doi:10.3969/j.issn.1000-436x.2013.10.019

语义层次的协议格式提取方法

潘璠，洪征，周振吉，吴礼发
(解放军理工大学 指挥信息系统学院，江苏 南京 210007)

摘 要：现有协议格式提取方法在语法层次对程序执行轨迹进行分析，字段识别结果可能存在冗余和冲突。为了提高字段识别准确率，提出了一种语义层次的协议格式提取方法。方法首先将执行轨迹中的二进制指令转换为语义等价的中间语言形式，并通过细粒度的动态污点分析跟踪字段语义解析过程，在此基础上，依据字段的语义不可分割性，利用语义层次的字段识别策略实现了协议格式提取。测试结果表明，该方法具有较高的识别精度和较低的分析复杂度。

关键词：协议逆向工程；协议格式提取；动态污点分析；中间语言
中图分类号：TP393.08 **文献标识码：**A **文章编号：**1000-436X(2013)10-0162-12

Protocol format extraction at semantic level

PAN Fan, HONG Zheng, ZHOU Zhen-ji, WU Li-fa
(College of Command Information System, PLA University of Science and Technology, Nanjing 210007, China)

Abstract: Present methods for protocol format extraction analyze the execution traces of programs at syntax level, which leads to redundancy and conflict in the results of fie identification. In order to improve the accuracy of field identification, a semantic level method was proposed for protocol format extraction. The method firstly translated the binary instructions into equivalent intermediate language, and tracked the parsing process of field semantics through fine-grained dynamic taint analysis. Further, it extracted otocol format using semantic level policies of field identification, based on the semantic indivisibility of fields. Experimental results show that the proposed method can achieve high identification accuracy with low complexity.

Key words: protocol reverse engineering; protocol format extraction; dynamic taint analysis; intermediate language

1 引言

目前，协议逆向工程在入侵检测、漏洞挖掘、协议重用等领域得到广泛的应用。根据分析对象的不同，现有的协议逆向方法大致分为 2 类：基于网络流量分析的方法[1,2]和基于执行轨迹分析的方法[3~7]。前者以截获的网络数据流为分析对象，依据协议字段的取值变化频率和特征推断得到协议格式。后者利用动态污点分析技术跟踪程序对报文数据的解析过程，并通过分析带污点信息的执行轨迹实现协议格式的提取。相比于网络流量分析，执行轨迹分析获取的协议知识更为精确且不受样本集限制，正逐渐成为协议逆向技术的主流。

Polyglot[3]首次提出基于执行轨迹分析的协议格式提取方法，但仅实现了对格式中分隔符、定位符和关键字的识别。Wondracek 等人[8]在 Polyglot 的基础上对多次分析的结果进行融合，得到更具通用性的协议格式描述。AutoFormat[9]在执行轨迹中加入了指令执行的上下文信息，通过上下文匹配实现了协议字段结构的提取。Dispatcher[5]结合对函数调用参数的分析，丰富了分析结果中字段的语义信息。应凌云等[7]则针对恶意软件分析的需求，在现

收稿日期：2012-05-08；**修回日期：**2013-02-20
基金项目：国家自然科学基金资助项目(61070173)；江苏省自然科学基金资助项目(BK2011115)；军用网络技术实验室创新开放基金资助项目
Foundation Items: The National Natural Science Foundation of China (6107 3); The Natural Science Foundation of Jiangsu Province (BK 2011115); The Opening Foundation of Laboratory of Military Network Technology

有方法的基础上实现了对协议字段的程序行为语义关联分析，进一步提升了协议逆向的应用价值。然而，现有方法仅对执行轨迹中处理污点数据的指令进行语法层次分析，可能导致字段识别存在冗余和冲突，进而影响了结构提取和语义推断的准确性。为此，提出了一种新的语义层次的协议格式提取方法。该方法在执行轨迹的重放过程中，首先将二进制指令转化为精简的中间语言形式；然后，针对具有单一语义的中间指令，通过细粒度的动态污点分析对字段语义解析过程进行跟踪；最后，依据字段的语义解析特征，在语义层次实现协议格式的提取。

本文的主要贡献如下：1) 首次将基于中间语言的动态污点分析思想应用于协议逆向分析领域。将语义复杂、种类繁多的二进制指令转化为精简、单一语义的中间语言形式，在降低指令语义分析复杂度的同时，提高了报文数据跟踪的准确性。2) 提出一种语义层次的字段识别策略。依据字段的语义不可分割性，在语义层次实现字段识别，避免了结果中的字段冗余和冲突，进而提高了协议格式提取的精确度。3) 实现了语义层次的协议格式提取工具原型系统(SPAE, semantic-level protocol format extractor)。利用原型系统对不同类型的多种协议进行了测试，并与现有方法进行了对比。测试结果验证了本文方法的有效性。

2　问题分析

协议格式由若干个字段组成，字段为具有特定语义的最小不可分割的连续字节序列，字段之间存在顺序、并列和层次关系。根据这一定义，协议格式提取可分为字段识别、结构提取和字段语义推断 3 个阶段[5]。字段识别的正确与否直接影响到结构提取和语义推断的准确性，是协议格式提取的关键。

传统方法利用细粒度的动态污点分析对每个污点报文字节进行标定和追踪，然后扫描执行轨迹中指令操作数的污点记录，根据污点数据块访问的粒度实现字段边界的划分。然而由于内存拷贝、校验和计算以及编译器优化等原因，这种在指令语法层次实现的字段识别方式存在粒度过细、冗余和冲突的缺陷[4]。表 1 给出了传统方法通过分析执行轨迹实现字段识别的简单示例。其中，带污点记录的执行轨迹片段取自网络程序 Emule0.48a 的报文解析过程。

表 1　传统方法的字段识别过程

序号	汇编指令	污点记录	识别结果
1	mov esi, [esp+019Ch]	esi a {0,1,2,3}	—
2	and esi, 0FFh	esi a {0,1,2,3}	<0,3>
3	lea eax, [esi-1]	esi, eax a {0,1,2,3}	<0,3>
4	cmp eax, 0E2h	esi, eax a {0,1,2,3}	<0,3>
5	jz loc_527169	esi, eax a {0,1,2,3}	—

在表 1 中，第 1 行指令将报文首部的 4 byte 数据读入寄存器 esi。由于第 2~4 行指令均操作了污点数据块<0,3>，传统方法对字段的识别明显存在冗余。另外，读入的 4 byte 污点数据经过和 0xFF 的与运算，参与比较并影响跳转的仅为单字节数据，实际的字段边界为<0,0>。由于污点数据块<1,3>还将被后续指令访问，这一误差将导致字段识别的冲突。

为了提高协议格式提取的精度，AutoFormat[9] 根据字段的边界范围构建报文结构树，通过删除重复节点去除冗余，但其在消除边界冲突时将使得字段划分过细。Tupni[4]以指令操作数访问的污点数据块作为候选字段，并将访问的次数作为该候选字段的权值，将边界划分问题转化为具有最大权值的不相交字段集合的计算。考虑到集合覆盖本质为 NP 完全问题，Tupni 仅采用贪心算法获取局部最优解，并不能保证结果的正确性。

