Library Functions Identification in Binary Code by Using Graph Isomorphism Testings

2019年12月22日 15:24

execution dependence grahgs(EDGs) to describe the behavior characteristics of binary code

两个作用：
一：节省时间
二：很多库函数结构很大，逆向花费很多时间

动态链接：有import tables，所以容易被识别

challenge:
一，inline库函数与普通函数区别不大

现在技术：
IDA:FLIRT，32字节的快速识别，对于内联函数有几个问题
一：内联函数不连续问题，编译器可能会改变一些指令的顺序
二：在不同的编译优化选项下，有不同的字节序列

目标：解决IDA两个问题

方法：EDG，是CFG的扩展
原理：
一、不管编译器怎么优化，数据和控制流最终的效果一致=>解决第一个问题
二、使用指令编号来提取内联函数的内部模板（use instruction numbering to extract the internal template of an inline function）=>给每个指令编号，便于比较和存储

判断两个函数一致的原理：如果f1能替换成f2使得功能不变，那么认为它们一致

0x01 EDG

定义一 Execution Dependence※※

Let R_i/W_i denote registers or memory that instruction v_i reads or writes, and D_i denote the instruction address of v_i.

Definition 1 (Execution Dependence). *Instruction v_2 has execution dependence on instruction v_1 if and only if one of the following holds true:*

1) *C1: v_1 and v_2 are both in the same basic block, $(W_1 \cap R_2) \cup (R_1 \cap W_2) \cup (W_1 \cap W_2) \neq \emptyset$ and $D_1 < D_2$.*

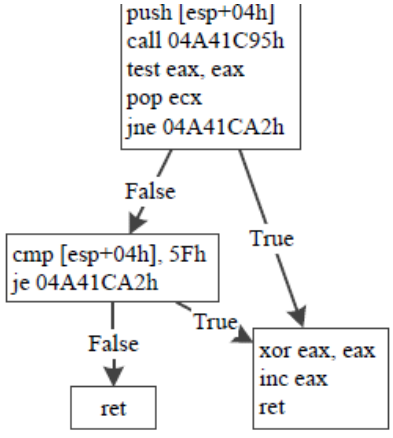
2) *C2: v_1 and v_2 are both in the same basic block, and v_2 is a control transfer instruction (CTI).*
3) *C3: v_1 and v_2 are in two basic blocks B_1 and B_2, respectively. v_1 can be executed last in block B_1, and v_2 can be execute first in block B_2. B_2 is a successor of B_1 in the control flow graph.*

定义二 Execution Dependence Graph

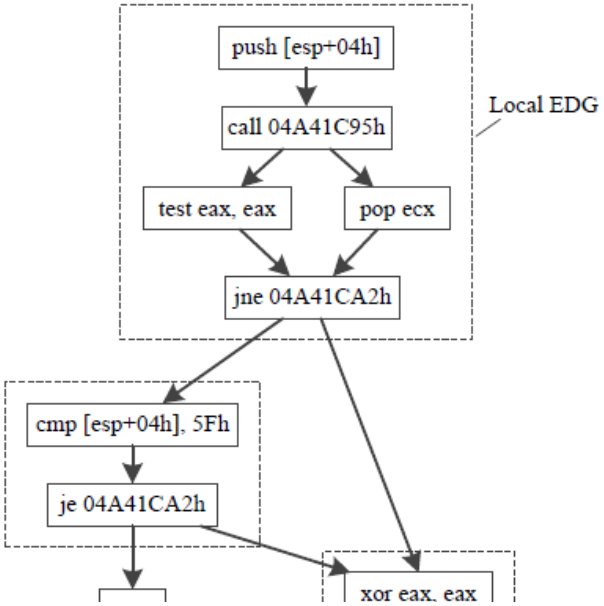
Definition 2 (Execution Dependence Graph)**.** *The execution dependence graph G of a function P is a 3-tuple element $G = (V, E, \delta)$, where*

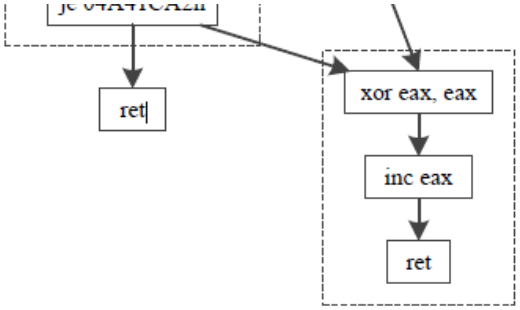
- *V is the set of instructions (i.e. nodes) in P.*
- *$E \subseteq V \times V$ is the set of execution dependence edges.*
- *$\delta : V \to I$ is a function mapping from an instruction to a 32-bit integer.*

The implementation details of δ are described in Section II-C.



