DOI: 10.3785/j.issn.1008-973X.2019.05.002

基于混合分析的二进制程序控制流图构建方法

朱凯龙，陆余良，黄晖，邓兆琨，邓一杰
(国防科技大学 电子对抗学院，安徽 合肥 230000)

摘 要：构建控制流图(CFG)是二进制程序分析的基础工作，针对静态构建方法无法处理间接跳转，动态构建方法效率低、不适用于大规模程序的问题，提出结合静态分析和动态分析的混合分析方法. 使用静态分析获得基础的控制流信息；采用模糊测试生成测试用例以进行动态分析，利用动态插桩获得间接跳转信息；融合静态分析和动态分析结果生成控制流图. 基于该混合分析方法，设计并实现了面向 x86 平台二进制程序的控制流图构建工具 CFGConstructor. 分别在示例程序和 CGC 数据集上进行实验，评估该工具的有效性和性能. 实验结果表明 CFGConstructor 相比于静态分析能够构建更加完备的控制流图，相比于动态分析分析效率更高，能够适用于大规模程序.

关键词：二进制程序分析；控制流图(CFG)；混合分析技术；模糊测试；动态二进制插桩
中图分类号：TP 311　　**文献标志码：**A　　**文章编号：**1008−973X(2019)05−0829−08

Construction approach for control flow graph from binaries using hybrid analysis

ZHU Kai-long, LU YU-liang, HUANG Hui, DENG Zhao-kun, DENG Yi-jie
(*Electronic Countermeasure College*, *National University of Defense Technology*, *Hefei* 230000, *China*)

Abstract: The construction of control flow graph (CFG) was the basis of binary program analysis. A hybrid analysis approach combining static and dynamic analysis techniques was proposed, for the problems that the static construction method cannot handle the indirect jump cases and dynamic construction methods were inefficient and not suitable for large-scale programs. The static analysis technique was used to obtain the basic control flow of the target program. Test cases generated by fuzz testing were used to dynamically analyze the target program, during which a dynamic binary instrumentation technique was used to obtain information of indirect jumps. Finally, the analysis results in the former two steps were integrated to generate CFGs. A CFG construction system CFGConstructor targeting on x86 binaries was designed and implemented based on the proposed hybrid analysis method. Experiments were carried out on the sample programs and CGC dataset to evaluate the effectiveness and efficiency. Results show that the proposed approach can construct more complete CFGs than static analysis do, and is more efficient than dynamic analysis, capable to analyze large programs.

Key words: binary analysis; control flow graph (CFG); hybrid analysis technology; fuzz testing; dynamic binary instrumentation

软件安全性分析面临大量非开源二进制程序，而面向二进制的程序分析不依赖程序的源代码，能够发现编译阶段引入的程序漏洞，具有重要的实用价值. 控制流图是二进制程序分析的基础数据，许多程序分析技术，如数据流分析[1-2]、污点分析[3]、符号执行技术[4]和二进制代码比对技术[5]

收稿日期：2018−04−19.　　**网址：**www.zjujournals.com/eng/fileup/HTML/201905002.htm
基金项目：国家重点研发计划“网络空间安全”重点专项资助项目(2017YFB0802900).
作者简介：朱凯龙(1991—)，男，博士生，从事程序分析研究. orcid.org/0000-0001-5241-0157. E-mail: 471801698@qq.com
通信联系人：陆余良，男，教授. orcid.org/0000-0002-1712-4224. E-mail: 13329018500@189.cn

等都需要在控制流图的基础上进行. 通过对程序控制流图进行检查和分析, 能够达到程序验证[6-8]、自动化生成测试用例[9]和发现软件漏洞[10-11]等目的. 因此, 面向二进制程序, 构建完备、精确的控制流图具有重要意义.

构建二进制程序控制流图的方法可以分为静态分析、动态分析和混合分析方法.静态分析方法[12-15]不需要动态执行程序, 通过反汇编获得程序的汇编代码, 对汇编指令进行解析构建程序控制流图. 静态分析的典型工具包括 IDA Pro[12], BINCOA[13], Jakstab[14-15], 能分析整个二进制程序的汇编指令, 具有较高的代码覆盖率, 但静态分析无法获得间接跳转地址, 所以利用该方法构建的控制流图不完备. 动态分析方法[16]利用测试用例集具体执行程序, 通过记录程序执行的控制流信息, 构建程序控制流图.该方法精确度高,并且可以获得间接调用信息, 但是路径覆盖率较低. 为了提高动态分析的路径覆盖率, FEX[16]基于强制执行技术构建控制流图, 通过动态跟踪程序执行, 记录程序每个分支及对应状态, 采用状态保存及恢复方法强制覆盖分支路径. FEX 有效提高了动态分析的路径覆盖, 但是状态保存及恢复消耗大量资源, 导致该方法效率较低, 不适用于大型程序分析. 混合分析方法[17-18]综合利用静态和动态分析的结果构建控制流图. 文献[17]、[18]采用混合分析方法分别基于汇编代码和中间代码构建二进制程序的控制流图. 混合方法均采用静态符号执行技术生成动态分析所需的测试用例, 提高了路径覆盖率, 但在面对大规模程序时该方法存在路径爆炸的问题, 并且依赖求解器[19]能力, 约束求解复杂, 开销大、效率低, 所以该方法不适用于构建大规模程序的控制流图.