仅在语法层次对二进制指令进行分析，是现有方法难以保证字段识别正确率的根本原因。对于输入数据而言，其解析过程分为语法解析和语义解析 2 个阶段[10]。在语法解析阶段，协议实体将字段对应的连续字节序列读取到程序变量中，并进行格式化和校验处理；在语义解析阶段，协议实体将对程序变量中字段的取值做出解释。但在二进制程序中，指令处理的对象为内存单元和寄存器，没有程序变量的概念。无论字段是数值、指针还是字符流，都被视为位宽有限的数字进行处理，其粒度与指令访问污点数据的粒度之间并不存在严格的对应关系。因此，现有语法层次的指令分析方式难免在识别字段的过程中存在误差。

字段作为协议格式的基本单位，在语义上具有最小不可分割性。因而在语义解析阶段，与指令相关的连续输入数据必然属于同一个字段，并且该字段仅包含这些数据。即使在经过编译器编译和优化的二进制指令级，这种特性将依然存在，否则协议实体无法与对应的协议规范相匹配。因此，利用字

段的语义解析特征划分字段边界，将有效克服语法层次方法存在的缺陷，得到更为精准的结果。

基于上述分析，本文考虑在语义层次实现字段识别。根据功能的不同，字段语义可以分为控制语义和操作语义 2 类[5]。控制语义指定报文自身的语法解析方式，如长度、分隔符、关键字等；操作语义则决定协议实体如何完成响应操作，如网络地址、端口、文件路径等。本文发现，二进制程序对控制语义的解析通过跳转条件与输入相关的控制流分支实现，而对操作语义的解析则通过参数与输入相关的函数调用来完成。因此，本文方法的基本思路是，通过动态污点分析技术跟踪报文解析过程，重点对分支指令的跳转条件和函数调用参数进行分析，进而实现语义层次的协议格式提取。

3 语义层次的协议格式提取方法

3.1 方法概述

方法以二进制程序和协议报文为输入，记录报文解析过程的完整执行轨迹，然后对执行轨迹进行重放，利用基于中间语言的细粒度动态污点分析技术，在语义层次实现协议格式的提取。对于执行轨迹中的每条二进制指令，重放过程分为 3 个阶段。

1) 中间语言转换。将二进制指令转换为精简的、语义等价的中间语言形式，避免直接对种类繁多、语义复杂的二进制指令进行处理。

2) 动态污点分析。依据中间语言的指令语义更新污点上下文 C，实现基于中间语言的细粒度动态污点分析，为语义层次的协议格式提取提供支撑。

3) 协议格式提取。分析字段语义解析相关的中间指令，根据语义层次的字段识别策略将指令操作数相关的污点标签合并为字段，并利用已有方法提取报文结构，最终得到 BNF 形式的协议格式描述 G_M。

方法流程更为精确的描述如算法 1 所示。后续将详细介绍上述阶段的具体处理策略。

算法 1 语义层次协议格式提取

输入：二进制程序 P，报文输入 M

输出：BNF 形式的协议格式描述 G_M

1) *ExtractFormat* (*Trace*)
2) let F, $G_M:=\varnothing$; /*F 为已识别字段集*/
3) *trace*:= *LogTrace*(*P*, *M*); /*执行轨迹的记录*/
4) C :=*InitTaintContext*(); /*污点上下文的初始化*/
5) for each *i* in *Trace* do
6) *ilSeq*:=*ILtranslate*(*i*); /*中间语言转换*/
7) for each *stmt* in *ilSeq* do
/*细粒度动态污点分析*/
8) C:=*DynamicTaintAnalysis*(*stmt*, C);
9) f:=*FieldIdentify*(*i*, C, F); /*语义层次字段识别*/
10) $F:=\{f\}\cup F$
/*报文结构提取*/
11) G_M:=*StructureExtract*(*stmt*, f , G_M);
12) end for
13) end for
14) return G_M

3.2 中间语言转换

在二进制级进行语义层次的字段识别主要面临 3 个方面的挑战。首先，指令语义的复杂度高，例如寄存器别名、指令前缀等问题必须考虑。如果对报文数据传播的跟踪不够精确，将无法避免字段语法解析阶段带来的误差。其次，指令集规模极其庞大，将导致指令语义分析策略过于繁冗。在 x86 指令集中，仅条件跳转指令的种类就超过 30 种，在保证完备性的前提下分析策略的复杂度将不可接受。最后，底层指令缺乏对函数调用的抽象，无法通过函数调用参数实现对字段语义的推断。

针对上述挑战，采用将二进制级指令转换为中间语言指令的方式进行分析。David Brumley 等[11]提出了面向二进制程序分析的中间语言 BIL，如图 1 所示。为了简化描述，省略了图 1 中 load 和 store 指令的部分参数，默认采用小尾的内存读写方式。

program	::=	*stmt* *
stmt	::=	*var* := *exp* | goto *exp* | if *exp* then goto *exp* else goto *exp* | store(*exps*,*exp*,t_{reg}) | label *label_kind* | special string | call *exp* with *argument* ret *var* | return
exp	::=	load(*exp*,t_{reg}) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | *integer* | cast(*cast _ kind*,t_{reg},*exp*)
label_kind	::=	*integer* | string
cast_kind	::=	high | low | unsigned | signed
var	::=	(string, id_v,t_{reg})
$\Diamond_b$	::=	+, −, *, /, $/_s$, mod, mod_s, ==, != <, ≤, $<_s$, $\leq_s$,&,|,⊕, =, ?, $?_a$
$\Diamond_u$	::=	−(unary minus), ∼(bit-wise not)
argument	::=	$(var)^+$
t_{reg}	::=	reg1_t | reg8_t | reg16_t | reg32_t | reg64_t

图 1 扩展的 BIL 语法

协议格式提取方法基于 BIL 实施，首先利用二进制代码分析平台 BAP[11]将执行轨迹中的汇编指令转换为中间语言，主要分为 2 个步骤：1) 将二进制指令转化为精简的 VEX IL[12]形式；2) 使用 BIL 指令显式描述 VEX 指令的副效应(side-effect)。

相比于原始汇编指令，BIL 的指令规模明显缩减，并且每条指令都具有单一且明确的语义，在指令语义描述方面具有明显的优势。为了提升跨平台的代码分析能力，BAP 在指令转换时将所有平台相关的特殊指令简化为无执行语义的 special string 指令，对其不做处理。由于执行轨迹中的跳转路径与指令参数取值均已确定，并且特殊指令通常与报文数据的解析无关，BAP 的简化不会造成后续动态污点分析的误差。

需要说明的是，中间语言转换得到的 BIL 指令序列并不唯一，例如 BAP 可以进一步将 BIL 指令序列转换为静态单赋值(SSA, single static assignment)形式。虽然转换后的 BIL 指令可以在语法特征上存在差异，但 BAP 能够保证除平台相关的特殊指令外，得到 BIL 形式的指令在语义上与原始指令等价。因此，方法能够在降低指令分析复杂度的同时，确保语义层次字段识别准确度。

3.3　细粒度动态污点分析

为了分析执行轨迹重放过程中的报文解析语义，需要为每个输入数据字节赋予唯一的污点标签，进而跟踪每个污点字节的传播过程。为此，提出了基于 BIL 的细粒度动态污点分析策略，通过维护污点上下文来保存输入字节的传播状态，并根据指令的语义对污点上下文进行更新，实现细粒度的动态污点分析。首先给出污点上下文的定义。