(a) CFG





(b) EDG

总的来说：他就是基本块的依赖图和函数的CFG的混合表示

定义三 Graph Isomorphism（同构图）

Definition 3 (Graph Isomorphism)**.** *A bijective function* $f : V \to V'$ *is a graph isomorphism from a graph* $G = (V, E, \delta)$ *to a graph* $G' = (V', E', \delta')$ *if*

- $\forall e = (v_1, v_2) \in E, \exists e' = (f(v_1), f(v_2)) \in E'$ *such that* $\delta(v_1) = \delta'(f(v_1))$ *and* $\delta(v_2) = \delta'(f(v_2))$.

- $\forall e' = (v'_1, v'_2) \in E', \exists e = (f^{-1}(v'_1), f^{-1}(v'_2)) \in E$ *such that* $\delta'(v'_1) = \delta(f^{-1}(v'_1))$ *and* $\delta'(v'_2) = \delta(f^{-1}(v'_2))$.

Definition 4 (Subgraph Isomorphism)**.** *An injective function* $f : V \to V'$ *is a subgraph isomorphism from* G *to* G' *if there exists a subgraph* $S \subset G'$*, such that* f *is a graph isomorphism from* G *to* S.

对寄存器符号做标准化处理

1) **Register Normalization**. We use `reg1` to denote the first register in the operands of an instruction, and `reg2` to denote the second different register and so on, such as “mov eax, ecx” ⇒ “mov reg1, reg2” and “xor eax, eax” ⇒ “xor reg1, reg1”.
2) **Memory Normalization**. We use `M` to denote the memory accessing operand of an instruction, such as “mov eax, [ebx]” ⇒ “mov reg1, M”.

Second, the instruction after eliminating the diversity is represented as a two-tuple $SR = (m, r)$, where m represents the instruction mnemonic, and r represents the characteristic value of the operators. The digits in r, from high to low, denote the types of operands. For example, $r = 1$ for "op reg1", $r = 11$ for "op reg1, reg1", and $r = 21$ for "op reg1, reg2". All characteristic values for an instruction with 0, 1 or 2 operands are listed in Table I.

0x02 INSTRUCTION NUMBERING

TABLE II. EXAMPLES OF INSTRUCTION NUMBERING

Instruction	Normalization	$SR = (m, r)$	$Hash(m)$	ID
mov eax, [esp]	mov reg1, M	("mov", 41)	EBAC	41EBAC
push edi	push reg1	("push", 1)	C792	1C792
mov eax, ecx	mov reg1, reg2	("mov", 21)	EBAC	21EBAC

0x03 EDG construction※

Algorithm 1: Algorithm to construct a local EDG

Input: Basic block B
Output: Local EDG G

```
1  L = B.Length − 1;
   // Step 1: Create the initial graph
2  for i := 0 to L do
3      for j := i + 1 to L do
4          if (R_i ∩ W_j) ∪ (W_i ∩ R_j) ∩ (W_i ∩ W_j) ≠ ∅ then
5              G := G ∪ {v_i ↣ v_j};
   // Step 2: Simplify the graph
6  for i := 0 to L do
7      for j := i + 2 to L do
8          if {v_i ↣ v_j} ∈ G then
9              for k := i + 1 to j do //  i < k < j
10                 if {v_i ↣ v_k, v_k ↣ v_j} ∈ G then
11                     G := G\{v_i ↣ v_j};
```