综合考虑静态和动态分析的特点, 提出基于混合分析的二进制程序控制流图构建方法. 讨论构建二进制程序控制流图所面临的处理间接跳转的难题; 介绍基于混合分析构建二进制程序控制流图的过程. 为了避免静态符号执行和强制执行等高开销技术, 选择使用模糊测试技术生成测试用例, 通过动态二进制插桩技术获得间接跳转地址, 融合静态和动态分析结果生成控制流图, 设计并实现了控制流图构建工具 CFGConstructor. 通过实验验证所提出方法的有效性和效率.

1 间接跳转分析

C/C++高级语言中的虚函数调用、函数指针调用等特性, 经编译后, 在二进制程序层就表现为间接函数跳转[20]. 间接跳转地址是在运行时决定的, 很难通过静态分析获得这些函数的跳转关系[21], 从而无法获得完备的控制流图, 导致基于该控制流图的安全分析不能发现某些潜在漏洞.

1.1 虚函数调用

虚函数是 C++实现多态的方法, 虚函数调用会在程序中引起间接跳转. 如图 1 所示为手工构造的虚函数引起间接跳转的示例程序 virtual_function 的源代码和汇编代码.

在示例程序中定义了基类 Base 及其派生类 Derive, 在基类和派生类中都定义了虚函数 foo, 并且派生类中的 foo 中存在安全缺陷 BUG. 基类指针 *p* 指向派生类对象 derive, 在通过指针 *p* 调用函数 foo 时, 将动态地从 derive 的虚函数表中定位被调函数 foo, 并调用派生类的 foo 函数. 可以看出指针 *p* 调用 foo 经过编译表现为间接调用 call rax, 通过静态分析构建控制流图很难获得间接调用 rax 中的函数地址, 无法发现该间接跳转关

```
class Base
{
  public:
    virtual int foo (int a)
    {
      return a;
    }
};
class Derive: public Base
{
  public:
    virtual int foo (int a)
    {
    BUG (a); //Here is a bug
      return a;
    }
};
intfunc (int a)
{
  Base *p;
  Derive derive;
  p = &derive;
  return p->foo (a);
}
```

(a) 源代码

```
// func
0x004006C8:learax,[rbp−10h]
0x004006CC:movrdi,rax
0x004006CF:call0x00040075A
0x004006D4:learax,[rbp−10h]
0x004006D8:mov[rbp−8h],rax
0x004006DC:movrax,[rbp−8h]
0x004006E0:movrax,[rax]
0x004006E3:movrax,[rax]
0x004006E6:movecx,[rbp−14h]
0x004006E9:movrdx,[rbp−8h]
0x004006ED:movesi,ecx
0x004006EF:movrdi,rdx
0x004006F2:callrax
0x004006F4:...
//Derive:foo
0x00400730:pushrbp
0x00400731:movrbp,rsp
0x00400734:mov[rbp−8h],rdi
0x00400738:mov[rbp−Ch],esi
...//Hereisabug
0x0040074B:moveax,[rbp−Ch]
0x0040074E:addeax,2
0x00400751:poprbp
0x00400752:retn
```

(b) 汇编代码

图 1 示例程序 virtual_function 的源代码和汇编代码

Fig.1 Source code and assembly code of sample program virtual_function

系. 如果对该程序进行静态的污点分析, 设 func 中 a 为污点源, BUG(a)为 sink 点, 通过基于静态分析构建的控制流图无法获得从 func 到 Derive::foo 的调用关系, 则污点信息不能传播到函数 Derive::foo 中, 导致不能发现 Derive::foo 中存在的 bug.

1.2　函数指针调用

函数指针调用是 C/C++常用的函数调用方法, 使用函数指针调用也会引起间接跳转. 如图 2 所示为函数指针调用的示例程序 function_pointer 的源代码和汇编代码. 根据输入变量 a、b 大小的不同, 函数指针 foo 可能指向 add 或者 sub, 调用 foo 经编译后表现为间接跳转 call eax, 跳转地址在动态执行时决定. 通过静态分析很难获得该间接跳转关系, 不能构建完备的控制流图, 可能导致基于该控制流图的安全分析无法发现被调函数中存在的漏洞.