定义 1　污点上下文 C 可定义为四元组 $<\mu, ?, T_\mu, T_?>$，其中，

1) m 为内存地址到取值的映射，$m[o]$ 表示地址为 o 的内存字节取值；

2) $?$ 为变量到取值的映射，$?[var]$ 表示变量 var 的取值；

3) T_μ 为内存地址到污点属性的映射，$T_\mu[o]$ 表示地址为 o 内存字节的污点属性；

4) $T_?$ 为变量及其字节偏移到污点属性的映射，$T_?[var, n]$ 表示变量 var 的第 n byte 的污点属性。

由于程序内部复杂的数据依赖关系，内存和寄存器变量通常与多个污点数据字节相关。因此在动态污点分析过程中，将污点属性定义为污点标签的集合。为了精确描述污点分析策略，采用式(1)所示推理规则对指令执行语义进行定义。

$$\frac{\text{computation}}{<\text{current state}>, stmt \ \hat{\ } \ <\text{end state}>} \tag{1}$$

其中，$stmt$ 为当前执行的指令，运算符 ^ 表示指令执行操作，computation 为完成污点属性更新所需的计算条件。在 computation 中，$\leftarrow$ 表示更新操作符，$\hat{}$ 表示连续操作符。

对于指令中的表达式赋值操作，采用类似的描述方式。$\mu, ?, T_\mu, T_? \bullet exp \Downarrow \langle v, w, \mathrm{T} \rangle$ 表示污点上下文为 $<\mu, ?, T_\mu, T_?>$ 时，表达式 exp 的取值为 v，位宽为 w，污点取值 $\mathrm{T} = t_1 t_2 \cdots t_w$ 为 exp 中字节对应的污点属性序列。为了方便描述，以 $\mathrm{T}[n]$ 表示第 n byte 的污点属性。

依据上述约定，基于 BIL 的细粒度动态污点分析策略如图 2 所示。动态污点分析策略由 3 条指令语义规则和 9 条表达式赋值规则组成，实现对污点属性的标记和传播。污点标记由函数调用指令的语义规则实现。以 recv 函数为例，规则 T-RECV 依据函数返回的实际报文长度创建污点标签，并对函数第 2 个参数指向的内存区域进行污点属性的初始化。污点传播是对污点属性映射 T_μ 和 $T_?$ 的更新，分别由指令语义规则 T-ASSIGN、T-RSTORE 完成。

在表达式赋值过程中，二元运算符&、|、⊕的结果与参数之间存在位对应关系。为了提高精度，污点分析策略采用 T-BBINOP 规则单独对这类运算符进行处理，得到位精确的表达式污点取值。而对于其余的二元运算符(+，*，mod 等)，则认为运算结果与所有参数中包含的污点标签相关，通过 T-ABINOP 规则进行表达式污点赋值。由于二进制程序中清零操作(xor EAX, EAX)和截取操作(and ECX, 0FFh)通过位运算指令实现，T-BBINOP 规则还根据指令的参数类型和取值对表达式的污点取值进行修正，以保证污点属性映射与指令语义之间的等价性。

动态污点分析策略仅对赋值、内存写入和函数调用 3 类指令进行处理，不考虑其他指令对动态污点上下文的影响。一方面，goto exp、return 等跳转指令只决定执行流程，无法更新取值和污点属性映射；另一方面，special string 表示的如浮点计算等复杂指令或者特权指令，通常情况下与报文解析过程无关。因此，对指令语义规则的简化是可以接受

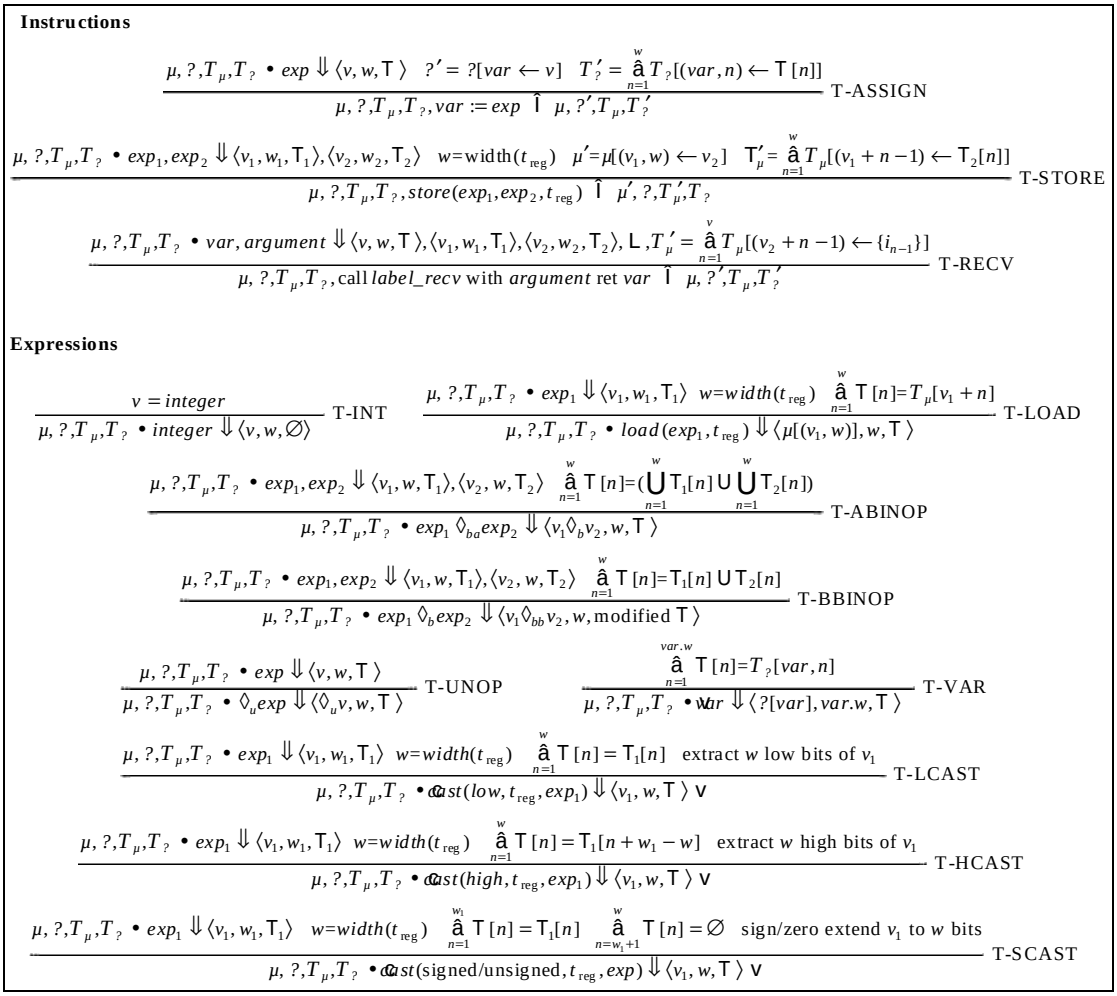


图 2 基于 BIL 的细粒度动态污点分析策略

的，并且可以提高规则匹配的效率。

由图 2 可以看出，基于精简的、具有单一语义的 BIL 指令集，方法采用少量规则即可实现细粒度的动态污点分析，显著降低了指令分析的复杂度。与已有基于中间语言的动态污点分析[13]不同，上述策略还考虑了指令参数位宽的差异，与二进制指令的语义更相吻合。

