Constructing Precise Control Flow Graphs from Binaries

Liang Xu Fangqi Sun Zhendong Su

Department of Computer Science
University of California, Davis
{leoxu, fqsun, su}@ucdavis.edu

ABSTRACT

Third-party software is often distributed only in binary form. For software engineering or security considerations, it is important to be able to analyze binaries. One fundamental obstacle to perform binary analysis is the lack of precise control flow information. Existing techniques to construct control flow of binaries are either static or dynamic. