&v19;
    v3 = (int)this;
    if ( !*((_DWORD *)this + 613) )
    {
        v13 = 500;
        LODWORD(v4) = sub_43EFE0(1);
        v14 = HIWORD(v4);
        v5 = v4;
        if ( !v4 )
            return sub_440710(v3);
        if ( v4 < 500 && (HIWORD(v4) & (unsigned int)v4) != -1 )
            v13 = v4;
        v6 = sub_419B4D(500);
```

- 基于字符串比较指令分析的关键字识别

- 关键字定义

- 协议规范中规定的具有特定含义的字段，通用以常量字符串的形式定义

- HTTP协议中表示请求消息类型的 “GET” 常量

- FTP协议中表示用户登录请求消息类型的 “USER” 常量

```
switch (len)
{
case 3:
    switch (method[0])
    {
    case 'P':
        return (method[1] == 'U'
            && method[2] == 'T'
            ? M_PUT : UNKNOWN_METHOD);
    case 'G':
        return (method[1] == 'E'
            && method[2] == 'T'
            ? M_GET : UNKNOWN_METHOD);
    default:
        return UNKNOWN_METHOD;
    }
}
```

Apache HTTP 服务器解析GET关键字

- 基于字符串比较指令分析的关键字识别方法
 - 提取所有污点相关比较指令（与分隔符提取相同）
 - cmp , test, sub
 - 提取比较指令执行结果为“成功”的所有指令（与分隔符提取不同）
 - 检查指令执行完毕后相应标志位是否设置
 - 扫描静态二进制文件，是否存在指定关键字
 - 通常“关键字”都是协议规范的一部分，会以硬编码形式存在

- 基于污点传播的关键字识别方法

- 标记可执行文件中可能存储有静态数据的区域为污点区域

- 标记可执行文件自身以及相关依赖库

- PE文件（包括EXE和DLL）的idata, rdata, data段

- 开始污点传播，网络消息发送函数的缓冲区部分区域若被污点标记，则该区域为关键字

- 基于运行环境语义的字段语义恢复

- 语义来源

- 软件运行所依赖的功能库软件、底层操作系统和其他相关支撑软件

操作系统	基础功能库软件	其他扩展功能库 (TLS实现为例)	系统调用
Windows	MSVCRT, MFC .Net JRE Python	Schannel OpenSSL	Ntdll.dll (pdb)
Linux	GLIBC JRE Python	GnuTLS OpenSSL	System.map
Mac	Objective C Runtime	SecureTransport	
Android	NDK, Adnroid SDK	BoringSSL	System.map

- 基于运行环境语义的字段语义恢复
 - 人工追溯字段语义来源示例
 - FilZilla Server 0.9.43 接受 RETR 命令 (下载文件)
 - RETR 命令格式示例：RETR README.txt

ControlSocket.cpp

```
519: void CControlSocket::ParseCommand()
520: {
525:     CStdString command;
