基于存储访问模型的细粒度存储变量识别算法

井 靖[1] 蒋烈辉[1,2] 何红旗[1,2] 张媛媛[3]

（信息工程大学计算机科学与技术学院 郑州 450000）[1]

（数学工程与先进计算国家重点实验室 郑州 450000）[2] （72495部队自动化站 郑州 450002）[3]

摘 要 现阶段对变量的识别通常采用基于特定编译习惯及内存访问地址模式匹配的方法，或基于内存模型和抽象解释的分析方法。前者针对性太强，不具备普适性；后者通常采用损失算法精度的方法来得到结果，这往往会造成识别变量粒度过大、漏识别和误识别率较高。首先定义一种存储访问模型，对存储操作进行细粒度的模拟；然后给出基于存储访问模型的抽象状态生成算法，实现了基于高级中间语言HBRIL的细粒度数据信息（抽象状态）的跟踪和记录；基于这些抽象状态设计了存储区域内的细粒度变量实体识别算法；最后通过测试给出变量识别的细化比例和识别率。由测试结果可以看出，该算法在动态分配变量的识别率方面具有明显优势。

关键词 细粒度内存访问模型，存储环境，存储操作模拟，变量实体，抽象状态

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.9.033

Fine-grained Variable Entity Identification Algorithm Based on Memory Access Model

JING Jing[1] JIANG Lie-hui[1,2] HE Hong-qi[1,2] ZHANG Yuan-yuan[3]

(Department of Computer Science and Technology, PLA Information Engineering University, Zhengzhou 450000, China)[1]

(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China)[2]

(Automation Station, Union 72495, Zhengzhou 450002, China)[3]

Abstract There are two popular methods for variable identification. One is based on specific compiler habits and matching on memory address access mode, another is based on memory model and abstract interpretation technology. The former method is applicable to some specific compliers; the latter one often gets coarse-grained variables and higher wrong identification rate, because it has to consider the balance of accurate and time costs. In this paper, a fine-grained memory access model was defined firstly, which can simulate the fine-grained memory operation. And an abstract-state generation algorithm was given based on this model, which can track and record the fine-grained data information for advanced intermediate language HBRIL. Then a novel variable entity identification algorithm on memory region was designed according to data information. At last, the variables' refinement proportion and recognition rate were given. The test results show that our approach gets higher identification ratio for dynamic allocated variables.

Keywords Fine-grained memory access model, Memory environment, Memory operation simulate, Variant entity, Abstract state

变量识别是反编译中的一个关键环节，即通过对汇编代码或中间语言代码的数据流分析，识别出具有独立变量特征的数据块的过程。准确识别变量一方面有助于反编译数据类型的精确恢复，有效提高二进制文件的可读性；另一方面有助于分析敏感变量的使用行为，有效提高二进制文件脆弱性发现及漏洞挖掘的效率[1]。

按照数据存储方式的不同，可将变量识别分为寄存器变量识别和存储变量识别两大类。寄存器变量的识别是指消除变量对寄存器使用的冲突，实现寄存器使用范围的划分，并将每个划分对应为变量的过程。通常使用基于静态单一赋值SSA的算法[2]，对目标代码进行"使用-定义"分析和常量传播等，可较好地解决寄存器变量识别问题[3,4]。存储变量的识别是指通过对存储访问相关操作的分析，将内存空间按照使用情况进行划分，并将每个划分对应为变量的过程。存储器通常可以存放各种类型的数据，数据的寻址方式灵活多样。由于动态分配的数据其地址通常无法确定，多个过程中的变量常常会分时共享运行时栈空间，利用传统的数据流分析技术很难得到准确的内存块的地址及大小信息，因此要实现对内存空间的准确划分是非常困难的[5,6]。本文的核心内容就是讨论基于内存访问相关操作的分析，如何对内存空间完成细粒度的划分，进而实现对存储变量的准确识别。

1 相关工作

目前最具代表性的存储变量识别方法有semi-naïve算

到稿日期：2014-05-14 返修日期：2014-07-21 本文受国家自然科学基金项目（61272489）资助。

井 靖（1980－），女，博士生，讲师，CCF会员，主要研究方向为软件逆向分析、嵌入式系统、二进制翻译，E-mail：centerplain@163.com；蒋烈辉（1967－），男，博士，教授，博士生导师，主要研究方向为嵌入式系统、软件逆向分析、体系结构；何红旗（1972－），男，硕士，副教授，主要研究方向为嵌入式系统、软件逆向分析；张媛媛（1979－），女，硕士，主要研究方向为软件逆向分析。

法[7]，以及由 Balakrishnan 等人提出的值集分析(Value Set Analysis,VSA)算法等[8,9]。semi-naïve 算法采用顺序扫描的方式对程序中的指令进行遍历，并记录指令中使用寄存器 ESP 和 EBP 相对寻址的内存地址，将这些地址定义为抽象位置(a_loc)，抽象位置之间的内存块就被看作一个存储变量。这种算法虽然简单，但使得基于非 ESP 和 EBP 寄存器相对寻址的内存读写操作被直接忽略，因此适用范围有限，变量识别准确率低。VSA 算法定义了基于 X86 平台的内存抽象模型，基于抽象解释框架[10,11]选取值集(带有跨度的区间)作为抽象域，通过对 X86 平台上二进制可执行文件的内存访问信息进行跟踪、记录，结合聚集结构识别算法 ASI[8]进行迭代分析，最终实现对内存区域的划分，识别出存储变量。该算法与常规的静态分析方法相比取得了很大的突破，可部分解决指令中含间接寻址类操作数等问题，并在 BAP(binary analysis platform)[12]、TIE(type inference on executables)[13]、Bitblaze[14]等多个二进制文件分析平台中得到应用。然而该算法也存在很明显的缺陷：首先该算法选取值集作为抽象域，为了使算法能够终止，需要进行必要的加宽操作(Widing)，这就会使某些值集直接被近似为最大⊤，造成一些存储变量无法正确识别；此外该算法存在 VSA-ASI 之间的反复迭代，时间复杂度较高。