2　基于混合分析构建控制流图

为了构建更加完备的控制流图, 提出基于混合分析的二进制程序控制流图构建方法, 并实现了控制流图构建工具 CFGConstructor, 工具架构如图 3 所示. CFGConstructor 主要包括 2 个组件: 静态分析组件和动态分析组件. 静态分析组件的主要功能是对二进制程序进行反汇编生成汇编代码, 并通过静态分析方法生成基本的控制流图. 动态分析组件的主要功能是利用测试用例集引导二进制程序具体执行, 通过插桩间接跳转指令确定跳转地址, 完善静态分析构建的控制流图.

CFGConstructor 构建控制流图的主要步骤如下: 1)二进制程序经过静态分析组件反汇编生成汇编代码, 并通过基本块划分、连接基本块生成基本控制流图; 2)采用基于遗传算法的模糊测试技术生成测试用例集, 引导二进制程序具体执行, 利用动态插桩方法监控间接跳转指令, 确定跳转地址; 3)融合静态分析和动态分析结果, 将间接跳转的目标基本块添加到控制流图中, 生成最终的控制流图.

2.1　静态分析

静态分析构建控制流图包括 3 个步骤: 反汇

```
int sub (int a, int b)
{
  return a−b;
}
int add (int a, int b)
{
  return a + b;
}
intfunc (int a, int b)
{
  int (*foo)(int, int);
  if (a>b)
  {
    foo = sub;
  }
  else
  {
    foo = add;
  }
  return foo (a, b);
}
```

(a) 源代码

```
0x0004844D:pushebp
0x0004844E:movebp,esp
0x00048450:moveax,[ebp+arg_4]
0x00048453:movedx,[ebp+arg_0]
0x00048456:subedx,eax
0x00048458:moveax,edx
0x0004845A:popebp
0x0004845B:retn
0x0004845C:pushebp
0x0004845D:movebp,esp
0x0004845F:moveax,[ebp+arg_4]
0x00048462:movedx,[ebp+arg_0]
0x00048465:addeax,edx
0x00048467:popebp
0x00048468:retn
...
0x0004846F:moveax,[ebp+8]
0x00048472:cmpeax,[ebp+12]
0x00048475:jle0x0048480
0x00048477:mov[ebp−12],0x4844D
0x0004847E:jmp0x0048487
0x00048480:mov[ebp−12],0x4845C
0x00048487:moveax,[ebp+12]
0x0004848A:mov[esp+4],eax
0x0004848E:moveax,[ebp+8]
0x00048491:mov[esp],eax
0x00048494:moveax,[ebp−12]
0x00048497:call eax
```

(b) 汇编代码

图 2　示例程序 function_pointer 的源代码和汇编代码

Fig.2　Source code and assembly code of sample program function_pointer

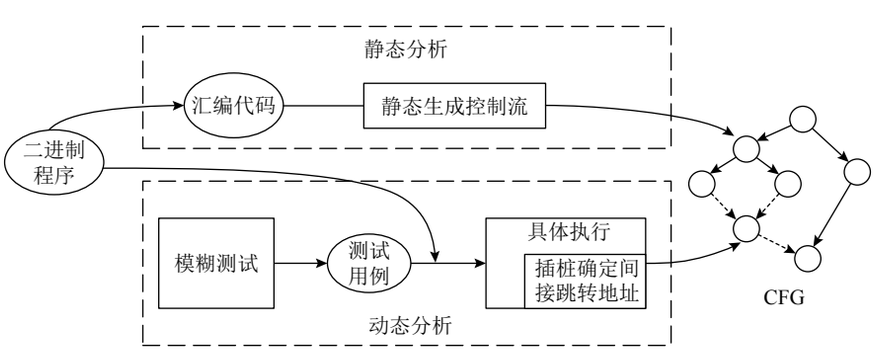

图 3　控制流图构建工具 CFGConstructor 架构

Fig.3　Architecture of CFG construction tool CFGConstructor

编二进制代码生成汇编代码；根据汇编代码划分基本块；根据跳转关系连接基本块.

1）二进制代码静态反汇编. 对二进制代码的静态反汇编算法包括线性遍历算法和递归遍历算法. 线性遍历算法是指从程序入口开始顺序遍历程序二进制代码，反汇编每条指令，当遇到非法指令则停止. 该方法不能区分代码和数据，反汇编结果不准确，且会造成后续反汇编错误. 本研究选择递归遍历算法进行反汇编，该方法使用控制流指令引导反汇编过程，从程序入口开始进行反汇编，当遇到非跳转类指令时，则顺序进行反汇编；当遇到改变控制流的跳转类指令时，则跳转到目标地址继续进行反汇编. 该方法以控制流为指导，可以识别并跳过代码中嵌套的数据部分，反汇编结果较准确.