526:     CStdString args;
527:     if (!GetCommand(command, args))
528:         return;
    ....
1075: case COMMAND_RETR:
1076: {
    ....
1102:     CStdString physicalFile, logicalFile;
1103:     int error = m_pOwner->m_pPermissions->CheckFilePermissions(m_status.user,
args, m_CurrentServerDir, FOP_READ, physicalFile, logicalFile);
}
```

- 基于运行环境语义的字段语义恢复
 - 人工追溯字段语义来源示例

Permissions.cpp

```
694: int CPermissions::CheckFilePermissions(LPCTSTR username, CStdString filename,  
CStdString currentdir, int op, CStdString& physicalFile, CStdString& logicalFile)  
695: {  
    ...  
740:     physicalFile = directory.dir + "\\\" + filename;  
741:     DWORD nAttributes = GetFileAttributes(physicalFile);  
}
```

转载自MSDN

1.1 Syntax

```
DWORD WINAPI GetFileAttributes(  
    _In_ LPCTSTR lpFileName  
);
```

1.2 Parameters

lpFileName [in]

The name of the file or directory.

In the ANSI version of this function, the name is limited to **MAX_PATH** characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\\?\" to the path.

- 字段语义恢复

- 字段的应用层语义

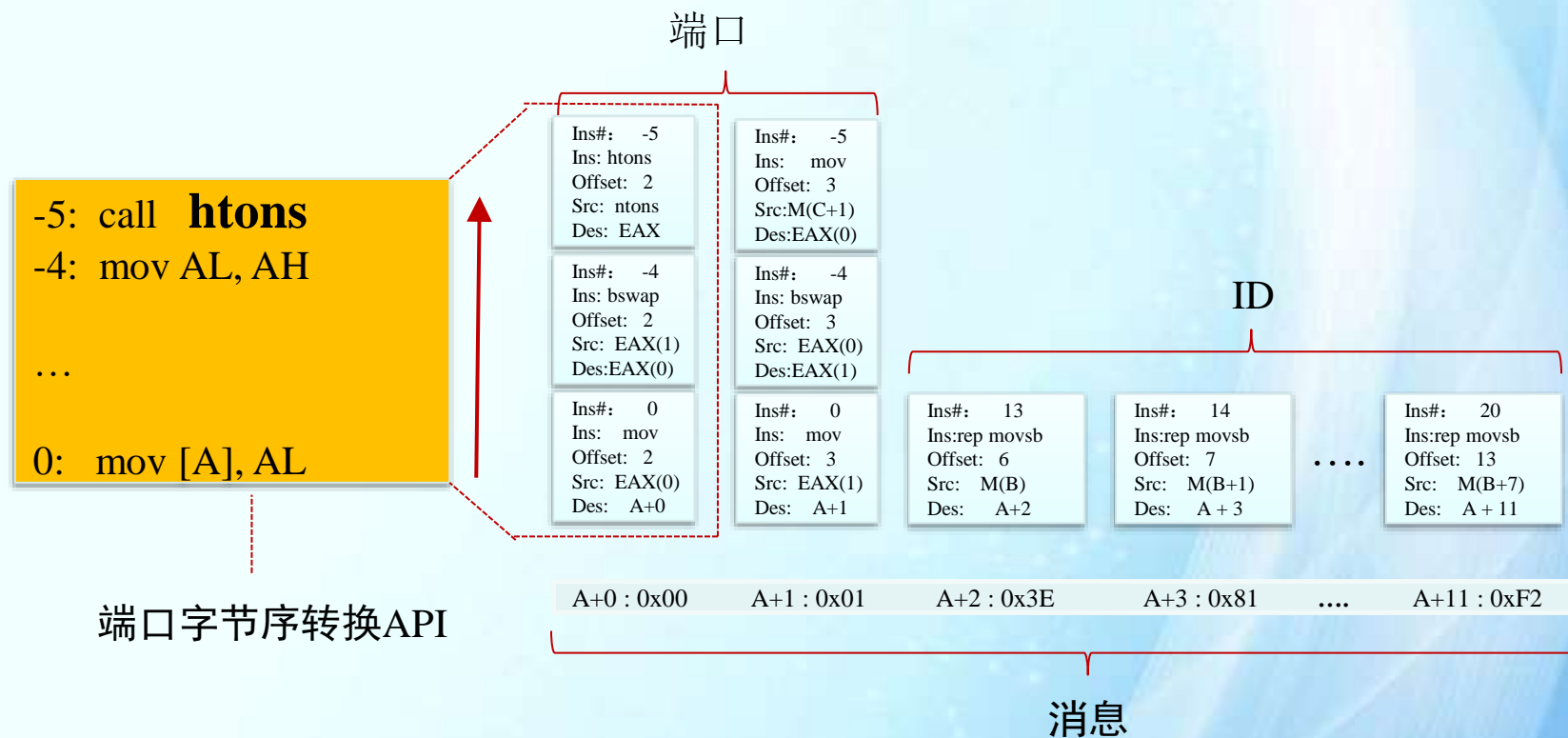
- IP地址、错误状态码、哈希值、文件名、长度等

- 语义逆向方法

- 建立目标软件宿主操作系统的系统调用列表，提取其参数和返回值语义，建立语义数据库
 - 发送消息的后向切片：
 - 从待发送数据开始作后向动态数据切片，若切片结果表明数据来自于某系统调用函数，则将系统函数调用的语义赋予待发送数据
 - 接收消息的污点分析（前向切片）：
 - 标记接受到的数据为污点，污点传播过程中若有已知语义的函数的相关参数或返回值被污点标记，则将相应语义赋予污点标记对应的接受到的数据

2、协议消息格式逆向 – 字段功能语义恢复

- 语义分析过程图示



- 小结

- 位置型字段的识别对于确定长度字段十分有效，无法适用于完整的语法分析
- 依赖消息处理指令上下文的字段层次结构识别在分析中尚存在切分过细，仍需要人工干预
- 基于后向数据切片和污点传播的分析方法能够有效获取部分与系统API直接相关的字段语义，其他字段语义的获取仍然需要人工参与

- 目标
 - 恢复消息交互过程：消息交互的次序和依赖关系
- 主要问题
 - 协议消息类型识别
 - 提取消息的类型，将多个消息按照类型进行分类
 - 状态机推断和化简
 - 提取二进制网络协议实现可接受的网络消息序列
 - 基于红-蓝框架的Exbar方法的推断：Prospex (S&P'09)