本文结合 semi-naïve 算法和 VSA 算法的缺点，对 Balakrishnan提出的内存抽象模型进行了改进，设计了一种细粒度内存访问模型(FMAM)，并基于该模型给出了抽象状态的定义及生成算法；最后基于抽象状态设计了细粒度存储变量识别算法，并通过实验对该算法的有效性进行了验证。

2 细粒度内存访问模型(FMAM)

细粒度内存访问模型是一种用于跟踪和记录程序中与内存读写访问相关的访问跟踪模型，主要包括存储环境描述和存储操作模拟两部分。其中存储环境描述是指对存储区域的描述和抽象地址的描述，实现内存空间粗粒度的划分；存储操作的模拟则是对有关内存读/写操作的模拟执行，实现内存空间细粒度的划分。

2.1 存储环境描述

广义上的存储环境(Memory Environment)包括虚拟内存(Virtual Memory)、寄存器和标志位等部分信息，而本文中的存储环境主要针对虚拟内存区域的信息，不考虑寄存器及标志位的信息。

定义 1(存储区域 MR(Memory Region)) 根据某种规则将存储空间划分为若干区域，这些区域统称为存储区域。其中存储空间是指一个进程所能使用的虚拟内存空间(Virtual Memory Region)。

定义 2(抽象地址 Abs_addr(Abstract Address)) 设 P 是二进制可执行程序，则称二元组(MR,$offset$)为 P 中的抽象地址。其中 MR 表示 P 的存储区域，$offset$ 表示距离存储区域起始地址的偏移量。

规则 1(生存区划分规则) 根据存储空间存储数据的生存期即存储期(storage duration)[15]可将存储空间划分为若干个生存区，包括静态生存区、自动生存区和动态生存区等。

对于一个可执行程序的虚拟内存空间，可划分为以下若干个生存区：1 个静态生存区(MR_S)、n 个自动生存区(MR_A_i)和 m 个动态分配生存区(MR_D_j)。其中，$0\leqslant i\leqslant n$，$0\leqslant j\leqslant m$，n 表示程序中包含的过程数，m 表示程序中动态申请堆块的数量。

1)静态生存区中的数据从程序开始运行就被分配和初始化，一直存在至程序运行结束，且始终位于内存的同一地址上。通常包括字符串常量以及程序员定义的其它全局变量，其中字符串常量具有只读属性 RO，程序员定义的全局变量具有可读可写属性 WR。

2)自动生存区中的数据通常是指那些被分配在栈上的变量，可能是局部变量 Local，也可能是函数参数 Para。当程序运行到它所在的模块时在栈上分配相应的空间，模块运行结束时对应的栈空间被释放。

3)动态分配生存区对应的是程序中通过 malloc()或 new 等操作在堆空间分配的内存空间，这部分内存空间是在程序执行过程中动态分配的，遇到 free()或者 delete 等操作才会被释放。

依据上述划分规则，存储区域可以形式化地表示为以下 $n+m+1$ 元组的形式，其逻辑组成如图 1 所示。

$$MR=(MR_A_1,\cdots,MR_A_n,MR_S,MR_D_1,\cdots,MR_D_m)$$

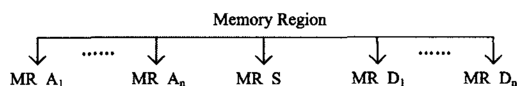

图 1 虚拟内存区域逻辑结构

图 2 是一段 C++源程序，图 3 是该程序对应的存储区域(生存区)。

```
#include <iostream>
using namespace std;
const double Pi=3.14;
double area(double p1);
int main(){
    double Radius1=5;
    double Radius2=6;
    double Area=0;
    Area=area(Radius1);
    Area=area(Radius2);
    double* heap1=new double;
    *heap1=Area;
    double* heap2=new double [10];
    for(int i=0;i < 10;i++)
    { heap2[i]=i; }
      delete heap1;
      delete [] heap2;
      return 0; }
double area(double p1){
    static double Sum=0;
    Sum += p1;
    return Pi * p1 * p1;}
```

图 2 C++源程序

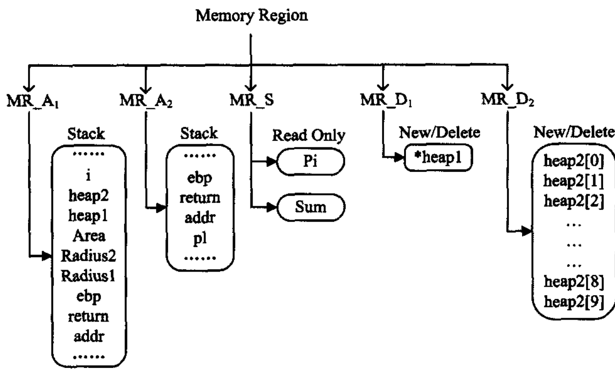

图 3　虚拟内存区域逻辑结构示例

说明 1　本文假设运行时程序只有一个栈帧处于活跃状态，多线程的情况不在本文考虑范围内。

说明 2　当 $MR=MR_S$ 时，$offset$ 表示变量的虚拟地址；当 $MR=MR_A_j(0\leqslant j\leqslant n)$ 时，$offset$ 表示当前地址处内存块的起始地址与当前栈帧指针的相对偏移量；当 MR 为 $MR_D_i(0\leqslant i\leqslant m)$ 时，$offset$ 表示当前地址处内存块的起始地址与当前申请的堆空间起始地址的相对偏移量。