由于污点属性为污点标签的集合，字节粒度的污点属性跟踪将带来巨大的内存开销，同时污点属性的复制和合并操作也会导致严重的性能损失。王铁磊等[14]依据污点数据传播的重叠性和连续性特征，引入 roBDD 表示集合形式的污点属性，在减少内存消耗的同时，显著提高了分析效率。方法在具体实现时沿用了这一思想，以确保动态污点分析策略的可行性。

3.4 协议格式提取

在动态污点分析的基础上，提出一种语义层次的字段识别策略。基本思想是：借助于污点上下文，分析分支跳转指令和函数调用指令的参数，将污点属性中连续污点标签识别为字段，并加入到字段集合 F 中。图 3 给出了策略规则的形式化描述。

以 F-IF 规则为例，对于 if exp_1 then goto exp_2 else goto exp_3 指令，假定跳转条件 exp_1 的污点取值为 T，计算 T 所包含污点标签的集合 $\bigcup_{n=1}^{w} T[n]$。若有 $\{i_p,\cdots,i_q\}=\bigcup_{n=1}^{w} T[n]$，可将 $[p,q]$ 识别为字段长度，并更新 $F=F\cup\{f<p,q>\}$。由于字段具有语义不可分割性，语义层次的字段识别将不会造成字段划分粒度过细和冲突的情况，但无法避免由语义重复解析导致的结果冗余。例如 if 指令作为循环的退出条件时，将多次触发 F-IF 规则对同一字段进行解析。因此，F-IF 规则还将判断识别的字段是否已经存在于集合 F 中，以避免字段识别的重复。

$$\frac{\mu,?,T_\mu,T_? \bullet exp_1 \Downarrow \langle v,w,\mathsf{T}\rangle \quad \{i_p,\mathsf{L},i_q\}=\bigcup_{n=1}^{w}\mathsf{T}[n] \quad \acute{o}f_0(f_0\in F\wedge f_0\subseteq\{i_p,\mathsf{L},i_q\})}{\mu,?,T_\mu,T_?,F,\text{if } exp_1 \text{ then goto } exp_2 \text{ else goto } exp_3 \;\hat{\mathsf{I}}\; \mu,?,T_\mu,T_?,F\cup\{f<p,q>\}}\ \text{F-IF}$$

$$\frac{\mu,?,T_\mu,T_? \bullet exp \Downarrow \langle v,w,\mathsf{T}\rangle \quad \{i_p,\mathsf{L},i_q\}=\bigcup_{n=1}^{w}\mathsf{T}[n] \quad \acute{o}f_0(f_0\in F\wedge f_0\subseteq\{i_p,\mathsf{L},i_q\})}{\mu,?,T_\mu,T_?,F,\text{goto } exp \;\hat{\mathsf{I}}\; \mu,?,T_\mu,T_?,F\cup\{f<p,q>\}}\ \text{F-GOTO}$$

$$\frac{\mu,?,T_\mu,T_? \bullet argument \Downarrow \langle v_1,w_1,\mathsf{T}_1\rangle,\langle v_2,w_2,\mathsf{T}_2\rangle,\mathsf{L} \quad \text{for each } j,\ \{i_{p_j},\mathsf{L},i_{q_j}\}=\bigcup_{n=1}^{w_j}\mathsf{T}_j[n] \quad \acute{o}f_0(f_0\in F\wedge f_0\subseteq\{i_{p_j},\mathsf{L},i_{q_j}\})}{\mu,?,T_\mu,T_?,F,\text{call } exp \text{ with } argument \text{ ret } var \;\hat{\mathsf{I}}\; \mu,?,T_\mu,T_?,F\cup\bigcup\{f_j<p_j,q_j>\}}\ \text{F-CALL}$$

图 3　语义层次字段识别规则

考虑到 goto *exp* 在跳转地址输入可控的条件下也作为分支跳转指令，F-GOTO 规则采用与 F-IF 规则相同的策略对跳转地址的污点取值进行分析。而对于函数调用指令，F-CALL 规则函数的输入参数依次采用类似的策略。

为了描述字段识别过程，以图 4 所示的代码片段为例。图 4(a)与表 1 对应，图 4(b)为转换后的 BIL 代码。

```
1) mov esi,
   [esp+019Ch]
2) and esi, 0FFh
3) lea eax, [esi-1]
4) cmp eax, 0E2h;
   jumptable
5) jz  loc_527169
```

(a)与表1对应的代码

```
1) R_ESI := load(R_ESP+
          0x19C, reg32_t)
2) R_ESI := R_ESI & 0xFF
3) R_EAX := R_ESI - 1
4) T_32t_1 := R_EAX - 0xE2
5) R_ZF := (T_32t_1 == 0x0) L
6) if  R_ZF == 0x1 then goto loc_527169
```

(b)转换后的BIL代码

图 4　BIL 形式的代码示例

依据动态污点分析策略，图 4(b)中代码片段的重放过程如表 2 所示。由于映射m、$?$、T_μ与字段识别无关，表中仅保留对映射$T_?$更新的描述。1)~5)行均为赋值指令，因此应用规则 T-ASSIGN 对映射$T_?$进行更新。而第 6 行为分支跳转指令，触发 F-IF 规则对字段进行识别。计算跳转条件 R_ZF==0x1 对应的污点取值 T 为$[\{i_0\},\{i_0\},\{i_0\},\{i_0\}]$，显然有$\{i_0\}=\bigcup_{n=1}^{4}T[n]$，因而识别的字段为<0,0>。对比表 1 和表 2 可以看出，语义层次的字段识别可以有效地避免字段语法解析过程带来的冗余，识别字段边界也更为精确。

由于协议实体功能的差异，并不是所有协议字段都会被程序解析，因而识别的字段结果往往对报文数据的覆盖并不完全。在执行轨迹重放完毕后，将所有已识别字段之间的连续字节区域视为一个完整字段加入到字段集合 F 中。考虑到存在连续多个未解析字段的可能性，这种处理方式可能导致字段识别的粒度较粗，但能够保证报文结构的完整性。

在字段识别的基础上，方法还结合已有的结构提取技术实现完整的协议格式逆向分析。Lin 等[15]指出并通过实验证明，当结构化数据按照自顶向下方式解析时，数据结构可通过分析指令之间的动态控制依赖关系得到，克服了原有启发式结构提取策略的缺陷。但 Cui 等[4]发现，当内存拷贝、校验和计算等操作导致字段划分过细甚至冲突时，Lin 的方法将失效。本文方法将语义层次字段识别与 Lin 的方法结合，有效弥补了这一缺陷。