 - 基于公共子序列 LCS 的推断：PEXT (WCRE'07)
 - 基于L*算法的学习：Cho等人 (CCS'10)
 - 基于QSM算法的学习：Zhao Zhang等人 (ICDMA'12)

3、协议状态机恢复

- FTP协议状态机示例

- FTP协议是基于请求应答机制的协议

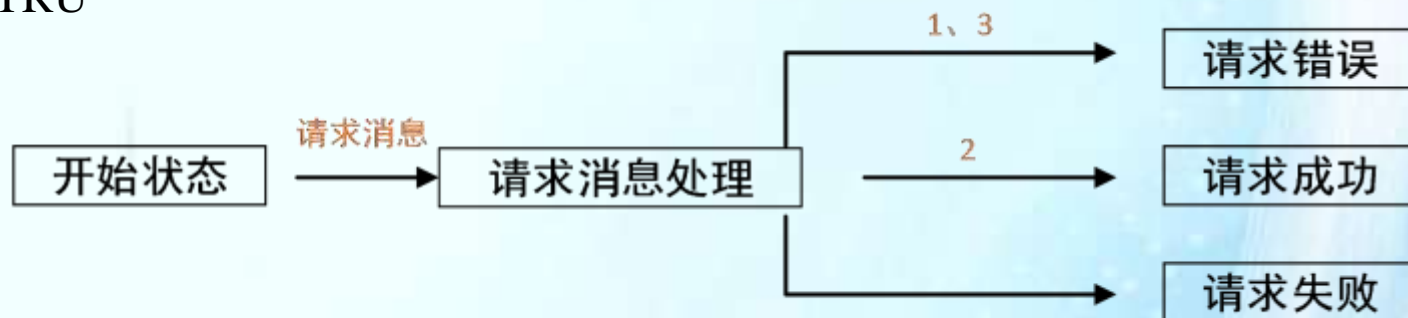
- 协议的每一个命令都有一套状态转换规则

ABOR, ALLO, DELE, CWD, CDUP, SMNT,
HELP, MODE, NOOP, PASV, QUIT, SITE,
PORT, SYST, STAT, RMD, MKD, PWD,
STRU

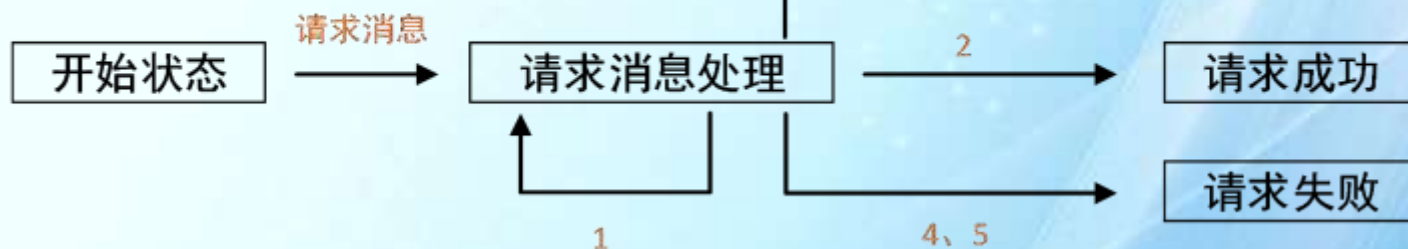
APPE, LIST, NLST, REIN,
RETR, STOR和STOU

A

A



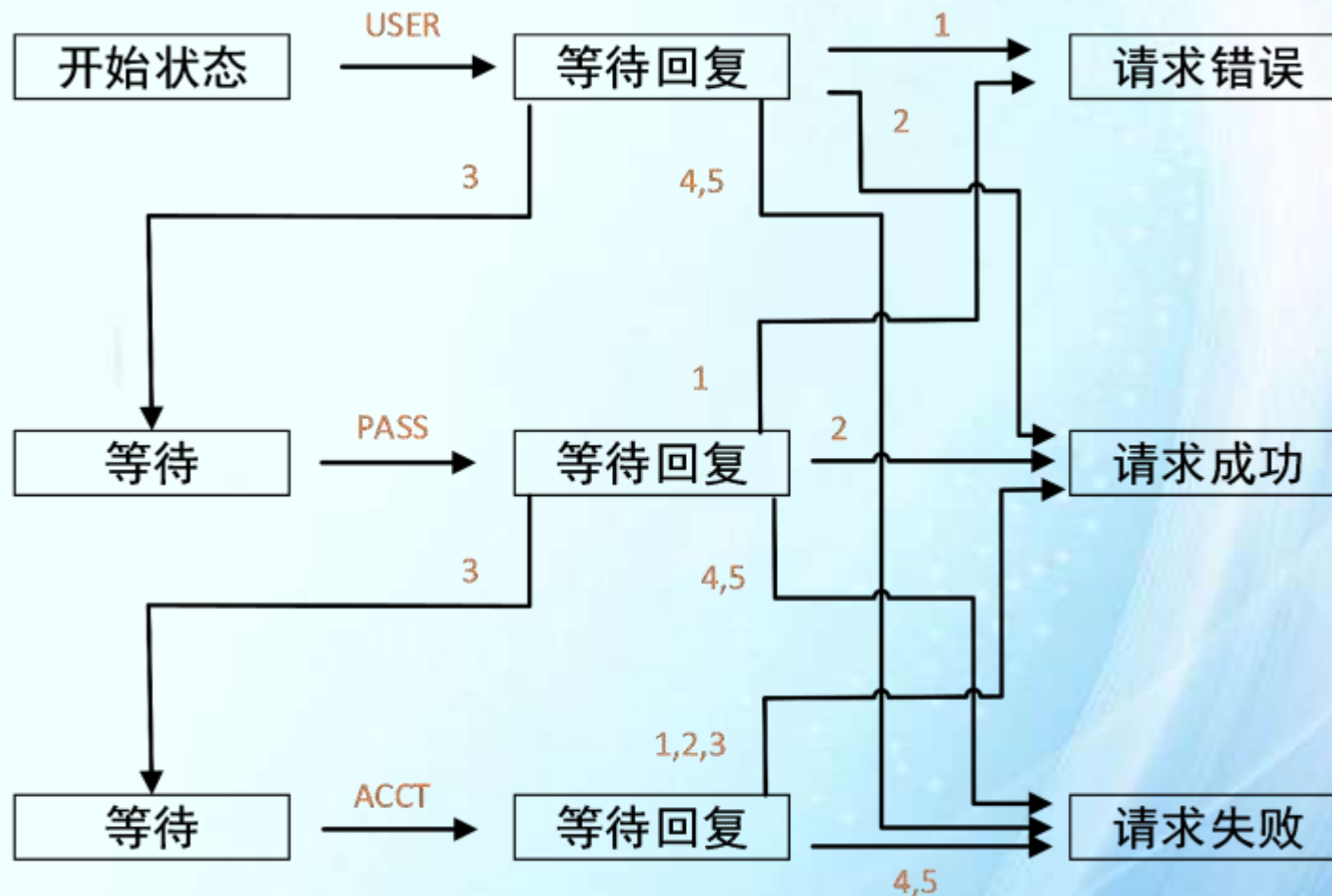
B



3、协议状态机恢复

- FTP协议状态机示例

- FTP协议用户登录状态转换过程



3、协议状态机恢复

- 协议消息类型识别

- 从FTP RFC规范入手认识消息类型

序号	协议	内容	消息的16进制编码
4	FTP	Response: 220-FileZilla Server version 0.9.41 beta	0x3232302D46696C655A
5	FTP	Response: 220-written by Tim Kosse (Tim.Kosse@gmx.de)	0x3232302D777269747A
7	FTP	Response: 220 Please visit http://sourceforge.net/projects/filezilla/	0x32323020506C656173
8	FTP	Request: USER anonymous	0x5553455220616E6F6E
10	FTP	Response: 331 Password required for anonymous	0x333331205061737377
11	FTP	Request: PASS User@	0x504153532055736572
13	FTP	Response: 530 Not logged in, user account has been disabled	0x353330204E6F74206C
20	FTP	Response: 220-FileZilla Server version 0.9.41 beta	0x3232302D46696C655A
21	FTP	Response: 220-written by Tim Kosse (Tim.Kosse@gmx.de)	0x3232302D777269747A
23	FTP	Response: 220 Please visit http://sourceforge.net/projects/filezilla/	0x32323020506C656173
24	FTP	Request: USER test	0x55534552207465737A
26	FTP	Response: 331 Password required for test	0x333331205061737377
27	FTP	Request: PASS test	0x50415353207465737A
29	FTP	Response: 230 Logged on	0x323330204C6F676765
30	FTP	Request: opts utf8 on	0x6F7074732075746638
32	FTP	Response: 200 UTF8 mode enabled	0x323030205554463820

- 区分请求类型与应答类型

- ASCII数字或字母?