第一步，基于之前讲的定义

j必须比i大

第二步，进行优化简化

如果vi->vk,vk->vj，那么一定有vi->vj，因此去掉vi->vj

0x04 LIBRARY FUNCTIONS IDENTFICATION算法

a node of S is defined as an internal node and the code corresponding to S is labeled as CS.（S的节点定义为内部节点，S对应的代码标记为CS。）CS contains all instructions of S, and both the first and last of CS are instructions of S

S is an inline function if and only if S can be outlined for all external nodes.（如果f1能替换成f2使得功能不变）
将在CS里和不在CS里的external node分成两半

A Identification Results Validation识别结果验证（识别为内联函数的验证）

一 external node不在CS里

case1 内部节点对外部节点具有C1依赖关系【外部节点在内部节点前】

external node的地址比Cs的start地址小，external node可以在任意的Cs指令之前执行，如果S被勾画出来，该BB块执行语义可以被表达

case2 内部节点对外部节点具有C2依赖关系

basic block的最后一条指令一定是internal node，是一条control flow transfer instruction
因此external node 地址比 Cs的起始地址小，external node一定在所有的cs指令前执行。如果S被勾画出来，该BB块执行语义可以被表达

case3 内部节点对外部节点具有C3依赖关系

如果内部节点没有前任，external node可以在所有的Cs指令之前执行，execution semanitic可以被提取出来的S表达；如果内部节点有前任，那么不是所有的CS指令都会在external node之前或之后执行，此时S不能表达执行语义

case4 外部节点对内部节点具有C1依赖关系

与case1相反，S被勾画出来也可以表达执行语义

case5 外部节点对内部节点具有C1依赖关系

与case2相反 S被勾画出来也可以表达执行语义

case6 外部节点对内部节点具有C3依赖关系

这种情况一般不存在，因为程序员写的函数都是结尾的，在结尾引入外部库不现实

二 external node在CS中

在这种情况里，external node地址都在cs中。但是external node不应该在CS中，因为如果包含了external node的一个block在cs中同时有前代和后代，**在某些执行路径中**，前面块中的指令应该在外部节点之前执行，或者后面块中的指令应该在外部节点之后执行，这样导致的后果是，S不能被提取来表达该段的执行语义

case7 内部节点对外部节点具有C1依赖关系【内部节点在外部节点前】

S can be outlined 只有在external node前面没有internal node执行才可以。因此该情况不可outlined

case8 内部节点对外部节点具有C2依赖关系

当没有内部节点在外部节点之前执行时，才可以outlined

case9 内部节点对外部节点具有C3依赖关系

case10 外部节点对内部节点具有C1依赖关系

case11 外部节点对内部节点具有C2依赖关系

case12 外部节点对内部节点具有C3依赖关系

三 验证识别结果

TABLE III. VALIDATION RESULTS OF ALL POSSIBLE DEPENDENCIES BETWEEN AN EXTERNAL NODE △ AND AN INTERNAL NODE ⊙. *OUT/IN* DENOTES THE EXTERNAL NODE IS OUT OF OR IN THE CORRESPONDING CODE OF S. *TRUE* AND *FALSE* DENOTE S CAN BE AND CANNOT BE OUTLINED, RESPECTIVELY. v_h IS A NODE WITHOUT PREDECESSORS IN S.

Cases	OUT	IN
$\triangle \rightarrowtail^{C1} \odot$ $\triangle \rightarrowtail^{C2} \odot$ $\odot \rightarrowtail^{C1} \triangle$ $\odot \rightarrowtail^{C3} \triangle$	TRUE	FALSE $\exists v \in S$ s.t. $v \rightarrowtail^{C1} \triangle$ TRUE Otherwise
$\triangle \rightarrowtail^{C3} \odot$	TRUE $\odot$ is v_h FALSE Otherwise	FALSE
$\odot \rightarrowtail^{C2} \triangle$	TRUE	Impossible

B 启发式过滤器降低算法复杂度

1) Instruction Number Filter(INF)

   Instructions of a function分为五类：①数据传输指令②算术指令③逻辑指令④字符串指令⑤其他指令

   当FT和FL的INF不同时，不可匹配

2) Basic Block Length Filter(BBLF)

   Block length不一致时不匹配

3) Basic Block Number Filter(BBNF)