2.2　存储操作模拟

与内存访问相关的机器指令主要包括：带有间接寻址操作数的指令、入栈/出栈指令和字符串传递指令等。存储操作的模拟需要记录上述指令中与内存访问相关的信息，包括“程序在什么时刻、什么地址，从内存读取（或向内存写入）多少字节的数据以及数据的内容是什么”等一系列信息的总和。因此首先需要定义一种内存访问信息的格式，其次给出从可执行程序中跟踪、提取内存访问信息的方法。

本文讨论的存储操作模拟面向一种较高级的中间代码形式 HBRIL，即基于通用中间语言 BRIL[16]，经过库函数识别[17,18]（包括 malloc()和 new 操作的识别）、静态单一赋值等优化操作后的形式。HBRIL 中间代码主要包括：(1)赋值语句 $Assign_{pc}$、$Assign_r$、$Assign_m$；(2)跳转语句 $Evaluate(e_1,e_2)$；(3)调用语句 Call；(4)返回语句 Ret 等。BRIL 的详细定义以及从机器指令到 BRIL 中间语言的转换、控制流图的生成以及相关优化可参考文献[16,19]，这里不再赘述。基于 HBRIL 中间代码的内存访问信息 I 以及模拟操作符◇的定义如下所示。

定义 3(内存访问信息 I)　与内存读写操作相关的属性信息可表示为下列四元组的形式：

$$I=(Abs_addr,Content,Size,R/W)$$

Abs_addr 表示抽象地址($MR,offset$)，见定义 2；$Content$ 表示向内存地址 Abs_addr 中写入或从中读出的数据内容；$Size$ 表示以字节为单位的内存访问大小；R/W 表示读写选项，读操作时 $R/W=0$，写操作时 $R/W=1$。

定义 4(提取操作符 Δ)　提取操作符实现从地址表达式 e 到抽象地址 Abs_addr 的映射，即转换函数 $\Delta:e\rightarrow Abs_addr$，详细定义如下：

$$\Delta e:=$$

$$\begin{cases}(MR_S,e), & \text{if } e \text{ 是直接寻址方式中的绝对地址}\\(MA_A_j,offset(e)), & \text{if } e \text{ 是基于 } esp/ebp \text{ 的间接寻址或基于 } esp/ebp \text{ 的间接寻址}\\(MR_D_i,offset(e)), & \text{if } e \text{ 是包含 } malloc() \text{ 或 } new \text{ 返回值的地址}\end{cases}$$

定义 5(模拟操作符◇)　模拟操作符实现从 HBRIL 中间代码语句 $stat$ 到内存访问信息集合的映射，即转换函数◇：$stat\rightarrow 2^I$，详细定义如下：

$$\Diamond(stat):=$$

$$\begin{cases}(\Delta(e),Fe(\not m),Size(e(\not m)),1), & \text{if } stat=[m(e):=e(\not m)]\\(\Delta(e)),Fm(e),Size(m(e)),0), & \text{if } stat=[\not m:=m(e)]\\(\Delta(e_1),Fm(e_2),Size(m(e_2)),1)\cup(\Delta(e_2),Fm(e_2),Size(m(e_2)),0), & \text{if } stat=[m(e_1):=m(e_2)]\\\varnothing, & \text{other}\end{cases}$$

其中，Δ 表示提取操作符，$stat$ 表示 HBRIL 中间代码中的语句，m 表示存储类操作数，$\not m$ 表示寄存器、pc 等非存储类操作数；$e(\not m)$ 表示不带有存储类操作数的表达式；$Fm(e)$ 表示存储类操作数的值，$Fe(\not m)$ 表示不带有存储类操作数表达式的值；$Size(\cdots)$ 表示操作数所占的字节数。

以图 2 中源代码编译生成的可执行程序为例，表 1 列出了经过存储操作模拟记录的部分结果。其中每一行表示可执行程序中一条汇编指令对应的 HBRIL 中间代码，以及经过存储操作模拟得到的内存访问信息。依据这些内存访问信息能够准确清晰地看到虚拟内存区域中数据的存储行为，为进一步实现内存区域的细粒度划分提供基础。

表 1　内存操作模拟记录的信息

.text:00414406	movsd [ebp+Radius1], xmm0
HBRIL 代码	m_u64[r_ebp_u32−0ch]=R_xmm0_u64
内存访问信息	((MR_A, −0Ch), (xmm0), 8, 1)
.text:00414413	movsd [ebp+Radium2], xmm0
HBRIL 代码	m_u64[r_ebp_u32−1ch]=R_xmm0_u64
内存访问信息	((MR_A, −1Ch), (xmm0), 8, 1)
.text:00414418	mov ?Pt@@3HA, 4E20h
HBRIL 代码	m_u32[41f220h]=4e20h
内存访问信息	((MR_S, 41f220h), 4e20h, 4, 1)
.text:00414432	movsd xmm0, [ebp+Radius1]
HBRIL 代码	r_xmm0_u64=m_u64[r_ebp_u32−0ch]
内存访问信息	((MR_A, −0Ch), ([ebp−0ch]), 8, 0)
.text:00414437	movsd [esp+158h+v_158], xmm0
HBRIL 代码	m_u64[r_esp_u32]=R_xmm0_u64
内存访问信息	((MR_A, −158h), (xmm0), 8, 1)
.text:00414469	mov [ebp+v_140], eax
HBRIL 代码	m_u32[r_ebp−140]=r_eax_u32
内存访问信息	((MR_A, −140h), (eax), 4, 1)
.text:0041446F	mov eax, [ebp+v_140]
HBRIL 代码	r_eax_u32=m_u32[r_ebp−140]
内存访问信息	((MR_A, −140h), (ebp−140), 4, 0)
.text:0041341E	movsd xmm0, Sum
HBRIL 代码	r_xmm0_u64=m_u64[41f228h]
内存访问信息	((MR_S, 41f228h), ([41f228h]), 8, 0)
.text:00413426	addsd xmm0, [ebp+p1]
HBRIL 代码	r_xmm0_u64=r_xmm0_u64+m_u64[r_ebp_u32+8]
内存访问信息	((MR_A, 8h), ([ebp+8]), 8, 0)
.text:0041342B	movsd Sum, xmm0
HBRIL 代码	m_u64[41f228h]=r_xmm0_u64
内存访问信息	((MR_S, 41f228h), (xmm0), 8, 1)
.text:00413433	movsd xmm0, ds:Pi
HBRIL 代码	r_xmm0_u64=m_u64[41cac0h]
内存访问信息	((MR_S, 41cac0h), ([41cac0h]), 8, 0)
text:004144C0	movsd qword ptr [ecx+eax*8], xmm0
HBRIL 代码	m_u64[r_ecx_u32+r_eax_u32*8]=r_xmm0_u64
内存访问信息	((MR_D_414486, eax*8), (xmm0), 8, 1)