2）划分基本块. 基本块[22]是组成控制流图的原子节点，每个基本块包含一组顺序执行的语句序列，有且仅有 1 个入口指令和出口指令. 控制流只能从入口指令进入基本块，从出口指令离开基本块，不会在中间结束或者跳转.

划分基本块首先需要确定每个基本块的入口指令. 确定基本块入口指令的规则如下：a. 每个函数入口的第 1 条指令；b. 所有非间接跳转指令的目标指令；c. 在汇编代码顺序中，紧跟在跳转指令后的指令.

通过静态分析无法获得间接跳转指令的目标指令，所以不使用间接跳转指令划分基本块. 为了构建函数间的控制流图，认为 call 指令属于无条件的跳转指令. 利用入口指令将汇编代码划分成为基本块，每个入口指令对应的基本块包括从它开始，到下一个入口指令或程序结束指令之间的所有指令.

3）连接基本块. 控制流图中的边表示基本块之间可能存在的跳转关系，当基本块 block_i 和 block_j 之间满足以下任一关系，则添加 1 条从 block_i 到 block_j 的边：a. 有从 block_i 结尾跳转到到 block_j 开头的跳转指令；b. 在原汇编代码顺序中，block_j 紧跟在 block_i 之后，且 block_i 不存在无条件跳转语句.

2.2 动态分析

针对静态分析不能处理间接跳转的问题，采用动态分析确定间接跳转地址. 使用模糊测试技术生成测试用例集，然后具体执行目标程序，采用动态二进制插桩获得间接跳转关系.

1）生成测试用例集. 文献[17]、[18]采用静态符号执行生成测试用例进行动态分析，该方法依赖于求解器的求解能力，对于大型程序存在约束复杂，求解耗时长甚至不能求解的问题. 为了能够适用于大型程序，避免使用静态符号执行等高开销的分析方法，采用模糊测试技术生成测试用例集.

CFGConstructor 选择当前最流行的模糊测试工具 AFL[23]生成测试用例. AFL 是低开销的模糊测试工具，以路径覆盖率为导向，采用遗传算法生成测试用例. AFL 通过轻量级插桩监控程序执行状态，检查输入是否能发现新的路径，并反馈评估输入的优劣，保留任何可以发现新路径的输入，并且将其进一步变异生成下一代输入. CFGConstructor 使用人工生成的方法构造模糊测试的种子输入，采用 AFL 中以路径覆盖为导向的变异策略生成测试用例. 为了避免在后续插桩分析时重复执行具有相同路径的测试用例，CFGConstructor 仅收集能够发现新路径的输入，并将其加入到测试用例集中.

2）获得间接跳转地址. 利用上述生成的测试用例集引导目标程序具体执行，并监控程序执行状态，确定间接跳转地址. CFGConstructor 采用插桩工具 Pin 对程序执行过程进行监控，对每个间接跳转指令进行插桩，在动态执行间接跳转时，记录跳转的目的地址，从而获取间接跳转关系. 由于间接跳转指令的数量在整个程序中所占比例较低，CFGConstructor 仅需要进行少量插桩，不会引起大量的额外开销，保证 CFGConstructor 可以对大型程序进行分析.

2.3 控制流图融合

CFGConstructor 融合静态分析和动态分析结果，将动态识别的间接跳转关系添加到静态获得的控制流图中，获得更加完备的控制流图. 静态分析和动态分析结果的融合过程如算法 1 所示. 算法遍历控制流图中所有基本块中的每条指令，对存在间接跳转指令的基本块进行处理，将间接跳转关系添加到控制流图中.

算法 1：融合静态分析、动态分析结果

输入： 静态分析获得的控制流图 staticCFG、动态分析获得的间接跳转地址

输出： 包含间接跳转的控制流图 CFG

```
1 copy staticCFG to CFG.
2 for bb in CFG.basicblocks:
3     for ins in bb.instructions:
```

```
4       if ins is INDIRECT_BRANCH:
5           tbs = getTargetBlock(ins). //可能存在多个目标基本块
6           if ins is not the end of bb:
7               (b1, b2) = split(bb, ins).
8               add block b1, b2 to CFG.
9               connect all bb.predecessors to b1.
10              for tb in tbs:
11                  connect b1 to tb.
12                  connect tb to b2.
13              connect b2 to all bb.successors.
14              delete bb.
15          else
16              disconnect bb to all bb.successors.
17              for tb in tbs:
18                  connect bb to tb.
19                  connect tb to all bb.successors.
20 return CFG
```