请求类型

命令字符串

命令参数

- 区分请求类型的子类型:

- **关键字**: 命令字符串

应答类型

三位数字编码

详细信息

- 区分不同应答类型:

- **关键字**: 三位数字编码

- 协议消息类型识别方法

- 针对多个消息进行聚类分析
- 聚类分析依据的特征

- 消息文本特征

- 基于消息本身提取文本特征，或者直接将消息全文作为特征进行相似性匹配

- 消息的程序行为特征

- 协议实现程序在接受消息后进行处理时的指令执行行为特征

类别	形式	特征	说明
系统调用	集合	系统调用序号	处理消息过程中所有发生的系统调用
进程行为	集合	系统调用序号	进程创建、销毁等行为（CreateProcess、ShellExecute）
本地函数调用	集合	程序内部的函数调用地址	Call 指令的地址，该地址需在程序代码段中
库函数调用	集合	库函数调用地址	Call 指令的地址，该地址在程序代码段之外
执行指令地址	集合	指令EIP	所有执行指令的EIP，地址需在程序代码段中

- 消息内容的聚类依赖于序列相似性计算方法
 - Needleman-Wunsch序列相似性计算方法（全局相似性比较）
 - 构建计算矩阵，参与比较的两个序列分别置于矩阵的两个轴上
 - 矩阵左上角置0
 - 矩阵第一行和第一列赋值，表示空白匹配惩罚值
 - 在矩阵的其他位置进行计算和赋值，每个位置的值 of 下列值中最大值
 - 当前位置左侧位置的值 + 空白匹配惩罚值
 - 当前位置上侧位置的值 + 空白匹配惩罚值
 - 当前位置左上侧位置的值 + 匹配或不匹配的打分数值
 - 在矩阵的各个位置赋值后，在当前位置与赋值来源位置之间放置一个方向箭头
 - 最佳匹配数值在矩阵的右下角，同时最佳匹配路径可以沿着各个位置放置的方向箭头回溯而得到

- 消息内容的聚类依赖于序列相似性计算方法

- Needleman-Wunsch序列相似性计算示例

序列a (AAAC) 序列b (ABC)

空白匹配惩罚值 -2 匹配成功奖励值：1，匹配失败惩罚值 -1

	空白	A	A	A	C
空白	0	← -2	← -4	← -6	← -8
A	↑ -2	↖ 1	↖ -1	↖ -3	↖ -5
B	↑ -4	↑ -1	↖ 0	↖ -2	↖ -4
C	↑ -6	↖ ↑ -3	↖ ↑ -2	↖ -1	↖ -1

3、协议状态机恢复

- 消息内容的聚类依赖于序列相似性计算方法

- Needleman-Wunsch序列相似性计算示例

序列a (AAAC) 序列b (ABC)

空白匹配惩罚值 -2 匹配成功奖励值: 1, 匹配失败惩罚值 -1

	空白	A	A	A	C
空白	0	← -2	← -4	← -6	← -8
A	↑ -2	↖ 1	↖ -1	↖ -3	↖ -5
B	↑ -4	↑ -1	↖ 0	↖ -2	↖ -4
C	↑ -6	↖ -3	↖ -2	↖ -1	↖ -1

A A A C
A B C

A A A C
A B C

A A A C
A B C

- 协议状态机恢复和化简

- 基于客户端和服务端之间交换的网络消息构建有限状态机
 - 接受所有正确的消息序列，同时拒绝所有不合法的消息序列
- 确定有限状态机

$$M = (Q, \Sigma, \delta, q_0, F)$$

- Q 是状态机所有状态的集合
- Σ 是状态机输入符号的集合
- δ 是状态转移函数，定义为一个映射 $\delta: Q \times \Sigma \rightarrow Q$
- $q_0 \in Q$ 是初始状态
- $F \subseteq Q$ 是终结状态集合

- 协议状态机恢复和化简

- 技术思路：

- 从采集的协议流量中提取上述有限状态机模型的各个要素

- Σ 状态机输入符号的集合

- 从网络流量中提取到的数据包（消息）序列

- $q_0 \in Q$ 初始状态

- 接受（或发送）第一条消息之前的程序状态

- $F \subseteq Q$ 终结状态集合

- 接受（或发送）最后一条消息之后的程序状态可以视作终结状态

- δ 状态转移函数， Q ： 难以直接定义和提取

- 算法 状态机推断
- 输入：会话集合 C
- 输出：有限自动机 $S \leftarrow (Q, \Sigma, \delta, q_0, F)$
 - $q_0 \leftarrow \text{NewState}()$ $Q \leftarrow \{ q_0 \}$ $\Sigma \leftarrow \phi$ $\delta(q, s)$ 未定义
 - q 为当前状态
 - $S \leftarrow (Q, \Sigma, \delta, q_0, F)$
 - For s in C : // 每个会话
 - $q \leftarrow q_0$
 - For m in s : // 按时间顺序遍历会话中的消息
 - If $\delta(q, m)$ 已定义
 - $q \leftarrow \delta(q, m)$
 - Else:
 - $p \leftarrow \text{NewState}()$ // 创建新的状态
