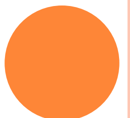
9조
200911408 이대희
201011308 고명준
201011325 김필제

About Control Flow Graphs

Constructing Precise Control Flow Graphs from Binaries

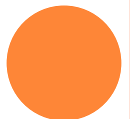
Control Flow Graph(CFG)

- a fundamental data structure representing all the control flow paths of a program that might be traversed during execution.
- The CFG is essential to many compiler optimizations and static analysis tools.



Reasons Of Using Binary

- First, source code is often unavailable since most commercial off-the-shelf (COTS) software and many third-party libraries are distributed in binary form only.
- Second, binary analyses allow us to directly reason about the actual code running on the system, which is useful because compilers can introduce discrepancies and even errors [4].
- Third, certain properties of the source code may no longer hold in the compiled binaries due to compiler optimizations. Therefore, it is possible to detect bugs and vulnerabilities in the binaries that are otherwise missed during source code analysis.



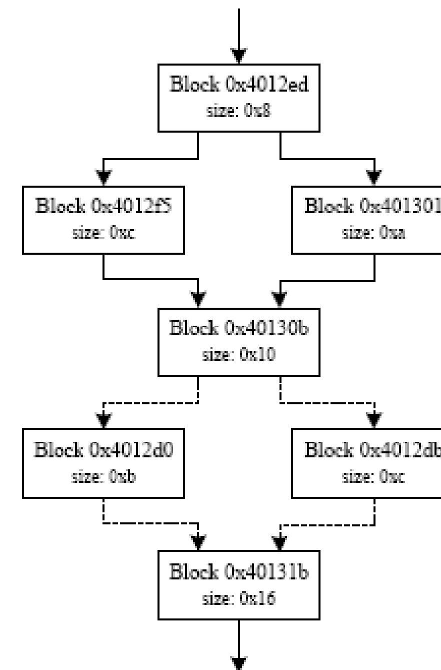
An Example of CFG Construction.

```
int (*foo)(int, int);

int add(int x, int y)
{
    return x + y;
}

int mul(int x, int y)
{
    return x * y;
}

int test(int a, int b)
{
    if (a < b)
    {
        foo = add;
    }
    else
    {
        foo = mul;
    }
    c = foo(a, b);
    return 0;
}
```

(a) Source Code

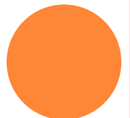
```
...
0x004012d0:  push   ebp
0x004012d1:  mov    ebp, esp
0x004012d3:  mov    eax, DWORD PTR [ebp+12]
0x004012d6:  add    eax, DWORD PTR [ebp+8]
0x004012d9:  pop    ebp
0x004012da:  ret
0x004012db:  push   ebp
0x004012dc:  mov    ebp, esp
0x004012de:  mov    eax, DWORD PTR [ebp+8]
0x004012e1:  imul   eax, DWORD PTR [ebp+12]
0x004012e5:  pop    ebp
0x004012e6:  ret
...
0x004012ed:  mov    eax, DWORD PTR [ebp+8]
0x004012f0:  cmp    eax, DWORD PTR [ebp+12]
0x004012f3:  jge    0x401301
0x004012f5:  mov    ds:0x403020, 0x4012d0
0x004012ff:  jmp    0x40130b
0x00401301:  mov    ds:0x403020, 0x4012db
0x0040130b:  sub    esp, 0x8
0x0040130e:  push   DWORD PTR [ebp+12]
0x00401311:  push   DWORD PTR [ebp+8]
0x00401314:  mov    eax, ds:0x403020
0x00401319:  call   eax
0x0040131b:  ...
```

(b) Assembly Code



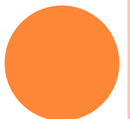
(c) Control Flow Graph

Figure 1: An Example of CFG Construction.



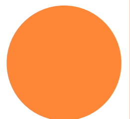
FXE

- This section presents our basic algorithm of CFG construction based on dynamic forced execution.
- It also describes key optimizations and extensions to scale FXE to realistic programs.



Definitions and Assumptions

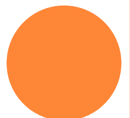
- Definition 3.1 (Control Transfer Instruction).
  A control transfer instruction (CTI) is either a conditional branch instruction (e.g., **jz, jecxz, loop) or an unconditional branch nstruction (e.g., call, jmp, ret) which directs the flow of program execution.**
  While an unconditional branch is always taken, a conditional branch can either be taken (where the execution continues at the branch target) or not taken (where the execution falls through to the next instruction).