   Basic Block Number不一致时不匹配

C library Identification Algorithm※

目标函数f_T和库函数F的识别算法

1) 反汇编f_T
2) 选择经过INF检测的F
3) 建立f_T的CFG
4) 选择经过BBNF检测的F，记为F_c
5) 对于每一个F_c中的library fucntion f_L

   a)如果f_L没有经过BBLF检测，continue

   b)建立f_T的EDG

   c)检测出all subgraphs of G_T与G_L同构的

   d)检查这些subgraphs是否可以outlined

   e)认为每个通过检查的subgraph是库函数

step 5(c)的检测算法为VF graph matching algorithm

首先应用INF，因为可以在不进行控制流分析的情况下执行INF。每个类别中的指令数和库函数的基本块数都在数据库中建立了索引，以便进行快速查询。BBNF很容易通过一个SQL查询执行，而BBLF则不是，因此BBNF是在BBLF之前执行的。例如，对于内联标识，SQL的条件是块数 <= C，其中C是目标函数的基本块数。

本文主要创新点：
EDG
Instruciton numbering

0x05 实验评估

TABLE IV. PROGRAMS FOR EVALUATION

Program	Version	Files	Compiler	Size/KB
7-Zip	9.2	9	VC7	3, 386
Httpd	2.0.64	10	VC6	1, 173
LAME	3.99.5	3	VC9	1, 409
Notepad++	6.3.2	1	VC8	1, 672
Putty	0.62	7	VC6	2, 525
Python	3.3.1	9	VC10	3, 596
WinMerge	2.14.0	4	VC9	2, 950
PHP	5.4.14	39	VC9	15, 794
WinXP	SP3	292	VC7	27, 558

样本获取：
8个开源应用程序，包括7-zip 9.20、 Apache HTTP Server 2.0.64、 Notepad++ 6.3.2、 WinMerge 2.14.0等等。我们用Microsoft Visual c++和/MT选项编译它们，/MT选项静态地链接到多线程C/c++库。包含调试信息的程序调试数据库(PDBs)也会为ground truth生成。

来自Windows XP SP3操作系统发行版的二进制文件。我们从微软符号服务器获得了他们的PDBs。这些二进制文件很可能是由VC7编译的(由ExeinfoPE[20]检测，它是一个封隔器检测工具)。

为了与目前只做完全识别的方法进行比较，我们选择了IDA Pro 6.4[21]作为比较。IDA Pro被广泛认为是最先进的工具。为了有效地标识库函数，我们的工具只处理链接到目标的库。这并不意味着我们的方法绑定到特定的编译器版本。事实上，我们总是可以尝试所有的库的评估，无论多么低效。为了在有限的时间内获得结果，为每个子图同构测试分配的时间最多为10秒。如果一对边的子图同构测试超时，则假定它不是同构的

我们使用精度、召回率和F-measure度量来评估算法的有效性。

表V为评价结果。INLINE表示内联库函数，FULL表示整个函数都是库函数的函数。内联函数的基本事实不在PDBs中，所以表v中没有列出内联函数的回忆。WinXP中的内联库函数的结果没有经过验证，因为在WinXP中有大量的可执行文件。表VI中列出了超时函数。

精度和查全率:表V表明，我们的工具在较高的精度下发现了更多的全库函数，查全率高于IDA，但7-Zip和winmerge的查全率略低于IDA。在Putty中只发现一个内联库函数的假阳性。代码库函数void cdecl forcdecpt l (char * Buf地区t地区)误认为strcpy()(图3),因为它未能通过规则:不能列出如果外部节点在CS和C1依赖内部节点(参见第三章案例7)。

其次，库函数的edg不完整，使得我们的工具无法获得准确的识别结果。在库函数识别过程中，当遇到间接跳转时，我们的工具将停止拆卸。这种策略可以防止生成过多的可能导致错误否定的汇编代码。但是，这种策略也会导致误报，因为两个不同的函数在间接跳转之前可能有相同的字节。类似地，有两种情况会导致我们的工具产生错误的底片。首先，目标函数的不完全边缘会导致假阴性。我们假设函数是一个同时具有起始地址和结束地址的指令序列。目标函数的范围是从IDA中读取的，不一定是正确的。通常，控制流在后面留下一个函数