 - $Q \leftarrow Q \cup \{ p \}$
 - $\Sigma \leftarrow \Sigma \cup \{ m \}$
 - $\delta(q, m) \leftarrow p$
 - $q \leftarrow p$ // 当前状态设置为 p
 - $F \leftarrow F \cup \{ q \}$

3、协议状态机恢复

- FTP通信交互完整流量

- 四个完整会话，不同用户同时登录并进行文件查询、下载、上传操作

```
USER XiaoWang
PASS test
TYPE I
TYPE A
PORT 192,168,4,104,9,74
NLST
TYPE I
PORT 192,168,4,104,9,75
RETR README.txt
QUIT
```

会话A
192.168.4.104

```
USER XiaoHuang
PASS test
TYPE I
TYPE A
PORT 192,168,4,105,8,32
NLST
TYPE I
PORT 192,168,4,104,8,33
STOR README.txt
QUIT
```

会话B
192.168.4.105

```
USER XiaoWang
PASS test
TYPE I
TYPE A
PORT 192,168,4,107,4,80
NLST
TYPE I
RNFR README.log
RNTD README.txt
TYPE A
PORT 192,168,4,107,4,81
LIST
TYPE I
QUIT
```

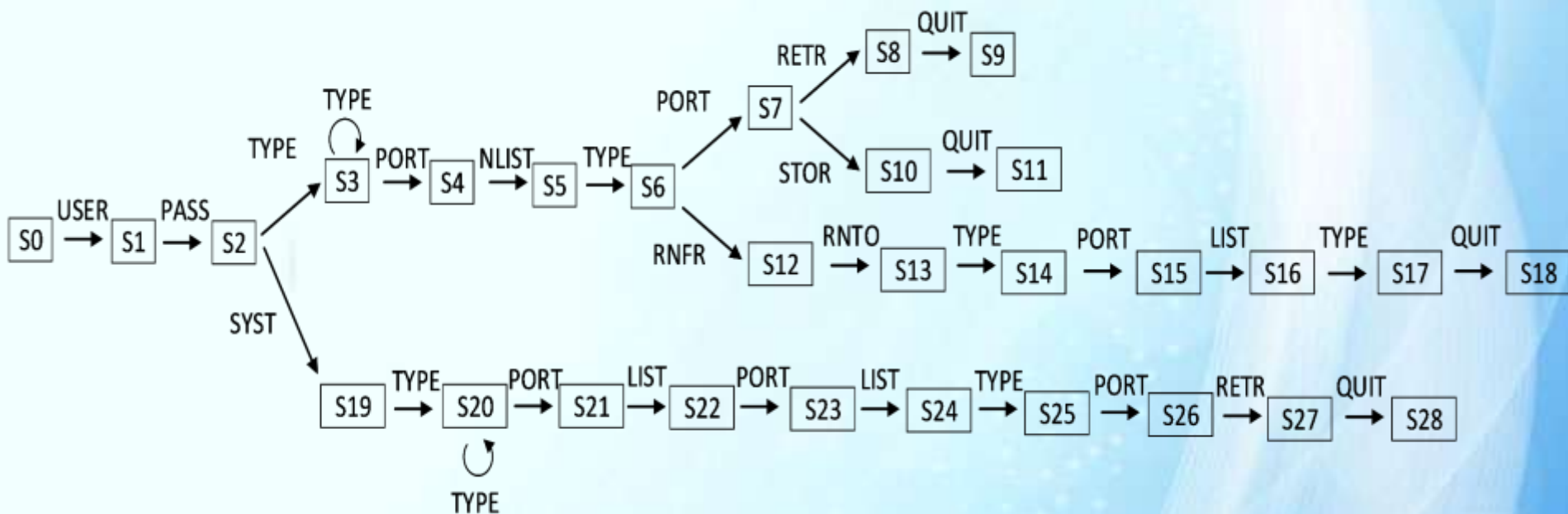
会话C
192.168.4.107

```
USER XiaoZhang
PASS test
SYST
TYPE I
TYPE A
PORT 192,168,4,111,218,22
LIST
PORT 192,168,4,111,224,225
LIST
TYPE I
PORT 192,168,4,111,130,26
RETR README.log
QUIT
```

会话D
192.168.4.105

3、协议状态机恢复

- FTP通信交互完整流量
 - 协议状态机一共包含28个状态



- 算法 状态化简1
- 输入：有限自动机 $IS \leftarrow (Q, \Sigma, \delta, q_0, F)$
- 输出：有限自动机 IS

For q in Q