4　实验评估

为了评估方法的有效性，本文在 Linux 平台上实现了原型系统 SPAE，并使用 SPAE 对若干协议报文进行了格式提取，同时将提取的结果与现有方

表 2　语义层次的字段识别过程

序号	$T_?$ 更新记录	执行规则	字段识别结果
1	R_ESI←$[\{i_0\},\{i_1\},\{i_2\},\{i_3\}]$	T-VAR, T-INT, T-ABINOP, T-LOAD, T-ASSIGN	—
2	R_ESI←$[\{i_0\},\varnothing,\varnothing,\varnothing]$	T-VAR, T-INT, T-BBINOP, T-ASSIGN	—
3	R_EAX←$[\{i_0\},\{i_0\},\{i_0\},\{i_0\}]$	T-VAR, T-INT, T-ABINOP, T-ASSIGN	—
4	T_32t_1←$[\{i_0\},\{i_0\},\{i_0\},\{i_0\}]$	T-VAR, T-INT, T-ABINOP, T-ASSIGN	—
5	R_ZF ←$[\{i_0\},\{i_0\},\{i_0\},\{i_0\}]$	T-VAR, T-INT, T-ABINOP, T-ASSIGN	—
6	—	T-VAR, T-INT, T-ABINOP, F-IF	<0,0>

法结果进行了对比分析。

4.1 原型系统架构

SPAE 由静态分析模块、执行轨迹生成模块、中间语言翻译模块、污点分析引擎以及格式提取模块 5 部分组成，如图 5 所示。

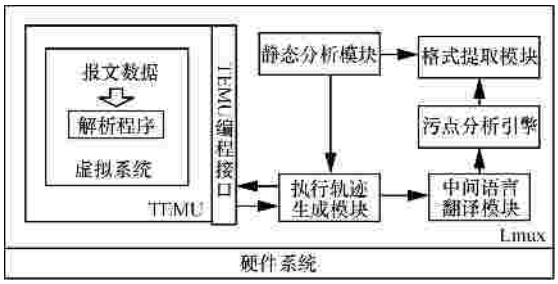


图 5 SPAE 设计架构

静态分析模块利用 IDA Pro 反汇编二进制程序，将识别的静态库函数信息提供给执行轨迹生成模块，并计算分支指令的直接必经节点信息，为报文结构提取时动态依赖关系的计算提供依据。

考虑安全性与跨平台需求，执行轨迹生成模块以插件形式与系统级动态分析平台 TEMU[16]进行交互，负责记录报文解析过程中指令静态信息、指令运行时信息以及函数调用的参数和返回值等。

中间语言翻译模块扩展了 BAP 平台[11]的组件 toil 和 iltrans，负责将执行轨迹中的指令转换为带函数抽象的 BIL 形式。

作为 SPAE 的核心，污点分析引擎负责维护重放过程中的污点上下文。此外，污点分析执行引擎还集成了 roBDD 开源库 BuDDY[17]，用于实现污点属性的压缩存储。

格式提取模块根据污点上下文分析指令语义，实现字段识别和结构提取，并输出分析结果。

4.2 实验及结果分析

本文利用 SPAE 对 Windows 环境下多种协议解析程序进行了测试。具体分析环境配置如下：运行 SPAE 的分析主机配备了 Intel Core Duo 2.93 GHz 的 E6500 CPU，4 GB 内存和 32 bit 的 Ubuntu9.04 系统，同时在 TEMU 中加载 32 bit 的 Windows XP 系统作为协议解析程序的执行环境。

根据协议类型，选择了 3 种二进制协议(DNS、eDonkey、DHCP)、3 种文本型协议(FTP、HTTP、SIP)以及混合型的 McAfee ePO 框架服务协议作为测试对象。报文样本及执行轨迹信息如表 3 所示。

SPAE 对记录的执行轨迹进行重放，并输出协议格式提取结果。将结果中识别的字段与公开的协议规范进行对比，以评估字段识别的准确率。同时作为对照，借助于 TEMU 自带的 tracecap 插件记录了带污点信息的执行轨迹，并分别采用 AutoFormat 的字段树构造方法和 Tupni 的加权集合覆盖计算方法完成字段识别。

表 4 给出了报文实际包含的字段数、执行轨迹中与污点相关的记录数以及 3 种方法的字段识别结果对比。$|F|$、$|F_c|$、$|F_o|$分别为正确识别、识别粒度过粗和识别粒度过细的字段数。可以看出，3 种方法识别的字段数与报文实际的字段数相当，均有效避免了大规模污点访问记录可能导致的字段重复识别。相比而言，AutoFormat 与 Tupni 的识别结果中同时存在粒度过粗和过细的字段，而 SPAE 的结果中则仅存在少量粒度划分过粗的字段。进一步统计得到 AutoFormat、Tupni 和 SPAE 的平均字段识别准确率分别为 71.1%、79.4%和 93.9%，这说明 SPAE 具有更精准的字段识别能力。在此，对字段识别误差产生的原因进行详细分析。

对于 Emule 客户端之间发送的 Hello 报文，SPAE 能够正确识别所有的字段，而 AutoFormat 和 Tupni 均存在字段识别的误差，如图 6 所示。对于单字节的 Protocol Type 字段，目标程序采用表 1 所

表 3 SPAE 测试样本信息

报文类型	解析程序	发送程序	报文长度/byte	执行记录规模/Mbyte
DNS Query	Bind 9.9.0	nslookup(Win)	28	3.7
eDonkey Hello	Emule 0.48a	Emule 0.48a	86	315.4
DHCP BOOTP Request	OpenDHCPServer 1.48	DHCP Client(Win)	176	20.4
FTP USER Request	FileZilla Server 0.9.41	FileZilla 3.5.3	16	2.6
HTTP GET Request	Apache 2.4.2	IE 7.0	318	16.2
SIP Register	MiniSIPServer Express	Nero IP phone 2.0	467	247.8
ePO POST PolicyCheck	McAfee ePO 4.5	McAfee Agent 3.0	940	95.6

表 4　字段识别的结果统计

报文类型	字段数	污点相关的指令数	AutoFormat			Tupni			SPAE		
			$\|F\|$	$\|F_c\|$	$\|F_o\|$	$\|F\|$	$\|F_c\|$	$\|F_o\|$	$\|F\|$	$\|F_c\|$	$\|F_o\|$
DNS Query	13	4 372	9	1	2	9	1	2	10	1	0
eDonkey Hello	23	17 983	21	0	4	21	1	1	23	0	0
DHCP BOOTP Request	39	25 386	28	5	0	28	5	0	28	5	0
FTP USER Request	4	2 469	4	0	0	3	0	0	4	0	0
HTTP GET Request	67	32 996	60	0	21	60	0	21	67	0	0
SIP Register	104	83 152	103	0	3	103	0	3	104	0	0
ePO POST PolicyCheck	65	120 903	0	0	940	27	0	832	60	1	0