TABLE V. RESULTS OF IDENTIFICATION

Program	Ours						IDA			
	INLINE		FULL				FULL			
	Found	Precision	True Positive	Precision	Recall	F-measure	True Positive	Precision	Recall	F-measure
7-Zip	40	1	**2,152**	0.75	**0.98**	0.85	1,843	**0.99**	0.84	**0.91**
Httpd	74	1	**1,555**	**1**	**0.94**	**0.97**	1,303	0.97	0.78	0.86
LAME	18	1	**1,160**	**1**	**0.94**	**0.97**	1,053	0.99	0.85	0.91
Notepad++	38	1	**416**	**0.99**	**0.77**	**0.87**	406	0.99	0.75	0.85
Putty	18	0.94	**1,678**	**1**	**0.96**	**0.98**	1,347	0.96	0.77	0.85
Python	24	1	**2,923**	**1**	**0.98**	**0.99**	1,562	0.97	0.78	0.86
WinMerge	9	1	**1,318**	0.98	**0.90**	**0.94**	647	**0.99**	0.74	0.85
PHP	173	1	**539**	1	**0.79**	**0.88**	517	0.91	0.75	0.82
WinXP	1,691	-	**1,816**	**0.99**	**0.36**	**0.53**	1,355	0.99	0.24	0.39

4)与IDA的比较:IDA使用函数的前32个字节作为函数的特性。当库函数的前32字节与目标函数的前32字节相同而其余字节不同时，IDA将目标函数错误地识别为库函数。例如，IDA在7-Zip.dll中将memmove()错误地识别为memcpy()。其他示例包括getc()和fgetc()、chkstk()和alloca probe()、wsopen s()和sopen s()等。我们的工具可以识别这些函数，因为它考虑了一个函数的所有字节。

最后还分析了一下IDA的缺陷所在

有三个原因使IDA产生了错误的否定。

①首先，一个库函数可能有不同的指令序列。在求值的修改后的二进制集合中，IDA没有识别出一些库函数，因为这些函数是用不同的前32字节修改的，如图1所示。指令规范化和EDG构造允许我们的方法识别出比IDA在语义上更等价的库函数变体。

②其次，一些库函数如adj fdivr m16i()不是由IDA导入的。在反汇编代码中，IDA将它们标记为未知的libname x，其中x是整数。其他例子包括crtMessageBoxA()、atodbl l()、safe fdiv()等。但是，这些函数在libcmt中。它们被导入到我们的工具。因此，我们的工具可以识别它们。

③最后，一些库函数(如可执行文件中的onexitinit())没有被IDA标识为函数。这种类型的函数不会被指令调用显式地调用，也没有对它的交叉引用。我们的工具也不能识别它们，因为目标函数的地址是从IDA导出的。此外，我们的工具可以识别PDBs中不存在的函数，以及IDA无法识别的函数，例如，位于0x1009533的ipv6.exe中的sbh alloc new region()。

0x06 不足与改进

我们的方法有几个限制。

首先，如果不同库函数的边是同构的，我们的方法不能区分它们。有两种方法可以提高精度:1)在库函数的EDG中，与指令调用相关的顶点可以作为属性附加重定位信息;

其次，一些编译器优化，例如公共子表达式消除，可能会消除目标代码中内联函数的指令。结果，目标函数中的内联函数是不完整的，可能无法识别。

第三，我们的方法的容量与它处理的库函数相关联。传统的基于模式匹配的技术也有这种局限性。我们使用了指令标准化技术来提取内联函数的内部模板，因此可以识别同一库函数的某些形式，如图1(b)中的库函数。但是，**一个库函数在不同的版本中可能有完全不同的字节**。向我们的工具提供一个库函数的所有可能版本几乎是不可能的，尽管可以很容易地提供已知编译器的这个库函数的所有可用版本。库函数在未知编译器或手工代码中的实现可能与在已知编译器中的实现完全不同。**这个问题的根源是，到目前为止，计算机还不能像人类一样理解函数代码的含义。**