如图 4 所示，根据间接跳转指令在基本块中位置的不同，添加间接跳转关系存在 2 种情况. 情况 1 如图 4(a)所示，如果间接跳转指令不是基本块的最后一条指令，根据该间接跳转指令将原基本块 block 分割成为 2 个新的基本块 block1、block2；分别将原基本块的入边和出边重定向到 blcok1、block2；根据动态分析获得的间接跳转关系确定该间接跳转指令的目标基本块，并添加 block1 到目标基本块、添加目标基本块到 block2 的边. 情况 2 如图 4(b)所示，如果间接跳转指令是基本块的最后一条指令，则不需要分割基本块，添加 block 到目标基本块的边，并将 block 所有出边的起点重定向到目标基本块.

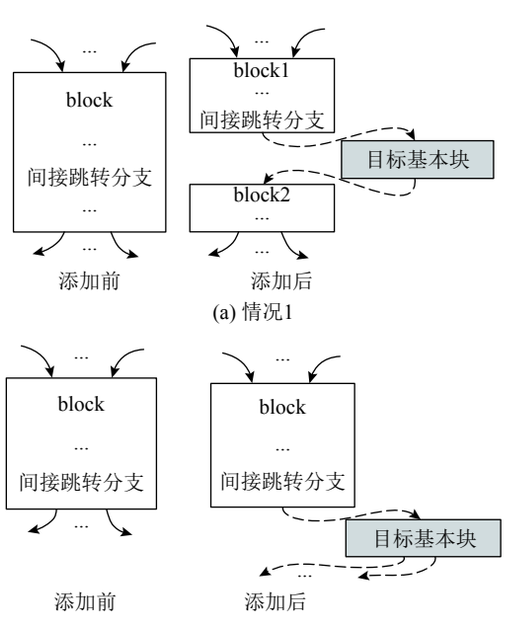

图 4　控制流图中的间接跳转添加

Fig.4　Addition of indirect jumps to CFG

如算法 1 第 5 行所示，间接跳转指令可能存在多个目标基本块. 如算法第 10、18 行所示，当存在多个目标基本块时，分别添加每个基本块为间接跳转指令的后继基本块，表示该跳转指令存在多个候选执行分支.

3　实验及分析

3.1　实验设置

设计实验评估 CFGConstructor 的有效性和性能. 实验平台为 Ubuntu 14.04LTS 32 位操作系统，处理器为 Intel Core i7-4702HQ @2.2 GHz, 内存为 DDR3 4 GB. 为了验证工具的有效性，与静态分析工具 IDA Pro 进行对比. 首先对 2 个手工构造的示例程序进行分析；然后对美国国防部高级研究计划局(DARPA)举办的网络安全挑战赛(CGC)[24]中部分测试集进行测试.

3.2　有效性验证

3.2.1　手工构造的示例程序　作为反汇编工具，IDA 没有直接提供函数间的控制流图，为了与 IDA 进行比较，使用 IDAPython 在 IDA Pro 6.8 中实现插件 IDA CFG. 该插件基于 IDA 反汇编结果构建函数间的控制流图，图中的边依赖于 IDA 识别的跳转关系. 分别使用 IDA CFG 和 CFGConstructor 对 2.1 节中构造的 2 个程序进行分析，构造程序控制流图如图 5 所示. 为了便于说明，图中仅显示包含间接调用的关键部分，其中矩形表示基本块，实线表示基本块间的直接跳转关系，是 IDA CFG 和 CFGConstructor 构造控制流图的公共部分；虚线表示间接跳转关系，仅包含在 CFGConstructor 构造的控制流图中.

如图 5(a)所示为实例程序 virtual_function 的控制流图，函数 func 的基本块 0x4006D4 中包含间接跳转指令 call rax. 静态分析工具 IDA 无法发现 rax 的具体值，不能判断该指令跳转的目标地址. CFGConstructor 通过动态二进制插桩该指令，获得具体执行时的 eax=0x400730，说明程序将跳转执行 Derive::foo 函数，从而发现基本块 0x4006D4 到 0x400730 的跳转关系，以及当函数返回时从基本

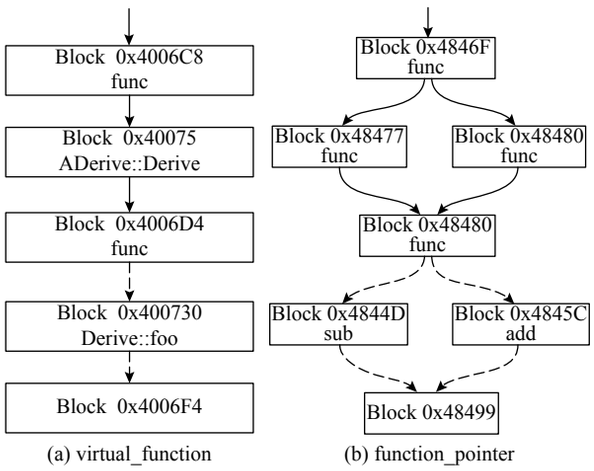

图 5 IDA、CFGConstructor 构建示例程序的控制流图

Fig.5 CFGs of sample programs constructing by IDA and CFGConstructor

块 0x400730 到 0x4006F4 的跳转关系.