3 基于细粒度存储访问模型的抽象状态生成

2.2 节中描述了如何通过提取操作符和模拟操作符获得代码中的内存访问信息。其中提取操作符通过判断内存访问地址 e 所在的内存区域来得到其对应的抽象地址 Abs_addr；模拟操作符在得到抽象地址的基础上，计算表达式 $Fe(\cdots)$ 和 $Fm(\cdots)$ 的值来获得数据的内容等信息。而内存访问地址 e、$Fe(\cdots)$ 以及 $Fm(\cdots)$ 的值通常与当前的寄存器以及内存单元的值相关，因此必须跟踪并记录这些值（或者叫状态）才能完成内存访问信息的精确提取。

3.1 抽象状态的定义

定义 6（抽象状态（$\hat{S}$）） 抽象状态$\hat{S}$是指每条语句（这里是指 HBRIL 语句）执行后，所有寄存器（包括）以及所有访问过的内存单元的值可表示为笛卡儿积的形式：$\hat{S}=\hat{S}_{PC}\times\hat{S}_R\times\hat{S}_M$。

设语句 $stat$ 所在的地址为 a，执行完 $stat$ 的下一条语句地址为 a'，则$\hat{S}(a')$表示执行语句 $stat$ 之后的抽象状态。其中 $\hat{S}(a').\hat{S}_{PC}$ 表示 $stat$ 执行后程序计数器 pc 的值，即 $\hat{S}(a').\hat{S}_{PC}:=pc\rightarrow A$，$A$ 表示地址集合；$\hat{S}(a').\hat{S}_R$ 表示其他寄存器的状态，即$\hat{S}(a').\hat{S}_R:=R\rightarrow Exp$，$R$ 表示寄存器集合，Exp 表示表达式；$\hat{S}(a').\hat{S}_M$ 表示访问过的内存单元的状态，即 $\hat{S}(a').\hat{S}_M:=M(Exp_1)\rightarrow Exp_2$，$Exp_1$ 表示内存单元的地址，Exp_2 表示内存单元存放的数值。

由前面定义的细粒度存储访问模型可知，进程所使用的内存空间被划分为 3 种不同的区域，不同区域中的内存单元可表示为抽象区域加偏移的形式，故内存单元状态$\hat{S}_M$ 中的内存地址 Exp_1 也表示为抽象地址的形式，比如初始抽象状态定义为：$\hat{S}_0=(\text{start},\{esp\rightarrow(MR_A,0),eax/ebx/ecx\cdots\rightarrow \text{null}\},\{(MR_S,addr_i)=val_i\mid 1\leqslant i\leqslant n$，$n$ 为全局变量的个数$\})$，即表示 pc 指向入口地址；esp 指向栈空间的栈底，其他寄存器初始化为空；地址为 $addr_i$ 的内存单元（全局变量）的初值为 val_i。

3.2 抽象状态生成算法

假设已有基于 HBRIL 的控制流图（生成算法可参考文献[19]），E_{CFG}表示控制流图中边的集合，每条边由三元组（a，$stat$，a'）表示，其中 a 和 a'分别表示边的起始地址和目的地址，$stat$ 表示地址 a 处的语句，执行语句 $stat$ 之前的抽象状态用$\hat{S}(a)$表示，执行语句 $stat$ 后的抽象状态用$\hat{S}(a')$表示。则 $G(\hat{S}(a),stat)$表示抽象状态的生成算子，即由起始地址的抽象状态$\hat{S}(a)$以及语句 $stat$ 生成目的地址的抽象状态$\hat{S}(a')$，其定义如下所示（其中 $Fe(e)\mid\hat{S}(a)$表示在地址 a 处抽象状态下计算表达式 e 的值，$Fm(e)\mid\hat{S}(a)$表示在地址 a 处抽象状态下计算地址 e 处内存单元的值，$\hat{S}(a).\hat{S}_{PC}+1$ 表示地址 a 处指令 $stat$ 的下一条指令的地址，$\hat{S}(Lib)$表示经过库函数调用返回后的状态）：