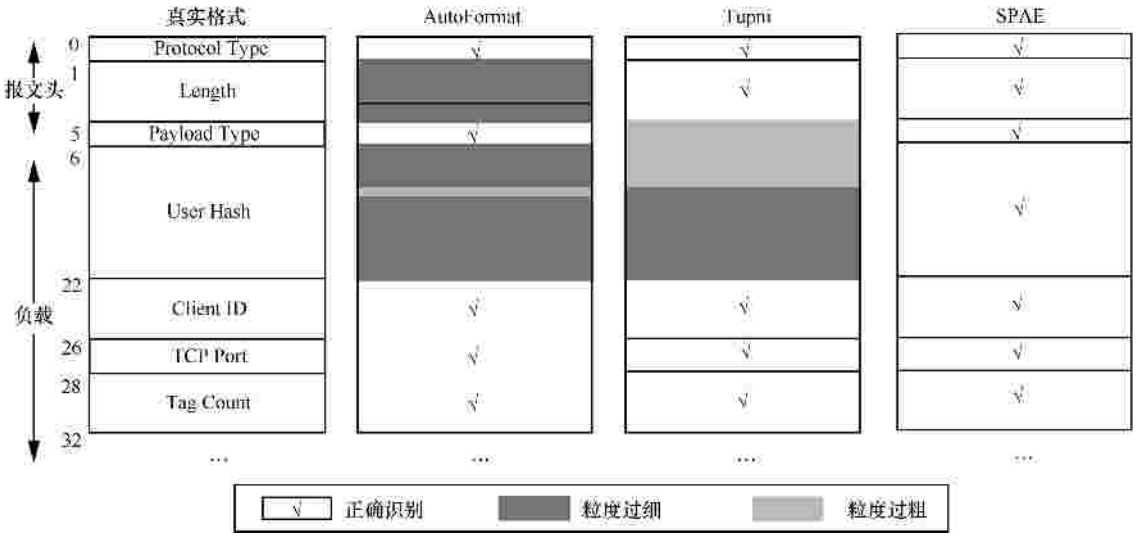


图 6　eDonkey Hello 报文的字段识别结果对比

示的方式进行解析，对 Protocol Type 字段和 Length 字段访问边界分别为<0,3>、<1,4>。AutoFormat 在构造字段树时，将数据块<0,3>、<1,4>作为结构处理，识别的字段为<0,1>、<1,3>和<3,4>，导致对 Length 字段的划分粒度过细。尽管边界存在冲突，Tupni 能够正确识别出 Protocol Type 字段和 Length 字段。这是由于 Length 字段作为循环条件被反复访问，<0,3>的累计权值要远小于<1,4>，因而在计算加权集合覆盖的过程中被剔除。

程序对 Payload Type 字段的解析方式与 Protocol Type 字段相同。类似地，AutoFormat 将 User Hash 字段错误识别为 2 个划分粒度过细的字段。另外，由于数据块<5,8>的累计访问权值大于数据块<7,23>（<7,23>通过对同一指令连续访问的数据块合并得到），Tupni 将字段边界错误识别为<5,8>和<9,23>。相比 AutoFormat 和 Tupni，SPAE 采用更为精确的动态污点分析策略，避免了位运算造成的字段识别误差。

对于 DNS 查询报文，3 种方法正确识别的字段数分别为 9、9 和 10，如图 7 所示。通过分析发现，程序分别采用不同的指令对 Flags 字段的 2 个字节进行读取和运算，因而 AutoFormat 与 Tupni 将 Flags 字段错误识别为 2 个单字节字段。而 SPAE 只在 Flags 比较时实现字段识别，因而正确划分了该字段边界。此外，AutoFormat、Tupni 和 SPAE 将连续的 Answer RRs、Authority RRs 和 Additional RRs 字段识别为同一字段，导致对字段的划分粒度过粗。这是由于目标程序在解析请求报文时跳过了对这些字段的处理，因而执行轨迹中并不存在对这些字段的处理指令，3 种方法只能将未识别的连续字节区域合并为单独的字段。

对于 DHCP 协议的 BOOTP 请求报文，3 种方法的字段识别结果相同，除将 Transaction ID、Server host name 等 11 个字段被错误合并为 5 个字段粒度