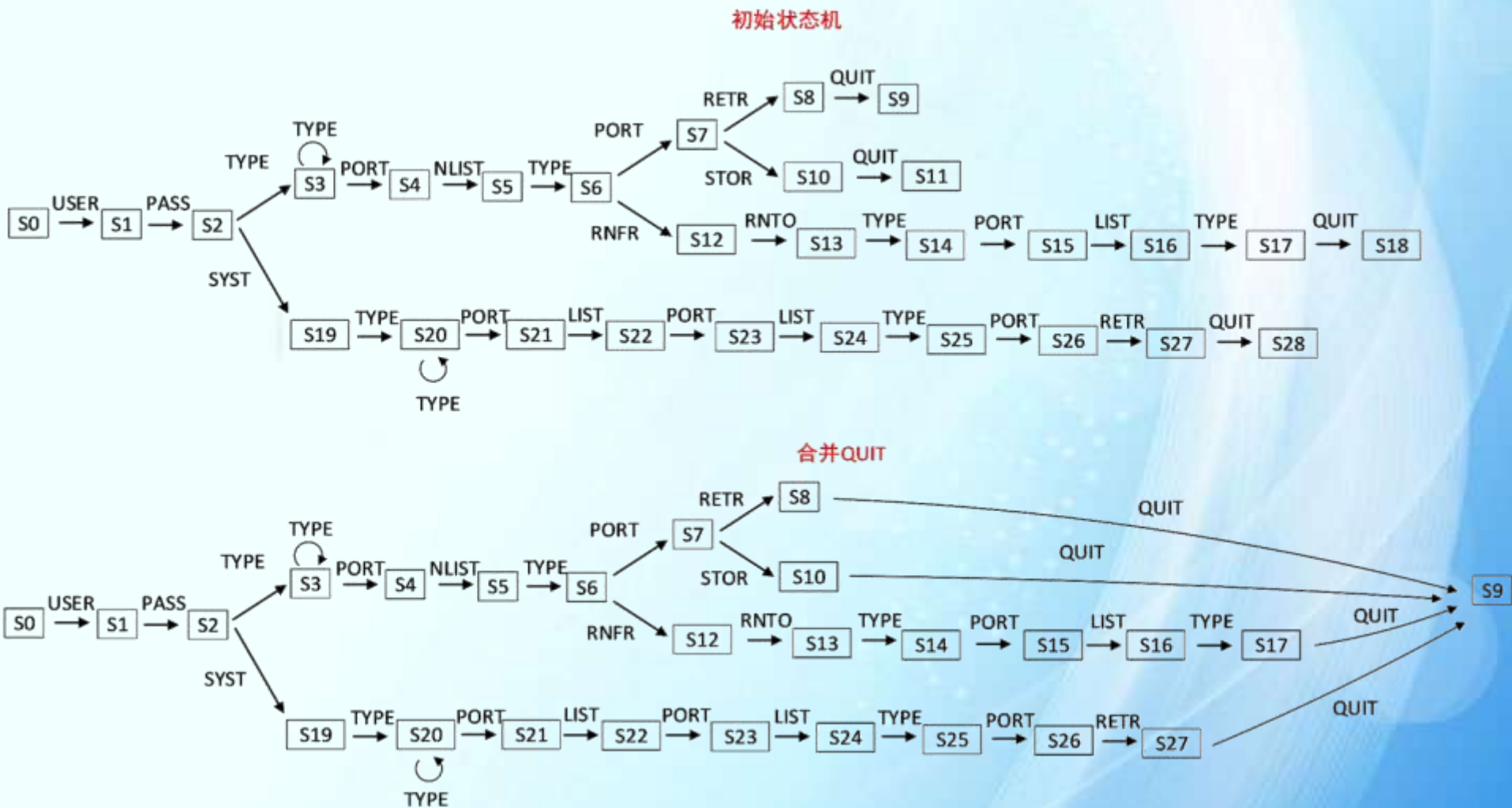
For p in Q :

If $\exists s \in \Sigma; r, t \in Q: \delta(q, s) = r \wedge \delta(p, s) = t$
MergeStates(r, t)

合并可通过相同消息类型到达的状态

3、协议状态机恢复

- FTP通信交互完整流量
 - 协议状态机一共包含28个状态

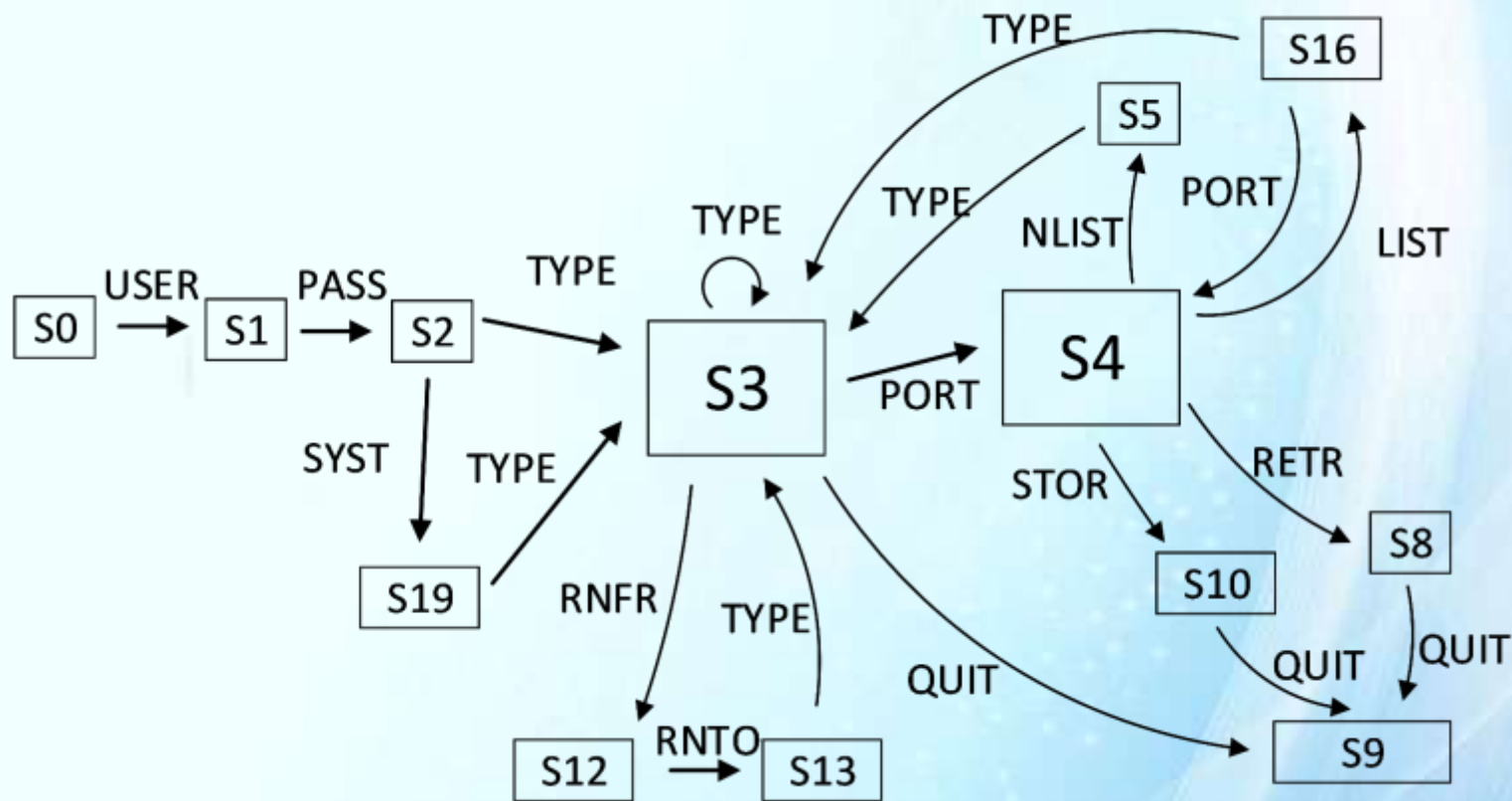


3、协议状态机恢复

- FTP通信交互完整流量

- 合同相同状态后形成的协议状态机

- 由28个状态化简为13个状态



- 算法 状态化简2
- 输入：有限自动机 $IS \leftarrow (Q, \Sigma, \delta, q_0, F)$
- 输出：有限自动机 IS

For q in Q

For p in Q

If $(\exists s \in \Sigma: \delta(q, s) = p \vee \delta(p, s) = q)$ or $(\exists s, t \in \Sigma: \delta(q, s) = p \wedge \delta(p, t) = q)$

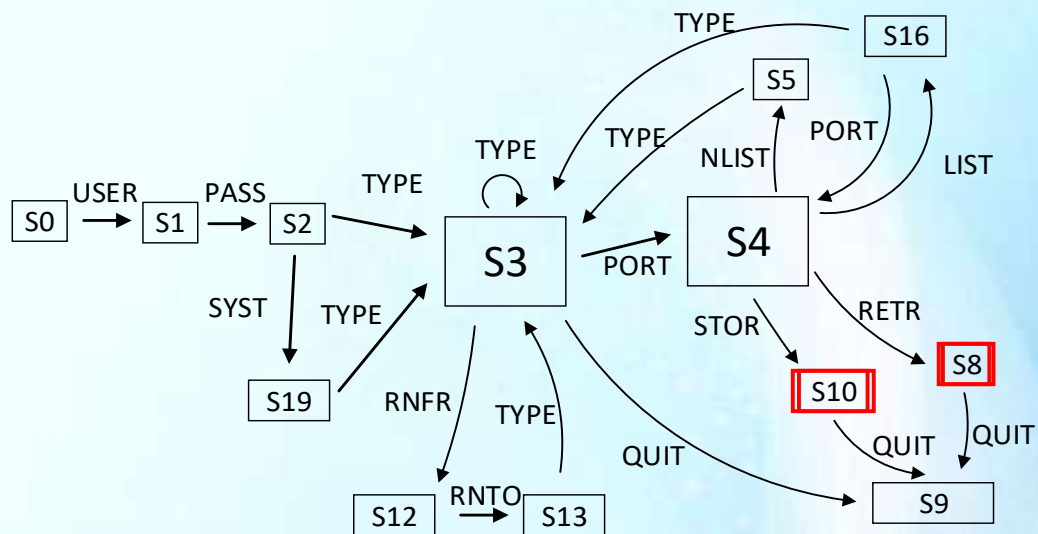
If $(\exists s \in \Sigma; r \in Q: \delta(q, s) = r \wedge \delta(p, s) = r)$

MergeStates(p, q)

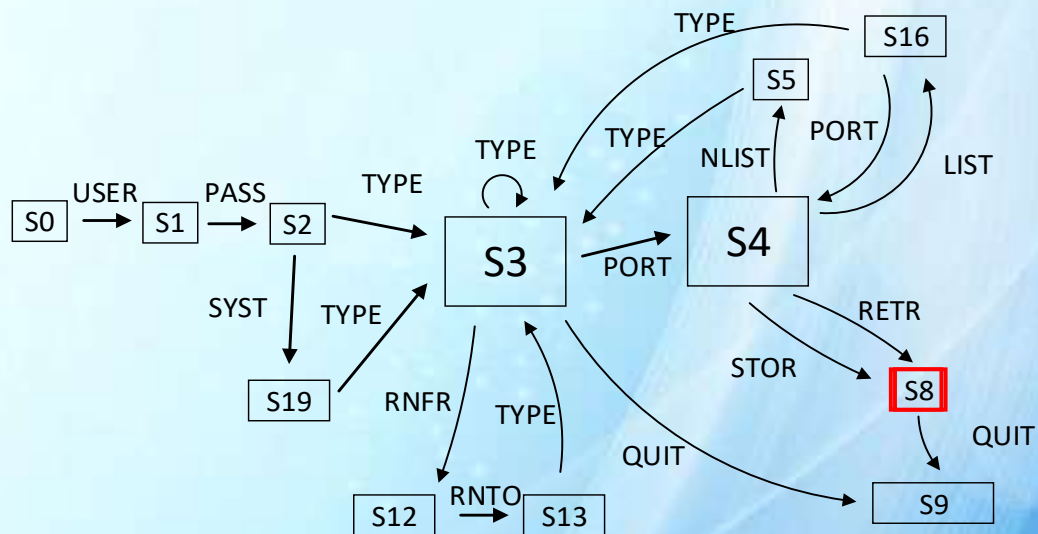
合并至少有一个后继消息相同且没有单向因果关系的状态

3、协议状态机恢复

- 合同相同状态

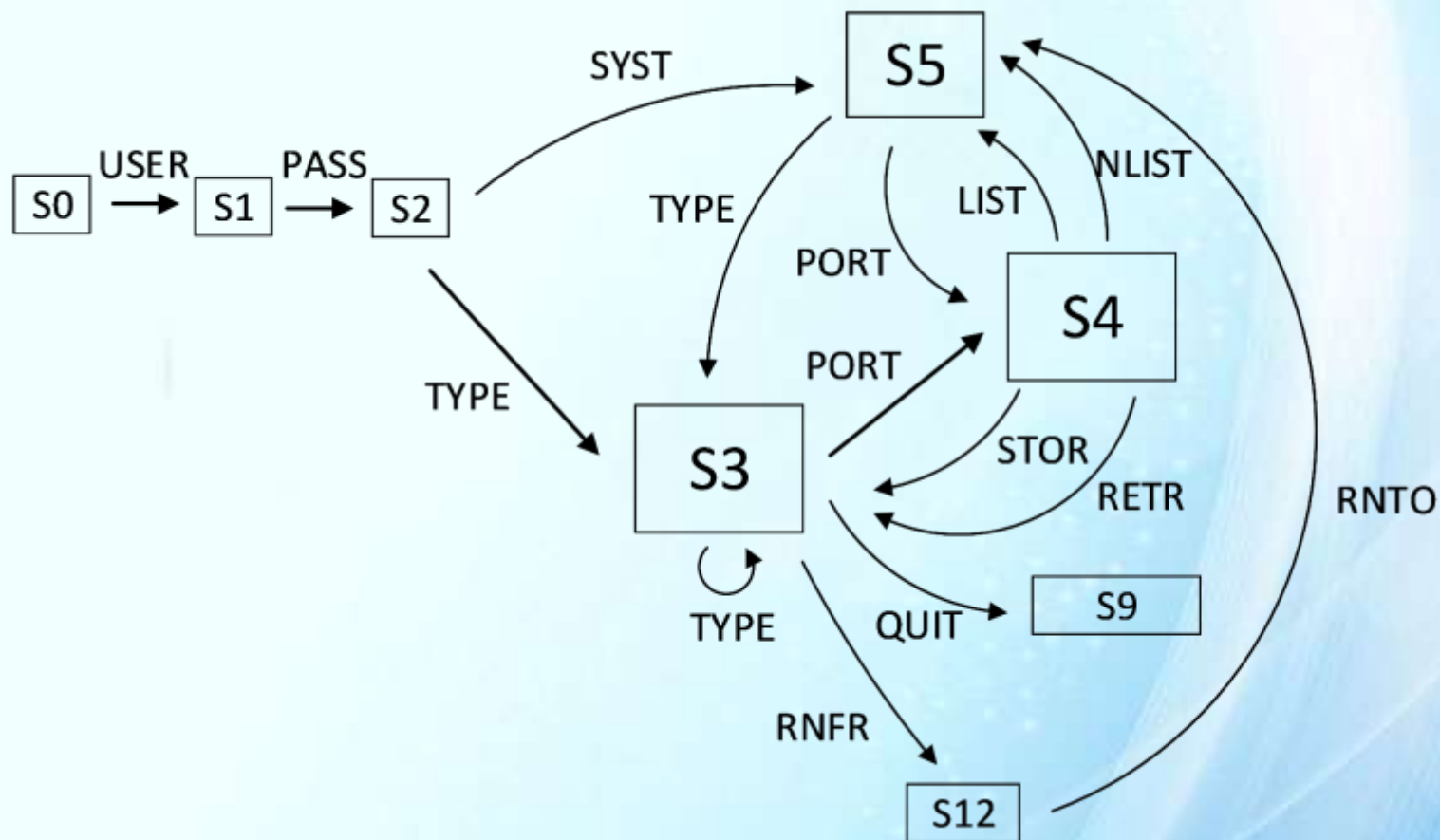


合并等价状态 S8 S10



3、协议状态机恢复

- 合同相同状态后形成的协议状态机
 - 由13个状态化简为8个状态



- 小结

- 针对有限的会话生成的状态机不够准确

- 过于具体：只能接受有限的消息序列

- 基于流量分析重构状态机的方法只有正例可用

- 没有拒绝接受的消息序列的流量（正则文法无法仅仅从正例学习得到）

- 密码运算

- 使用密码运算提高安全性的安全协议日益增多
 - 公开协议：
 - TLS
 - HTTPS, SSH
 - 私有协议：
 - Skype, 各类商业通信软件, Bitcoin