Definitions and Assumptions

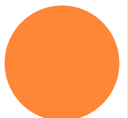
- Definition 3.2 (Basic Block).
  A basic block is a maximal sequence of consecutive instructions with a single entry and a single exit, i.e., only the first instruction can be a CTI target and only the last instruction can be a CTI.

- Definition 3.3 (Control Flow Graph).
  Given a program P, its control flow graph is a directed graph G = (V;E), where V is the set of basic blocks and E V V is the set of edges representing control flow between basic blocks. A control flow edge from block u to v is e = (u; v) 2 E.



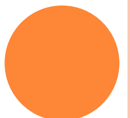
DEFINITIONS AND ASSUMPTIONS

- Assumption 1: Target Feasibility (TF). Both directions of a conditional branch are feasible. This assumption applies to most compiler-generated binaries because compilers can easily resolve trivial conditions (such as always true or false) in the source code.
  Similar to source-level CFG construction, we assume every remaining conditional branch can follow both its directions for the purpose of CFG construction.



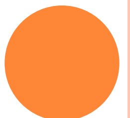
Definitions and Assumptions

- Assumption 2: Target Resolution (TR). The target of an indirect branch is completely determined by a control flow path to this indirect branch and is independent of intermediate program states.
  Indirect branches usually serve for the purpose of calling a function that is in a different module using the import address table (IAT) or calling a function using a function pointer.
  In the first situation, the target of the indirect branch is determined when the program is loaded and will remain unchanged.
  For the second case, the function pointer is properly defined or assigned a valid address along a control flow path before the branch, as is the case in Figure 1.
  An exception to this assumption is the use of jump tables; we discuss how to handle jump tables later in this section.



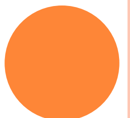
Definitions and Assumptions

- Assumption 3: Call-Return (CR).
  Every function call returns to its call site.
  When a **call instruction is issued, it saves the** address of the next instruction (i.e., the return address) onto the stack and transfers the program control to the target procedure.
  We assume that when the function returns at the **ret instruction, the** control flow always returns to the call site so that the execution continues immediately after the **call instruction.**
  This assumption holds for most functions, as it follows the semantics of the **call** and **ret instructions and also consistent with the way that compilers** generate code [1].
  However, in practice, we observe that a few library functions do not behave this way, and we treat them as special cases.



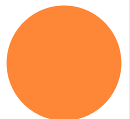
Dynamic Forced Execution

- The core of our analysis is dynamic forced execution.
  We execute the program under analysis in a virtual environment, monitor and analyze the execution trace as the program runs.
  As our goal is to construct the control flow graph, we work on basic blocks instead of individual instructions.
  During execution, a basic block ends with a control transfer instruction.
  Under Assumption 1 (Target Feasibility), each conditional branch has two possible directions.
  At the high level, we explore both directions of each conditional branch using forced execution.
  To this end, we control the execution of the program and construct the CFG using Algorithm 1.
  Initially, the CFG is empty.
  During program execution, FXE adds each basic block that belongs to the analyzed program (Line 6) to the CFG.
  We retrieve the program entry point from the executable file header, and calculate the exit point based on the value of the entry point and the size of the code section.
  For each instruction inside these basic blocks, FXE determines its type (Line 10).



Dynamic Forced Execution

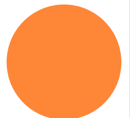
- If it is a new conditional branch, FXE predicts whether the branch is taken or not by emulating the semantics of the branch instruction using the current values of the CPU flags.
  If the branch is taken, FXE saves the address of the instruction immediately after the branch (Line 13).
  Otherwise, it records the jump target address (Line 15).
  Either way, FXE saves the current CPU and memory states.
  With these information, it is possible to later revert execution to the branch point and force the execution down the alternative path.
  When we have explored all the branch alternatives on the current execution path, or when the program terminates, FXE checks whether there are any gray branches (Line 28).
  If so, FXE rewinds the execution to the nearest gray branch, restores the CPU and memory states associated with that branch, sets the instruction pointer to point to the unexplored path, and marks that branch as black.
  FXE then forces the execution to continue along an unexplored program path.
  In this way, we explore all the program paths in a depth-first order.