$$G(\hat{S}(a),stat):=\begin{cases}(Fe(e)\mid\hat{S}(a),\hat{S}(a).\hat{S}_R,\hat{S}(a).\hat{S}_M), & \text{if } stat=[pc:=e]\\(\hat{S}(a).\hat{S}_{PC}+1,\hat{S}(a).\hat{S}_R\cup[r\rightarrow Fe(e)\mid\hat{S}(a)\},\hat{S}(a),\hat{S}M), & \text{if } stat=[r:=e]\\(\hat{S}(a).\hat{S}_{PC}+1,\hat{S}(a).\hat{S}_R,\hat{S}(a).\hat{S}_M\cup\{Fm(e_1)\mid\hat{S}(a))\rightarrow Fe(e_2\mid\hat{S}(a)\}), & \text{if } stat=[m(e_1):=e_2]\\(Fe(e_2)\mid\hat{S}(a),\hat{S}(a).\hat{S}_R,\hat{S}(a).\hat{S}_M), & \text{if } stat:=[\text{Evaluate}(e_1,e_2)],\text{and } Fe(e_1)\mid\hat{S}(a)\neq 0\\(\hat{S}(a).\hat{S}_{PC}+1,\hat{S}(a).\hat{S}_R,\hat{S}(a),\hat{S}_M), & \text{if } stat:=[\text{Evaluate}(e_1,e_2)],\text{and } Fe(e_1)\mid\hat{S}(a)=0\\(\hat{S}(a).\hat{S}_{PC}+1,\hat{S}(Lib).\hat{S}_R,\hat{S}(Lib).\hat{S}_M), & \text{if } stat:=\text{Call } Lib\\((\text{esp})\mid\hat{S}(a),\hat{S}(a).\hat{S}_R\cup\{esp\rightarrow esp+4\},\hat{S}(a).\hat{S}_M), & \text{if } stat:=\text{Ret}\end{cases}$$

利用初始抽象状态$\hat{S}_0$ 以及控制流图 E_{CFG} 中边的集合，迭代运用抽象状态生成算子，即可得到抽象状态的生成算法 $Abs_State_Generate(\hat{S}_0,E_{CFG})$，伪代码如下所示。

算法 1 抽象状态生成算法

```
输入：初始状态Ŝ0，控制流图 ECFG
输出：抽象状态集Ŝ
Proc Abs_State_Generate(Ŝ0,ECFG)
  为入口点以外的每条语句 stat 的抽象状态赋初值Ŝ⊥；
  为入口点语句 stat0 的抽象状态赋初值Ŝ0；
  将入口点语句 stat0 所在的地址 a0 加入 worklist；
  While(worklist ≠∅) do
    从 worklist 取出一个地址 a；
    在 ECFG 中查找以 a 为起始地址的边；
    For(i=1;i≤k;i++) do
      //k 为以 a 为起始地址边的个数
     If(G(Ŝ(a),stati) ⋢ Ŝ(ai)) do
      //得到 a1,…,ak 地址处的抽象状态
        Ŝ(ai)=Ŝ(ai)∪G(Ŝ(a),stati);
      将 ai 加入工作列表 worklist;
    End If
   End For
  End While
End Proc
```

算法的正确性说明：该算法基于抽象解释框架，抽象状态生成算子是对真实程序状态的保守逼近，很容易证明抽象状态$\hat{S}=\hat{S}_{PC}\times\hat{S}_R\times\hat{S}_M$ 的生成符合抽象解释框架中递增链的要求，因此在理论上该算法是可以终止的。由于分支结构及循环结构的存在，地址 a'的抽象状态可能会不断被更新，为了保证算法最终能够终止，即达到不动点，可使用 widing 加宽操作。这里采用文献[18]中的精度可调策略，即当某处抽象状态更新的次数达到一个阈值时，就不再继续更新，以此来保证算法在精度和可行性之间取得一个平衡。

4 基于抽象状态的存储变量识别

4.1 存储变量实体

定义 7(存储变量实体(VarEntity)) 存储变量实体是指在过程 P 中具有独立数据意义的内存单元。设过程 P 中出现的抽象地址集合为:

$$Abs_addr=\{(MR_S,offset_i)\}\cup\{(MR_A,offset_j)\}\cup\{(MR_D,offset_k)\}$$

其中,$offset\in Z$, $i,j,k\in N$,则存储变量实体表示为:

$$VarEntity_i=\{(Abs_addr_i,lowerbound_i,uperbound_i)\mid upperbound_i\leqslant lowerbound_{i+1}(1\leqslant i\leqslant N)\}$$

其中,$lowerbound_i$ 表示变量实体的下边界(起始相对偏移地址),$upperbound_i$ 表示变量实体的上边界。

假设内存访问地址集合按照其地址偏移大小由小到大排列,即 $offset_i<offset_{i+1}$,则任意两个内存访问地址对应的内存块的位置关系如图 4 所示,其中 $size_i$ 表示内存访问操作的大小。

图 4(a)表示两个内存访问操作的内存块正好相邻,即 $offset_i+size_i=offset_{i+1}$。

图 4(b)表示两个内存块之间有间隙,即 $offset_i+size_i<offset_{i+1}$。

图 4(c)表示两个内存块之间有交叉,即 $offset_{i+1}<offset_i+size_i\leqslant offset_{i+1}+size_{i+1}$。

图 4(d)表示两个内存块之间有包含关系,即 $offset_i+size_i>offset_{i+1}+size_{i+1}$。

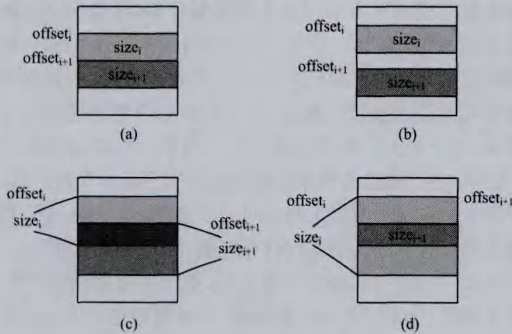

图 4 内存抽象地址信息示意图

图 5(a)-(d)分别表示图 4(a)-(d) 4 种情况下对应的存储变量实体,lb_j 表示变量实体的下边界,ub_j 表示变量实体的上边界,$1\leqslant j\leqslant N$。