- 密码运算严重扰乱了数据流, 协议逆向难度增大
 - 污点传播、程序切片等分析方法失效

4、密码运算逆向恢复

- 解决协议逆向中密码运算的干扰

- 主要目标：密码运算识别

- 输入和输出缓冲区的位置

- 次要目标：

- 加密算法类型、密钥

- 思路：

- 密码运算的特征

- 指令特征

- » 指令模式

- » 计算指令出现频率

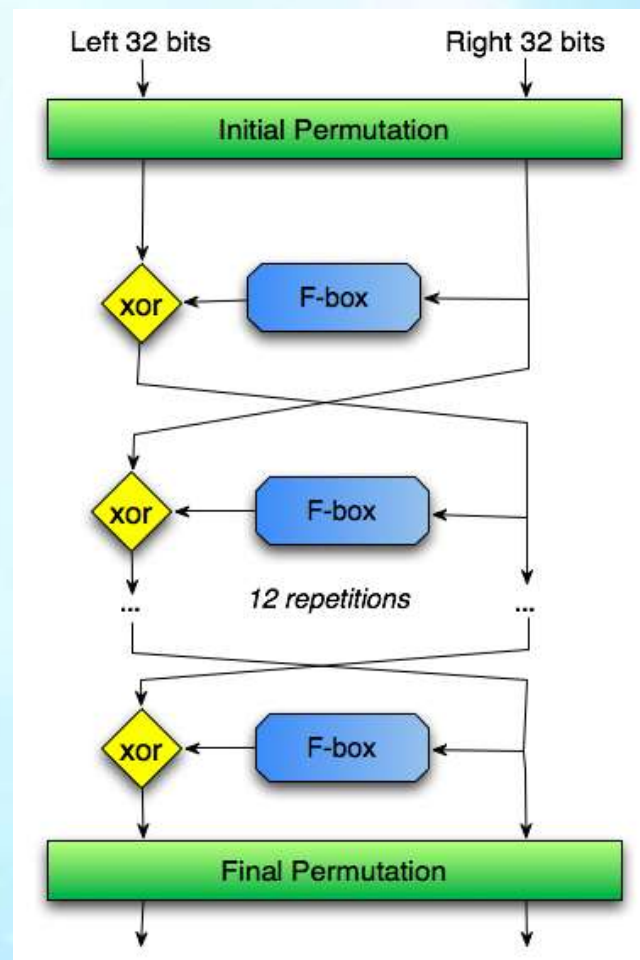
- 数据流特征：

- » 输入和输出之间的关系

- » 混乱度增大

- 结构特征

- » 循环

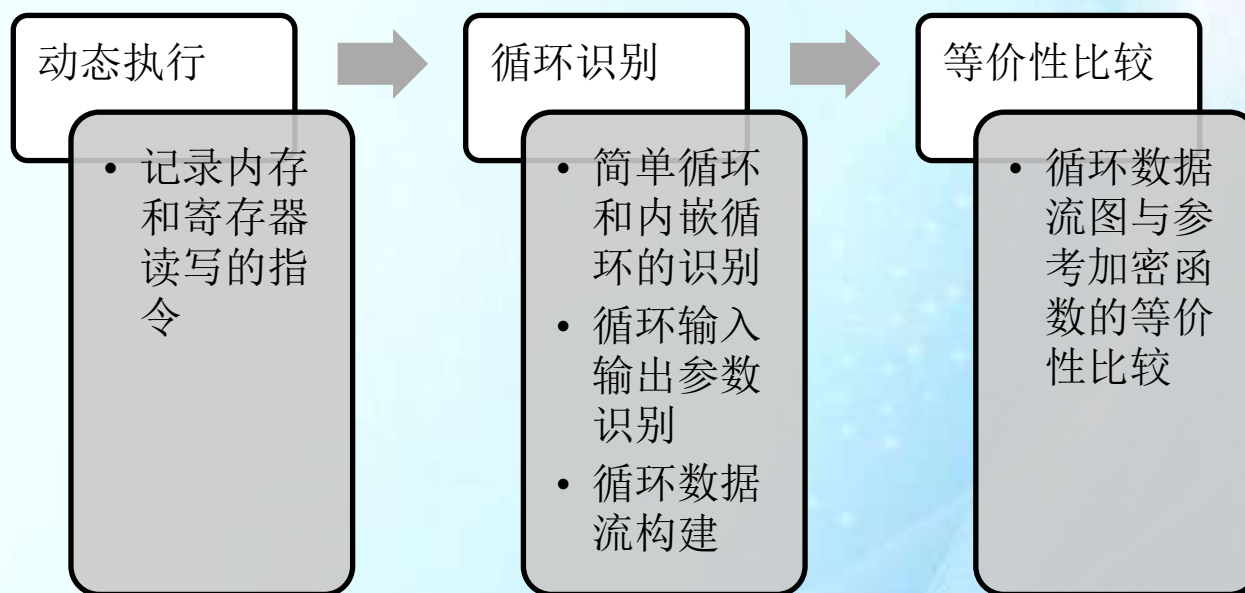


- 基于指令特征的密码算法恢复
 - 指令模式
 - 特定的一组指令形成识别密码运算的指纹
 - PEiD的KANAL插件
 - IDAPro 的Findcrpt 插件
 - 效果：
 - 判定加密算法是否存在，识别加密算法类型
 - 算术运算指令比例
 - 指令：Add, Sub, Mul, Div
 - 统计结果
 - 数据加解密运算时80%的指令为数据运算指令
 - 一般的明文处理时只有20%的指令为数据运算指令
 - 效果
 - 判定加密算法是否存在，识别加密算法类型

- 基于数据流特征的密码算法恢复
 - 输入与输出之间的混乱度变化
 - 思路：
 - 加密运算时：输入的混乱度较小，输出混乱度较大
 - 解密与加密相反
 - 效果
 - 识别输入和输出缓冲区位置
 - 输入和输出之间的雪崩效应
 - 思路：
 - 利用污点传播跟踪输入消息A
 - 若发现一个连续缓冲区B，B中每个字节均与A中每个字节有数据关系，则可认定B为输出缓冲区
 - 效果：
 - 识别输入和输出缓冲区位置

- 基于循环的密码算法恢复

- 以实现无关的方式提取加密函数实现的输入和输出参数，并将其和参照密码算法实现进行对比以实现算法的精确识别
 - 即使加密函数的实现代码被混淆，其输入参数和输出参数之间的关系仍然保持不变
 - 加密函数的实现包含指令序列的循环



- 网络协议逆向针对协议二进制实现，自动分析和获取协议规范约定，是目前未知网络协议的主要分析手段
- 网络协议逆向主要有两部分组成
 - 协议消息格式逆向
 - 字段划分
 - 字段间关系的分析
 - 字段语义恢复
 - 协议状态机恢复
 - 消息类型识别
 - 协议状态机恢复
- 网络协议逆向已成功应用于网络协议实现的安全性分析、网络流量分析、入侵检测特征生成等领域