如图 5(b)所示为实例程序 function_pointer 的控制流图，函数 func 的基本块 0x48480 中包含间接跳转指令 call eax. 与示例 virtual_function 不同的是，该跳转目标地址 eax 在实际执行中存在多个可能取值，CFGConstructor 通过模糊测试生成的测试用例集覆盖多个路径分支，从而发现这些取值. 当测试用例中 $a>b$ 时，eax=0x4844D，程序跳转到 sub 函数执行；当测试用例中 $b<a$ 时，eax=0x4845C，程序跳转到 add 函数执行. 因此，能够发现从基本块 0x48480 分别到 0x4844D、0x4845C 的跳转关系.

实验结果表明，基于静态分析的 IDA 仅能处理直接跳转关系而不能识别间接跳转关系，从而不能获得某些基本块的可达路径，导致无法发现一些潜在漏洞. 基于混合分析的 CFGConstructor 不仅能发现直接跳转关系，而且能够识别间接跳转关系，并且可以处理多目标地址间接跳转的情况，该工具能够针对二进制程序构建更加完备的控制流图.

在二进制程序分析中，如污点分析、符号执行生成测试用例等，通常更关心“从程序入口可达”的代码. 统计 IDA、CFGConstructor 构建控制流图中可达代码情况，结果如表 1 所示. 其中，F_i、V_i、E_i 分别为在 IDA 构建的控制流图中，从程序入口可达的函数集合、基本块集合和连接集合；F_c、V_c、E_c 分别为在 CFGConstructor 构建的控制流图中，从程序入口可达的函数集合、基本块集合和连接集合；I 为 CFGConstructor 识别间接跳转的数量. 可以看出，相较于 IDA，CFGConstructor 通过识别间接调用关系，能够发现一些静态分析不能到达的函数，进而发现更多的可达基本块. 值得注意的是，在构造 2 个示例程序时，分别人工引入了 1 个和 2 个间接跳转，而 CFGConstructor 则分别发现了 2 个和 3 个间接跳转，额外发现了更多的跳转关系. 这是因为在对源代码进行编译时，为了生成完整的可执行文件，编译器会自动添加一些额外代码，在这个过程中也会引入间接跳转，CFGConstructor 能够发现这部分间接跳转关系. 综上所述，CFGConstructor 能够有效识别间接跳转，发现更多可达基本块，构建更加完备的控制流图.

表 1 IDA、CFGConstructor 对示例程序的分析结果

Tab.1 Analyzing results on sample programs by IDA and CFGConstructor

二进制程序	$\|F_i\|$	$\|V_i\|$	$\|E_i\|$	$\|F_c\|$	$\|V_c\|$	$\|E_c\|$	I
virtual_function	5	9	8	8	24	27	2
function_pointer	4	11	12	8	27	33	3

3.2.2 CGC 测试集 为了进一步评估 CFGConstructor 构建程序控制流图的性能，实验从 CGC 标准测试集中选择了 6 个不同大小的二进制程序进行实验. 分别使用 IDA、CFGConstructor 对所有程序构建控制流图，分析结果如表 2 所示. 其中，S 为二进制程序大小，F_c-F_i 为 CFGConstructor 能够发现但 IDA 不能发现的可达函数集合，类似地，V_c-V_i、E_c-E_i 分别为 CFGConstructor 能够发现但 IDA 不能发现的可达基本块和连接的集合. 可以看出，在二进制程序中普遍存在间接跳转关系，而基于静态分析的 IDA 无法发现通过间接跳转到达的基本块. CFGConstructor 可以识别间接跳转关系，发现间接跳转的目标基本块，并将间接跳转目标基本块及其后继基本块均标记为可达基本块，从而发现更多的程序入口可达代码. 在测试集中，CFGConstructor 平均能够比 IDA 多发现 12.62% 的可达函数、24.72% 的可达基本块和 33.38% 的连接. 结果表明，相较于静态分析方法，基于混合分析的 CFGConstructor 能够构建更加完备的控制流图.