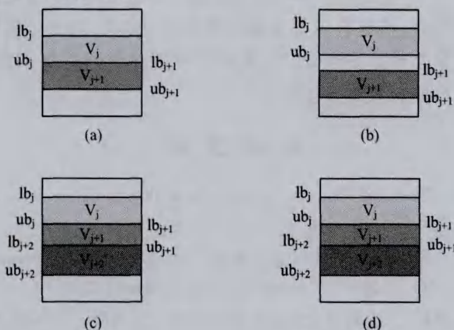

图 5 存储变量实体示意图

4.2 细粒度存储变量识别算法

由 4.1 节定义 7 可知,存储变量实体是一个存储区域内互相之间没有重叠的内存块,这些内存块实际上就是细粒度的存储变量,因此细粒度存储变量的识别就是对存储变量实体的识别。首先需要从代码(这里指 HBRIL 中间代码)中提取所有存储访问相关的内存块的内存访问信息,然后根据这些内存块之间的关系来识别存储变量实体。

内存访问信息的提取主要由模拟操作符◇来完成,即根据每条语句的抽象状态 $\hat{S}(a)$ 来提取与存储操作相关的抽象地址 Abs_addr、表达式的值 $Fe(\cdots)$、大小 $Size$ 及读写状态 R/W 等信息。因此定义 5 中的模拟操作符可扩展为基于抽象状态的模拟操作符 $\Diamond\mid\hat{S}$,其定义如下所示:

$$\Diamond\mid\hat{S}(a,stat):=$$

$$\begin{cases}(\Delta(Fe(e)\mid\hat{S}(a)),Fe(\not m)\mid\hat{S}(a),Size(e(\not m)),1), & \text{if } stat=[m(e):=e(\not m)]\\ (\Delta(Fe(e)\mid\hat{S}(a)),Fm(e)\mid\hat{S}(a),Size(m(e)),0), & \text{if } stat=[\not m:=m(e)]\\ (\Delta(Fe(e_1)\mid\hat{S}(a)),Fm(e_2)\mid\hat{S}(a),Size(m(e_2)),1)\cup \\ \quad(\Delta(Fe(e_2)\mid\hat{S}(a)),Fm(e_2)\mid\hat{S}(a),Size(m(e_2)),0), & \text{if } stat=[m(e_1):=m(e_2)]\\ \varnothing, & \text{others}\end{cases}$$

基于提取出的内存访问信息,可进一步判断、识别存储变量实体。设 P 表示 HBRIL 代码,I 表示由 $\Diamond\mid\hat{S}$ 操作符得到的内存访问信息集合,$I.Abs_addr$、$I.Contant$、$I.Size$、$I.rw$ 分别表示对应内存访问中内存块的抽象地址、数据内容、大小以及读写属性。首先,按照抽象中存储区域的不同将内存访问信息集合进行一次划分,即属于不同内存区域 MR 的内存访问信息被划分在不同的集合中;其次,对每个集合中的内存访问信息按照其抽象地址中偏移位置 $offset$ 从小到大顺序排列,$offset$ 相同的,按照 $size$ 的大小从小到大排列,最终组成一个队列 $Queue_MR$;基于某个队列 $Queue_MR$ 生成其存储区域内的细粒度存储变量实体的过程如算法 2 所示。

算法 2 存储区域内变量实体生成算法(也可称为基于滑动窗口的变量实体生成算法)

输入:某个存储区域内的内存访问信息队列 Queue_MR
输出:该存储区域内的变量实体集合 V

```
Proc Var_Entities_Window(Queue_MR)
  V=∅;i=0;num=0;
  //V 表示变量实体的集合,num 表示内存访问信息的个数
  winb=q[0].offset;//初始化滑动窗口的低地址
  wina=q[0].offset+q[0].size;//初始化滑动窗口的高地址
  while(i<num) do
    If(q[i+1].offset==winb) do
      //如果 q[i+1]的起始地址与 winb 一样
    V :=V∪(q[i].MR,winb,wina);
    V :=V∪(q[i].MR,wina,q[i+1].offset+q[i+1].size);
    End If
  If(winb<q[i+1].offset<wina && winb<q[i+1].offset+
  q[i+1].size < wina) do
    //当 q[i+1]的起始地址和结束地址都在 winb 和 wina 之间时
    V :=V∪(q[i].MR,winb,q[i+1].offset);
    V :=V∪(q[i].MR,q[i+1].offset,q[i+1].offset+q[i+1].
  size);
    V :=V∪(q[i].MR,q[i+1].offset+q[i+1].size,wina);
```

```
    End If
    If(winb <q[i+1].offset< wina && q[i+1].offset+q[i+1].
    size>wina) do
    //当 q[i+1]与 q[i]有交叉时
      V := V∪(q[i].MR,winb,q[i+1].offset);
      V :=V∪(q[i].MR,q[i+1].offset,wina);
    End if
    If(q[i+1].offset > wina) do
          //当 q[i+1]与 q[i]没有交叉重叠时
      V := V∪(q[i].MR,winb, wina);
    End If
    winb := MAX(winb, q[i+1].offset );//调整窗口的低地址
    wina := MAX(wina, q[i+1].offset + q[i+1].size);
          //调整窗口的高地址
  End while
End Proc
```

算法的正确性说明:该算法的正确性是由算法 1 的正确性保证的,因为算法 1 得到的抽象状态是对实际状态的保守逼近,所以基于抽象状态得到的内存访问信息也是实际内存访问信息的保守逼近,那么基于内存访问信息的变量实体生成也是对实际变量实体的保守逼近。设内存访问信息的条数为 n,则算法 2 的时间复杂度为 $O(n)$。

5 测试与比较

测试的目的是验证算法的有效性,即测试该算法对可执行程序中变量实体的识别率以及细化程度等。实验选用开源套件 coreutils8.20 中的部分程序,以及含有指针、虚函数的 C++程序:pointer.cpp 和 funcpp.cpp,程序的基本信息如表 2 所列。实验机型为联想 ThinkCentre M8400s,处理器为 Intel 酷睿 i5 2400,4GB 内存,主机操作系统为 Windows 7,虚拟机操作系统版本为 Ubuntu 12.04。测试程序先用编译器 GCC-0编译成可执行程序,然后利用反汇编器(可以是 IDA 或其他通用反汇编器)、中间代码转换程序和相关优化程序[16,18,19]将可执行程序转换为对应的 HBRIL 代码,最后调用抽象状态生成算法和基于抽象状态的细粒度存储变量识别算法得到各个存储区域内的细粒度存储变量实体。测试的部分统计结果如表 3 所列。