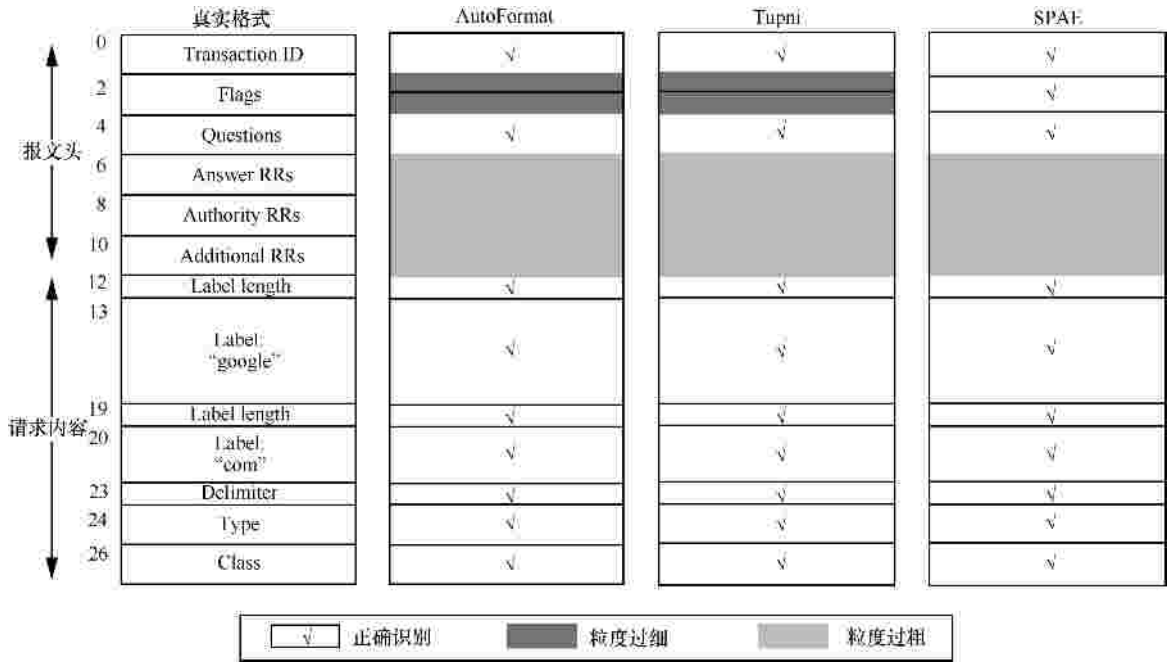


图 7 DNS 查询报文的字段识别结果对比

划分过粗的字段之外，正确识别了其余的 28 个字段。对执行轨迹的分析表明，字段误识别的原因与 DNS 报文解析类似，均在于目标程序未对这些字段进行处理。

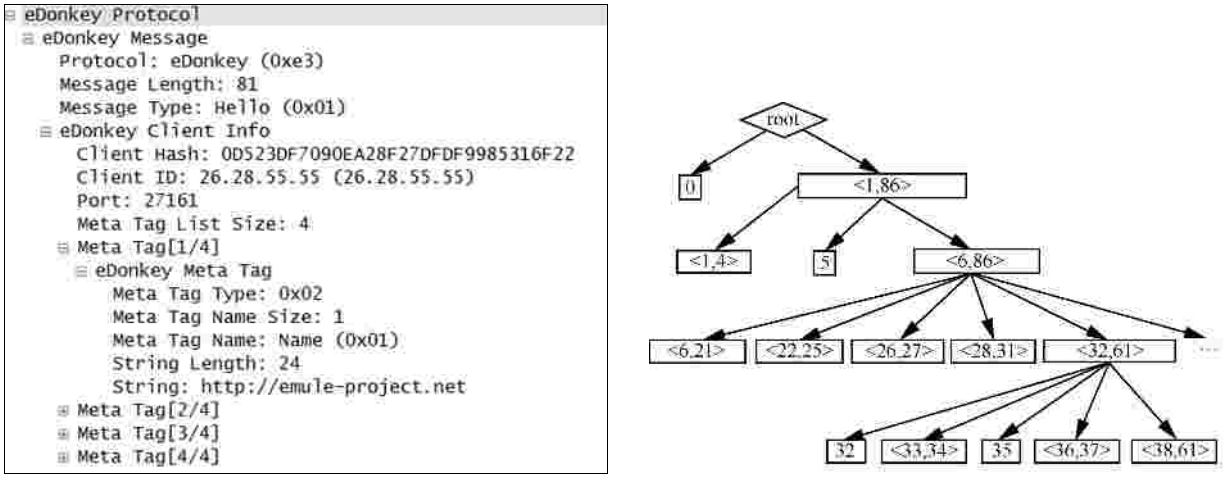
对于文本型的 FTP 协议，由于用户请求报文的格式较为简单，3 种方法均正确识别了所有的字段。需要说明的是，虽然程序以字节为粒度读取和处理字符串，AutoFormat 与 Tupni 通过对同一指令连续污点记录的合并避免了字段误识别；SPAE 则通过对字符串处理函数的抽象，实现了语义层次的文本型字段的识别。

对于 HTTP 请求报文和 SIP 注册报文，SPAE 的识别结果与协议规范一致，而 AutoFormat 和 Tupni 识别结果中存在少量的划分粒度过细的情况。通过分析发现，误识别的字段均为 Quality Factor 字段(如“q=0.500”)，用于指定服务器返回媒体类型的优先级。由于此类字段为字符串形式的浮点值，目标程序对小数点前后的字符采用了不同的加权方式，导致了 AutoFormat 和 Tupni 错误地将该字段细分为包括小数点在内的 3 个字段。可以看出，得益于语义层次的字段识别方式，SPAE 消除了类似语法解析过程给字段识别带来的误差。

对于 McAfee ePO 框架服务协议的策略检查报文，SPAE 正确识别的字段数远多于 AutoFormat 和 Tupni。通过对执行轨迹的分析发现，ePO 服务器首先将报文数据与 0xAA 进行异或，以完成对报文的解密。因而在字段树构造过程中，AutoFormat 将报文数据的每一字节都识别为字段，而将实际的报文字段识别为结构体。同样受到解密过程的影响，Tupni 仅能正确识别出累计访问权值超过自身长度的字段，其中包括 1 个类型字段与 26 个长度字段。而 SPAE 正确识别出了绝大部分的字段，仅报文尾部与数字签名相关的 5 个字段因客户端版本过低而未被服务器处理，导致了字段划分粒度过粗的误差。

在字段识别的基础上，还对报文样本的结构进行了识别。结果表明，SPAE 能够精确识别出所有报文样本的结构。以 eDonkey Hello 报文为例，Wireshark 和 SPAE 识别出的报文结构如图 8 所示。可以看出，SPAE 识别的报文结构层次与 Wireshark 的识别结果基本一致，仅对协议类型字段的识别存在差异。这是由于 Emule0.48a 同时支持协议类型取值为 0xC5 的 Emule 扩展协议，需要对协议类型字段进行比较以选择对应的指令分支，因此协议类型字段在 SPAE 识别结果中的层次要高于其他字段。由此说明，SPAE 的识别结果更加接近于程序对报文的解析方式。

SPAE 通过基于 BIL 的动态污点分析策略提高了字段识别的精确度，但可能存在时空开销过于庞大的问题。表 5 给出了 SPAE 与 tracecap 跟踪报文解析过程的时空开销对比。为了精确评估动态污点分析对效率的影响，在统计过程中 SPAE 未启用格



(a) wireshark　　(b) SPAE

图 8　eDonkey Hello **报文结构识别对比**

表 5　　**SPAE 与 tracecap 的时空开销对比**

报文类型	执行指令数	时间开销/s		内存开销/KB	
		tracecap	SPAE	tracecap	SPAE
DNS Query	92 169	<1	19	324	2 392
eDonkey Hello	7 453 698	37	1 748	417	2 368
DHCP BOOTP Request	479 348	2	142	495	2 426
FTP USER Request	64 855	<1	26	308	2 304
HTTP GET Request	389 491	2	171	667	2 457
SIP Register	5 812 767	25	1 502	872	2 480
ePO POST PolicyCheck	1 984 856	11	1 184	1 231	2 521

式提取模块。

从表 5 中可以看出，SPAE 的时间开销约为 tracecap 的 30~70 倍。特别是对于拥有图形界面的协议解析程序(eDonkey 和 SIP)，协议格式提取过程耗费的时间接近 0.5 h。但考虑到漏洞挖掘、协议重用等逆向应用对实时性的要求较低，认为以这样的时间开销为代价换取格式提取的精确度是值得的。另外，SPAE 使用的内存空间要明显多于 tracecap，但仍在可接受的范围之内。同时 SPAE 的内存开销并未随着报文数据规模的增大而显著增长，由此验证了文献[14]的结论：基于 roBBD 的污点属性存储的开销并不依赖于污点数据的多少，更适用于大规模的细粒度动态污点分析。

5　讨论

由于缺乏公开的测试样本集和原型工具，本文对 AutoFormat 和 Tupni 的字段识别部分进行了实现，以对比 SPAE 与已有方法的字段识别准确度。结合已有测试结果[4,9]可以发现，本文中 AutoFormat 和 Tupni 的字段识别准确率与已有测试结果基本一致，仅因测试目标的不同而存在细微差别。由此说明，SPAE 在协议格式提取的精确度方面的确更具优势。此外认为，所提出的细粒度动态污点分析策略并不局限于协议逆向分析领域，还能与恶意攻击检测[18]、测试用例生成[19]等领域的研究成果相结合，进一步推进动态污点分析在网络安全各个领域的应用。

本文方法在抽象的中间语言层进行指令分析，在降低分析策略复杂度的同时，能够以较少的代价移植到其他指令系统，具有跨平台的优势。值得说明的是，在 SPAE 的设计架构中，完全可以将 TEMU 替换为其他二进制插桩工具，如 PIN、DynamoRIO 等，以提高执行轨迹生成的效率。但考虑到插桩操作可能对协议解析流程产生干扰，采用类似 TEMU 的虚拟机系统得到的分析结果更为可靠。另外，本文方法还可以采用在线方式实现，以避免记录指令

数目庞大的程序执行轨迹。但采用离线方式使执行轨迹记录阶段和协议格式提取阶段分离，能够避免分析过程给目标程序执行带来过高的时延，确保报文解析流程的正常执行；同时执行轨迹分析可以在多个更高性能的计算平台上并行实现，以提高协议格式提取的效率。

隐式信息流的跟踪是动态污点分析所面临的一个重要挑战，目前还没有通用可行的解决方案。已有研究主要采用指针污染和控制流污染的方式，极易引起过污染 (over-tainting) 的问题[20,21]。与程序异常检测不同，协议格式提取对过污染缺陷更为敏感。虽然欠污染 (under-tainting) 可能导致部分字段无法识别，但过污染将给整个协议格式提取过程带来大量的干扰。目前在实验中还尚未发现欠污染给字段识别带来的影响，因此在动态污点分析策略中忽略了对隐式信息流的处理。