3.3 分析效率

对 CFGConstructor 的分析效率进行评估，利用 CFGConstructor 对不同大小的程序进行分析，记录分析所消耗时间. 相较于动态分析，静态分析时间较短，所以实验不考虑静态分析的时间开销. 实验使用 AFL 对所有测试程序分别进行 2 h 的分析，生成测试用例集合，利用 AFL 生成的测

表 2　IDA、CFGConstructor 对 CGC 测试集的分析结果

Tab.2　Analyzing results on CGC dataset by IDA and CFGConstructor

二进制程序	S/KB	$\lvert F_i\rvert$	$\lvert V_i\rvert$	$\lvert E_i\rvert$	$\lvert F_c\rvert$	$\lvert V_c\rvert$	$\lvert E_c\rvert$	I	$\frac{\lvert F_c-F_i\rvert}{\lvert F_i\rvert}$/%	$\frac{\lvert V_c-V_i\rvert}{\lvert V_i\rvert}$/%	$\frac{\lvert E_c-E_i\rvert}{\lvert E_i\rvert}$/%
pwns	6	17	99	113	19	112	150	8	11.76	13.13	32.74
fsb	8	9	19	20	11	32	38	1	22.22	68.42	90.00
silver_bullet	22	18	102	148	20	115	166	3	11.11	12.75	12.16
ret2shellcode	86	40	72	86	64	102	135	2	35.00	16.67	30.23
pwn250	716	362	14 540	23 038	394	18 742	29 526	370	8.84	28.90	28.16
pwn300	727	453	19 193	28 473	495	20 015	30 461	40	9.27	4.28	6.98
平均值	–	–	–	–	–	–	–	–	12.62	24.72	33.38

试用例集对每个测试程序进行动态分析，重复 3 次并记录分析时间，结果如表 3 所示. 其中，t 为动态分析所需时间；T 为生成测试用例的数量，由于程序路径复杂性不同，生成的测试用例数目存在差异. 可以看出，为了能够覆盖更多的路径，针对越复杂的目标程序，AFL 生成的测试用例越多. CFGConstructor 仅保留 AFL 中能够发现新路径的测试用例，避免盲目执行大量测试用例.

由表 3 可以看出，CFGConstructor 能够在较短时间内完成对目标程序的分析，例如，大小为 727 KB 的目标程序 pwn300 中包含 453 个可达函数、19 193 个可达基本块和 28 473 条连接，利用 CFGConstructor 构建该程序控制流图需执行 321 个测试用例，时间开销仅为 82.42 s. 如果采用文献[16]提出的强制执行方法对该程序进行分析，每个路径分支点都需要保存、恢复虚拟机快照，将消耗大量时间；如采用文献[17]、[18]中基于静态符号执行方法生成的测试用例集，将面临路径爆炸、约束困难、开销大等问题，不能满足实际分析需求. 综上所述，相比于其他方法，基于混合分析的 CFGConstructor 消耗较少时间就能构建程序的控制流图，具有较高的实用价值.

表 3　CFGConstructor 动态分析的时间开销

Tab.3　Timecost of CFGConstructor dynamic analysis

二进制程序	S/KB	T	t/s			
			第 1 次	第 2 次	第 3 次	平均值
pwns	6	104	56.12	56.86	58.72	57.23
fsb	8	2	0.70	0.72	0.71	0.71
silver-bullet	22	93	38.48	38.38	38.61	38.49
ret2she-llcode	86	80	28.58	29.15	29.01	28.91
pwn250	716	45	11.11	11.18	11.27	11.19
pwn300	727	321	83.09	82.22	81.95	82.42

4　结　论

提出动静结合的二进制程序控制流构建方法，使用动态分析识别程序间接跳转关系，结合静态分析结果生成较完备的控制流图，并实现工具 CFGConstructor. 实验表明，相比于静态分析，该方法构造的控制流图更加完备，在测试集上平均能多发现 24.72% 的可达基本块和 33.38% 的跳转关系. 该方法避免使用强制执行、静态符号执行等高开销分析方法，提高了针对大规模程序的分析能力. 研究成果为进一步开展二进制程序分析，发现潜在漏洞提供了基础支撑.

所提出的方法仍存在局限性，其发现间接跳转的能力依赖于 AFL 生成测试用例集的质量，如果某些间接跳转指令没有被测试用例覆盖，则该方法无法获得该间接跳转关系. 在未来工作中，将以覆盖间接跳转为导向改进 AFL，生成更加适用于 CFGConstructor 的测试用例集，构建更加完备的二进制程序控制流图.

参考文献 (References):

[1] HENDERSON A, YAN L, HU X, et al. DECAF: a platform-neutral whole-system dynamic binary analysis platform [J]. **IEEE Transactions on Software Engineering**, 2017, 43(2): 164–184.