表 2 coreutils 及 C++测试程序列表

测试对象 (Test Prog)	指令条数 (Insts)	过程数 (Proc)	Malloc/calloc/new 等操作个数(Dyn)
dd	2303	36	2
factor	2531	35	2
install	1002	19	1
join	1185	26	1
ls	4882	78	9
cat	783	6	3
nl	627	11	4
od	1916	22	2
pr	2876	38	10
ptx	2150	22	8
sort	4709	96	16
tail	2219	32	8
tr	1947	35	9
wc	800	7	1
pointer	312	4	3
funcpp	549	8	5

细化比例是由识别出的变量实体个数与源代码中的变量个数的比值来标识的,表 3 中分别给出了测试对象在栈空间和堆空间的细粒度变量实体的个数及细化比例。栈空间变量实体的平均细化比例为 1.8,堆空间变量实体的平均细化比例为 2.4。

表 3 变量实体识别测试结果

测试对象	Ln	Hn	本算法识别的变量实体			
			栈变量实体个数	栈空间细化比例	堆变量实体个数	堆空间细化比例
dd	603	19	896	1.5	42	2.2
factor	597	19	875	1.5	39	2.1
install	369	10	598	1.6	31	3.1
join	402	9	676	1.7	21	2.3
ls	1375	32	1996	1.5	76	2.4
cat	292	21	478	1.6	48	2.3
nl	284	19	587	2.1	43	2.3
od	576	16	1045	1.8	37	2.3
pr	631	28	1254	2.0	58	2.1
ptx	612	22	1356	2.2	45	2.0
sort	1372	58	2537	1.8	121	2.1
tail	623	22	1134	1.8	50	2.3
tr	605	21	1323	2.2	46	2.2
wc	314	8	556	1.8	17	2.1
pointer	168	10	378	2.3	38	3.8
funcpp	269	18	596	2.2	49	2.7

本算法识别出的变量实体是细化了的变量,无法与源代码中的变量个数进行直接比较,但通常局部变量在编译后被分配在寄存器或栈空间中,动态分配的变量则被分配在堆空间中,因此这里可用已识别出的变量实体所占的空间大小与源代码中变量所占的大小的比值来近似表示算法的识别率。由于本测试没有统计寄存器变量的个数,因此局部变量的平均识别率为 85%。但本算法利用抽象状态生成准确保留并传递了 malloc/new 等动态分配内存操作的属性,故动态分配变量的平均识别率为 98%,与 semi-naïve 和 VSA-ASI 典型算法的部分测试用例测试结果相比具有明显优势。

结束语 本算法记录并跟踪细粒度的内存访问信息,用有限的状态集合来表示程序点的抽象状态,并基于这些抽象状态识别出变量实体。由测试结果可知,本算法在堆空间动态分配变量的平均识别率为 98%。另外与 VSA 中采用带跨度的区间来表示抽象域相比,本算法对每个变量属性的描述更精确,不会因为加宽等操作而丢失精度。在算法复杂度方面,本算法可以通过阈值来限定抽象状态更新的次数,可以在不丢失变量精度的前提下降低时间复杂度,而 VSA-ASI 算法需要在两个算法之间迭代执行很多遍,时间复杂度较高。

本文中的抽象状态生成算法中,由于采用阈值来控制算法的复杂度,会造成变量的漏识别,这种漏识别可以通过后期的行为分析加以修正;对已经识别出的库函数,暂时使用人工的方式设置其对调用者状态的影响,在以后的工作中,可以对已识别的库函数建立抽象状态变迁描述,使过程间的抽象状态生成更加自动化和高效。基于本算法对变量实体的识别,收集了大量与变量实体相关的信息,比如每个程序点的状态集合,根据这些集合可提供面向各种应用的数据流分析算法,判断变量实体之间的联系,从而完成更高级数据类型的恢复以及程序漏洞挖掘等工作。

参 考 文 献

[1] Lin Zhi-qiang. Reverse engineering of data structures from binary [D]. West Lafayette:Purdue University,2011

[2] Van Emmerik M . Single Static Assignment for Decompilation [D]. Queensland:University of Queensland,2006

[3] 孙维新.二进制翻译中基本数据类型分析的研究与实现[D].郑州:解放军信息工程大学,2007
Sun Wei-xin. Study and implementation basic data type analysis in static binary translation[D]. Zhengzhou: PLA Information Engineering University,2007

(下转第 182 页)

vents/fast02/patterson/sld001. htm

[2] 钱迎进. 大规模 Lustre 集群文件系统关键技术的研究 [D]. 长沙:国防科技大学,2011:97-118
Qian Ying-jin. Research on Key Issues in Large Scale Clustered File System Lustre [D]. Changsha: National University of Defense Technology,2011:97-118

[3] 李晖. 基于日志的集群文件系统高可用关键技术研究[D]. 北京:中国科学院计算技术研究所,2005:9-10
Li Hui. Research on Journal Based High-availability Mechanism of Cluster File Systems [D]. Beijing: Institute of Computing Technology,Chinese Academy of Science,2005:9-10

[4] 钱迎进,伊瑞海,肖侬,等. Lustre 文件系统元数据服务恢复机制研究 [J]. 高性能计算技术(第 19 届全国信息存储技术大会收录),2013,255(6):10-16
Qian Ying-jin, Yi Rui-hai, Xiao Nong, et al. Research on Recovery Mechanism for Lustre Metadata Service [J]. High Performance Computing Technology (19th Annual National Conference on Information and Storage Technology),2013,255(6):10-16