笔者注意到，包括 SPAE 在内的现有方法虽然实现了对字段边界和语义的识别，但无法获取字段之间的依赖关系。例如 SPAE 能识别出字段具有长度语义，但无法获知其取值所表示的真实长度。存在这一缺陷的原因在于动态污点分析仅能跟踪污点数据的传播，但不能提供协议解析过程对污点数据处理的运算关系。在未来工作中，将尝试引入基于中间语言的符号执行技术[13,14]，以获取更为丰富的协议格式信息。

目前，本文方法主要针对明文协议的格式提取，对复杂加密协议无能为力，并且没有考虑到对协议状态机的推断。但现有的加密协议逆向方法[5,22]和协议状态机推断方法[6,23]均以协议格式提取为基础，因而相信，结合本文提出的语义层次的字段识别思想，这些方法的识别准确率将得到进一步提高。

6 结束语

针对现有方法存在的字段识别准确度不高的缺陷，本文提出了一种语义层次的协议格式提取方法。该方法将二进制指令转换为语义等价的中间语言形式并进行重放，通过细粒度的动态污点分析技术跟踪报文字段的语义解析过程，在语义层次实现了对协议格式的提取。对原型系统的测试结果表明，本文提出的方法能够以可接受的时间和内存开销实现协议格式的提取，相比于 AutoFormat 和 Tupni 具有更高的识别准确率。

下一步，计划结合符号执行技术以增加对字段依赖关系的提取能力，并实现更为完善的支持加密协议和状态协议识别的协议逆向系统。

参考文献：

[1] CUI W, KANNAN J, WANG H. Discoverer: automatic protoco reverse engineering from network traces[A]. The 16th USENIX Security Symposium[C]. Boston, Sheraton, USA, 2007. 199-212.

[2] 李伟明,张爱芳,刘建财等. 网络协议的自动化模糊测试漏洞挖掘方法[J].计算机学报,2011, 34(2):242-255.
LI W M, ZHANG A F, LIU J C, *et al*. An automatic network protocol fuzz testing and vulnerability discovering method[J]. Chinese Journal of Computers, 2011, 34(2):242-255.

[3] CABALLERO J, YIN H, LIANG Z, *et al*. Polyglot: automatic extraction of protocol format using dynamic binary analysis[A]. ACM CCS[C]. Alexandria, VA, USA, 2007.317-329.

[4] CUI W, PEINADO M, CHEN K, *et al*. Tupni: automatic reverse engineering of input formats[A]. ACM CCS[C]. Alexandria, VA, USA, 2008. 391-402

[5] CABALLERO J, POOSANKAM P, KREIBICH C, *et al*. Dispatcher: enabling active botnet infiltration using automatic protocol Reverse-engineering[A]. ACM CCS[C]. Chicago, IL, USA, 2009. 621-634.

[6] COMPARETTI P, WONDRACEK G, KRUEGEL C, *et al*. Prospex: protocol specification extraction[A]. IEEE S&P[C]. Oakland, California, USA, 2009.110-125.

[7] 应凌云,杨轶,冯登国等. 恶意软件网络协议的语法和行为语义分析方法[J]. 软件学报, 2011, 22(7): 1676-1689.
YING L Y, YANG Y, FENG D G, *et al*. Syntax and behavior semantics analysis of network protocol of malware[J]. Journal of Software, 2011, 22(7): 1676-1689.

[8] WONDRACEK G, COMPARETTI P, KRUEGEL C, *et al*. Automatic network protocol analysis[A]. NDSS[C]. San Diego, Cali nia, USA, 2008.1-14.

[9] LIN Z, JIANG X, XU D, *et al*. Automatic protocol format reverse engineering through context-aware monitored execution[A]. NDSS[C]. San Diego, California, USA, 2008.29-43.

[10] GODEFRIOD P, KIEZUN A, LEVIN M, Grammar-based whitebox fuzzing[J]. ACM SIGPLAN Notices, 2008, 43(6):206-215.

[11] BRUMLEY D, JAGER I, AVGERINOS T, *et al*. BAP: a binary analysis platform[A]. ACM CAV[C]. Snowbird, Utah, USA, 2011.463-369.

[12] NICHOLAS N. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy[D]. University of Cambridge, 2004.

[13] EDWARD J, AVGERINOS T, BRUMLEY D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[A]. IEEE S&P[C]. Oakland, California, USA, 2010. 317-331.

[14] **王铁磊. 面向二进制程序的漏洞挖掘关键技术研究**[D]. **北京: 北京大学**, 2011.
WANG T. Research on Binary-Executable-Oriented Software Vulnerability Detection[D]. Beijing: Peking University, 2011.

[15] LIN Z, ZHANG X. Deriving input syntactic structure fro execution[A]. ACM SIGSOFT[C]. Atlanta, Georgia, USA, 2008. 83-93.

[16] YIN H, SONG D. TEMU: binary code analysis via whole-system layered annotative execution[EB/OL]. http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html, 2010.

[17] A BDD package[EB/OL]. http://buddy.sourceforge.net/.

[18] **刘豫, 聂眉宁, 苏璞睿等. 基于可回溯动态污点分析的攻击特征生成方法**[J]. **通信学报**, 2012, 33(5):21-28.
LIU Y, NIE M Y, SU P R, *et al.* Attack signature generation by traceable dynamic taint analysis[J]. Journal on Communications, 2012, 33(5):21-28.

[19] **陈恺,冯登国,苏璞睿等.基于彩色污点传播的黑盒测试方法**[J]. **中国科学:信息科学**, 2011, 41(5):526-540
CHEN K, FENG D G, SU P R, *et al.* Black-box testing based on colorful taint analysis[J]. Science China Information Science, 2011, 41(5):526-540.

[20] SLOWINSKA A, BOS H. Pointless tainting?: evaluating the practicality of pointer tainting[A]. ACM European Conference on Computer Systems[C]. Nuremberg, Germany, 2009. 61-74.

[21] KAND M, MCCAMANT S, POOSANKAM P, *et al.* DTA++:dynamic taint analysis with targeted control-flow propagation[A]. NDSS[C]. San Diego, California, USA, 2011.26-39.

[22] WANG Z, JIANG X, CUI W, *et al.* ReFormat: automatic reverse engineering of encrypted messages[A]. CESORICS[C]. Athens, Greece, 2010. 200-215.

[23] CHO C, DOMAGOJ B, SHIN E, *et al.* Inference and analysis of formal models of botnet command and control protocols[A]. ACM CCS[C]. Chicago, Illinois, USA, 2010. 426-439

作者简介：

潘璠（1987-），男，安徽芜湖人，解放军理工大学博士生，主要研究方向为协议逆向分析、漏洞挖掘。

洪征（1979-），男，江西南昌人，博士，解放军理工大学副教授，主要研究方向为网络信息安全、人工智能。

周振吉（1985-），男，江苏连云港人，解放军理工大学博士生，主要研究方向为虚拟化安全、云计算安全。

吴礼发（1968-），男，湖北蕲春人，博士，解放军理工大学教授、博士生导师，主要研究方向为网络信息安全、Web 服务安全。