[2] 万志远, 周波. 基于静态信息流跟踪的输入验证漏洞检测方法[J]. 浙江大学学报: 工学版, 2015, 49(4): 683–691. WAN Zhi-yuan, ZHOU Bo. Static information flow

tracking based approach to detect input validation vulnerabilities [J]. **Journal of Zhejiang University: Engineering Science**, 2015, 49(4): 683–691.

[3] NEWSOME J, SONG D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [C]// **The 12th Annual Network and Distributed System Security Symposium**. San Diego: The Internet Society, 2005: 253–260.

[4] ZHANG B, FENG C, WU B, et al. Detecting integer overflow in Windows binary executables based on symbolic execution [C]// **17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing**. Shanghai: IEEE, 2016: 385–390.

[5] FLAKE H. Structural comparison of executable objects [C]// **2004 IEEE Conference on Detection of Intrusions and Malware and Vulnerability Assessment**. Vienna: IEEE, 2004: 161–173.

[6] BERGERON J, DEBBABI M, DESHARNAIS J, et al. Static detection of malicious code in executable programs [J]. **Requirements Engineering**, 2001, 32(5): 132–139.

[7] JENSEN T, THORN T. Model checking security properties of control flow graphs [J]. **Journal of Computer Security**, 2012, 9(3): 217–250.

[8] YAMPOLSKIY M. Code security analysis with assertions [C]// **20th IEEE/ACM International Conference on Automated Software Engineering**. California: IEEE/ACM, 2005: 392–395.

[9] PANICHELLA A, KIFETEW F, TONELLA P. Automated test case generation as a many-objective optimisationproblem with dynamic selection of the targets [J]. **IEEE Transactions on Software Engineering**, 2018, 44(2): 122–158.

[10] BISWAS P, FEDERICO A, SCOTT A, et al. Venerable variadic vulnerabilities vanquished [C]// **Proceedings of the 26th USENIX Security Symposium**. Vancouver: USENIX, 2017: 183–198.

[11] SIDIROGLOU S, LAHTINEN E, RITTENHOUSE N, et al. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement [C]// **21th International Conference on Architectural Support for Programming Languages and Operating Systems**. Atlanta: ACM, 2016: 473–486.

[12] Hex-Rays. IDAPro disassembler [EB/OL]. [2018-02-27]. https://www.hex-rays.com/.

[13] BARDINS, HERRMANNP, LEROUX J, et al. The BINCOA framework for binary code analysis [C]// **Proceedings of the 23rd International Conference of Computer Aided Verification**. Snowbird: CAV, 2011: 165–170.

[14] KINDER J, VEITH H. Jakstab: astatic analysis platform for binaries [C]// **International Conference on Computer Aided Verification**. Berlin: Springer-Verlag, 2008: 423–427.

[15] BARDIN S, HERRMANN P, VEDRINE F. Refinement-based CFG reconstruction from unstructured programs [C]// **12th International Conference on Verification, Model Checking, and Abstract Interpretation**. Austin: VMCAI, 2011: 54–69.

[16] XU L, SUN F, SU Z. Constructing precise control flow graphs from binaries [J]. **University of California**, 2012, 32(3): 156–169.

[17] NGUYEN M H, NGUYEN T B, QUAN T, et al. A hybrid approach for control flow graph construction from binary code [C]// **20th Asia-Pacific Software Engineering Conference**. South Korea: APSEC, 2014: 159–164.

[18] 叶志斌, 姜鑫, 史大伟. 一种面向二进制的控制流图混合恢复方法[J]. 计算机应用研究, 2018, 35(7): 2168–2171.
YE Zhi-bin, JIANG Xin, SHI Da-wei. Combined method of constructing binary-oriented control flow graphs [J]. **Application Research of Computers**, 2018, 35(7): 2168–2171.

[19] Microsoft Research. Z3: an efficient SMT solver [EB/OL]. [2018-04-16]. https://github.com/Z3Prover/z3.

[20] 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 Tie-lei. Research on binary-executable-oriented software vulnerability detection [D]. Beijing: Peking University, 2011.

[21] YAN S, WANG R, SALLS C, et al. SOK: (state of) the art of war: offensive techniques in binary analysis [C]// **37th IEEE Symposium on Security and Privacy**. Fairmont: IEEE, 2016: 138–157.

[22] ALFREDV A, MONICA S L, RAVI S, 等. 编译原理: 第 2 版[M]. 赵建华, 郑滔, 戴新宇, 译. 北京: 机械工业出版社, 2009.

[23] ZALEWSKIM. American fuzzy lop [EB/OL]. [2017-11-05]. http://lcamtuf.coredump.cx/afl/.

[24] DARPA. DARPA cyber grand challenge [EB/OL]. [2017-02-01]. https://github.com/CyberGrandChallenge.