[5] Bhide A, Elnozahy E N, Morgan S P. A Highly Available Network File Server [C]// Proceedings of the Usenix Winter 1991 Conference. Dallas,TX,USA: USENIX Association,1991:199-205

[6] Devarakonda M, Kish B, Mohindra A. Recovery in the Calypso File System [J]. ACM Transaction on Computer Systems,1996, 14(3):287-310

[7] Mogul J C. Recovery in Spritely NFS [J]. Computing Systems, the Journal of the USENIX Association, Spring, 1994, 7(2): 201-262

[8] Baker M, Ousterhout J. Availability in the Sprite Distributed File System [J]. Operating Systems Review,1991,25(2):95-98

[9] Welch B, Baker M, Douglis F, et al. Sprite Position Statement: Use Distributed State for failure Recovery [C]// Proceeding of the Second Workshop on Workstation Operating System. Pacific Grove,CA,USA: IEEE Computer Society,1989:130-133

[10] Baker M. Fast Crash Recovery in Distributed File Systems [D]. California: University of California at Berkeley,1994:34-104

[11] Kistler J, Satyanarayanan M. Disconnected Operation in the Coda File System [J]. ACM Transactions on Computer Systems, 1992,10(1):3-25

[12] 钱迎进,金士尧,肖侬. Lustre 文件系统 I/O 锁的应用与优化 [J]. 计算机工程与应用,2011,47(3):1-5,26
Qian Ying-jin, Jin Shi-yao, Xiao Nong. Application and Optimization for Lustre File I/O Locking [J]. Computer Engineering and Applications,2011,47(3):1-5,26

[13] 钱迎进,肖侬,金士尧. Lustre 分布式锁管理器的分析与改进 [J]. 计算机工程与科学,2009(S1):146-149
Qian Ying-jin, Xiao Nong, Jin Shi-yao. Analysis and Improvement of Lustre Distributed Lock Manager [J]. Computer Engineering & Science,2009(S1):146-149

(上接第 176 页)

[4] 何东,尹青,谢耀宾,等. 反编译中数据类型自动重构技术研究 [J]. 计算机科学,2012,39(5):133-136
He Dong, Yin Qing, Xie Yao-bin, et al. Automatic data type reconstruction in decompilation [J]. Computer Sciense, 2012, 39 (5):133-136

[5] 马金鑫,李舟军,忽朝俭,等. 一种重构二进制代码中类型抽象的方法[J]. 计算机研究与发展,2013,50(11):2418-2428
Ma Jin-xin, Li Zhou-jun, Hu Chao-jian, et al. A reconstruction method of type abstraction in binary code[J]. Journal of Computer Research and Development,2013,50(11):2418-2428

[6] Ding Wei, Gu Zhi-ming, Gao Feng. Reconstruction of data type in obfuscated binary programs[C]// 16th International Conference on Advanced Communication Technology. PyeongChang, South Korea,2014:393-369

[7] Balakrishnan G, Reps T. WYSINWYX: What you see is not what you execute[J]. ACM Transactions on Programming Languages And Systems,2010,32(6):202-213

[8] Balakrishnan G, Reps T. DIVINE: discovering variables in executables[C]// Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation. Nice, France,2007:1-28

[9] Anand K, Elwazeer K, Kotha A, et al. An accurate stack memory abstraction and symbolic analysis framework for executables [C]// 29th IEEE International Conference on Software Maintenance. Eindhoven, Netherland,2013:90-99

[10] Cousot P, Cousot R. Interpretation: A unified lattice model for static analysis[C]// Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New York, USA 1977:238-252

[11] 王雅文,宫云战,肖庆,等. 基于抽象解释的变量值范围分析及应用[J]. 电子学报,2011,39(2):296-302
Wang Ya-wen, Gong Yun-zhan, Xiao Qing, et al. A method of variable range analysis based on abstract interpretation and its applications[J]. ACTA Electronica Sinica,2011,39(2):296-302

[12] Brumley D, Jager I, Avgerinos T, et al. BAP: A binary analysis platform[C]// 23rd International Conference on Computer Aided Verification. Snowbird, UT, USA,2011:463-469

[13] Lee J, Avgerinos T, Brumley D. TIE: Principled Reverse Engineering of Types in Binary Programs[C]// Proceedings of the Network and Distributed System Security Symposium. San Diego, USA,2011: session 5

[14] Song D, Brumley D, Yin Heng, et al. BitBlaze: A new approach to computer security via binary analysis[C]// 4th International Conference on Information Systems Security. Hyderabad, India, 2008:1-25

[15] Aho A V, Lam M S, Sethi R, et al. Compilers: Principles, Techniques, and Tools (2nd Edition) [M]. Boston: Addison Wesley, 2007

[16] 刘絮颖. 反编译中控制流重构与控制结构恢复技术研究[D]. 郑州:解放军信息工程大学,2011
Liu Xu-ying. Research on technology of control flow reconstruction and control structure recovery in decompilation[D]. Zhengzhou: PLA Information Engineering University,2011

[17] Durfina L, Kroustek J, Zemek P, et al. Detection and recovery of functions and their arguments in a retargetable decompiler[C]// 19th Working Conference on Reverse Engineering. Kingston, Canada,2012:56-60

[18] 吴滨. 汇编级程序辅助分析中的库函数识别技术研究[D]. 郑州:解放军信息工程大学,2011
Wu Bin. Research on library function identification technology in assemble level program auxiliary analysis[D]. Zhengzhou: PLA Information Engineering University,2011

[19] Jing Jing, Jiang Lie-hui, Liu Tie-ming, et al. A precision-tunable CFG reconstruction algorithm[C]// International Conference on Mechatronic Sciences, Electric Engineering and Computer. Shenyang, China,2013:2095-2099