DYNAMIC FORCED EXECUTION

Algorithm 1: Dynamic Forced Execution

```
Output   : ControlFlowGraph cfg
Input    : Executable exe
LocalVar: BasicBlock current, block; Instruction inst;
          InstructionPointer ip; ConditionalBranch branch
1  cfg = NULL;
2  current = NULL;
3  ip = get_instruction_pointer();
4  while true do
5      while block = get_block(ip) do
6          if exe.EntryPoint ≤ block.pc < exe.ExitPoint then
7              connect_block(cfg, current, block);
8              current = block;
9              while inst = get_instruction(block, ip) do
10                 if inst.type == ConditionalBranch and
                   find_branch(inst) == NULL then
11                     branch = get_branch(inst);
12                     if branch_is_taken(branch) then
13                         branch.next_ip = ip + inst.length;
14                     else
15                         branch.next_ip = get_target(branch);
16                     branch.state = gray;
17                     save_cpu_mem_state(branch);
18                     add_to_branch_list(branch);
19                 try
20                     ip = execute_inst(inst);
21                 catch (exception)
22                     error_handler(ip, block, exception);
23         else
24             try
25                 ip = execute_block(block);
26             catch (exception)
27                 error_handler(ip, block, exception);
28     if branch = get_last_gray_branch() then
29         load_cpu_mem_state(branch);
30         ip = branch.next_ip;
31         branch.state = black;
32         current = get_branch_block(branch);
33     else
34         break;
35 split_block(cfg);
36 return cfg;
```



FXE Architecture Overview

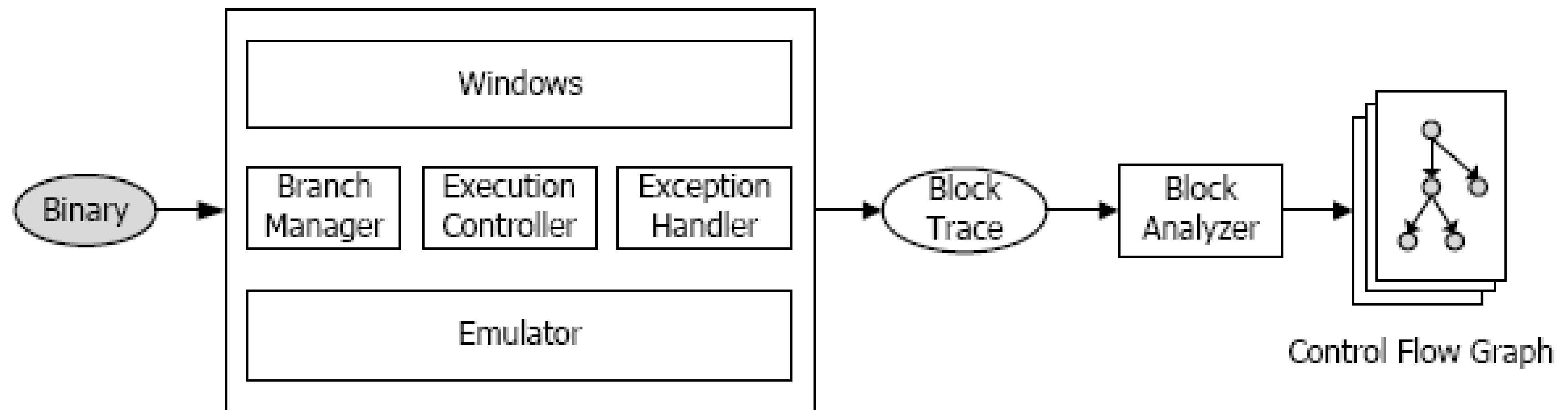
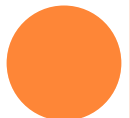


Figure 3: FXE Architecture Overview.



Statement Of Purpose

Control Flow Graph (CFG)

We have presented a novel, practical technique based on forced execution to extract precise control flow information from binaries.

Our goal is to have the constructed control flow graph as close to a binary's ideal control flow information as possible.

Using forced execution, our technique systematically explores both directions at each branch point and computes the targets of indirect branches at run time.

We do not consider the detail error.

We only focus on effective technique that can automatically generate precise CFGs from binaries.