最后，对内存访问指令的分析可以更加精确。一方面，避免对内存操作数进行别名分析会导致我们的工具产生假阴性。所有内存访问指令之间应该具有数据依赖关系，这是简单而粗糙的。在某些情况下，这个假设失败了。例如，对于两个相邻的指令A: mov [ecx]， eax和B: inc [ecx+4]，它们是相互独立的。**如果目标函数中的顺序是A B，而库函数中的顺序是B A，则我们的工具无法识别目标函数**。为了构建更精确的edg，应该引入指针别名分析。二进制码的串码分析是一项困难的工作，其讨论超出了本文的范围。另一方面，在对指令编号的过程中，对内存的抽象会导致误报。当前内存操作数的抽象丢失了操作数的细节，因此一些指令不能通过它们的id彼此区分。例如，指令mov eax， [eax+4]与指令mov eax， [eax]具有相同的ID。这可能会导致误报。

期望:

1.更好的numbering技术

2.执行更精确和广泛的依赖分析，如对函数调用的指针-别名分析和过程间分析。

0x07近期研究进展

A Library Functions Identification

①著名的库识别技术是IDA FLIRT[3]，它使用字节模式匹配算法来确定目标函数是否匹配IDA已知的任何签名。

②Dcc[2]类似于FLIRT。

③Hancock[4]扩展了调情技术，使用库函数引用启发法，假设一个函数是库函数(如果该函数在库函数中被静态调用)。这种方法的主要缺点是只使用函数的前n个字节作为匹配模式，增加n很容易提高精度，但不能保证包含所有库函数。根据包装器函数与系统调用接口的交互，UNSTRIP[5]标识二进制代码中的包装器函数。但是，这种方法仅限于包装器函数。库函数可能没有调用指令。因此，这种方法不适用于识别这些库函数。

④GraphSlick是一个自动检测内联函数的IDA插件。它从CFG构造开始，并在一个程序[22]中找到类似的子图。但是，它更像是一个代码克隆检测工具，不能提供已识别的内联函数的函数名。

B code mining

库标识是一个代码挖掘问题，涉及到代码克隆检测、软件剽窃和恶意软件检测。

①源代码层面

源的水平。在源级，函数可以用程序依赖图(PDG)来表示，而克隆可以通过子图同构检测[10]、[23]来识别。GPLAG是一个基于pdgf的剽窃检测工具，通过挖掘PDGs[24]来检测剽窃。在GPLAG中，剽窃检测采用了图同构的方法，并提出了一种统计有损滤波的方法来剔除剽窃搜索空间。

基于PDG的方法可能不适用于内联库标识，原因有二。首先，用于构建PDG的数据流分析对于大型函数来说非常耗时。其次，对于一个函数来说，PDG的结构要比EDG复杂得多。子图同构是一个NPC问题[17]。在可接受的时间内完成PDG子图同构检测可能是不可能的。因此，GPLAG的实现没有考虑到控制依赖关系对效率的影响。

②二进制代码层面

Sæbjørnsen等人提出了一个可扩展的算法检测代码克隆在二进制可执行文件[25]。该算法扩展了现有的基于标记树特征向量聚类的树相似度框架。Bruschi等人提出了一种检测程序内部变质恶意代码的策略，该策略基于程序的CFG与已知恶意软件的CFG集的比较。Kruegel等人提出了一种基于蠕虫s CFG分析的蠕虫检测技术，并介绍了一种原始图着色技术，该技术支持对蠕虫s结构的更精确描述。这些算法采用预处理规范化和比较的方法。它们可用于完全标识，但不适用于内联标识。关键原因是他们没有考虑块指令之间的数据依赖性，这样就无法识别具有不连续字节的库函数。

0x08 总结

本文提出了一种通过建立和分析执行相关图来识别二进制代码中**全库函数和内联库函数**的新方法。将库函数识别转化为EDG子图的同构识别。样机实验表明，该方法是可行的。最后，在几种流行的软件上进行了实验，验证了该方法在全内联库识别中的有效性。该工作的主要贡献包括:(1)为识别而引入的执行依赖图(EDGs);(2)一个统一的标识框架，标识完整的和内联的库函数和(3)一个完全工作的原型工具。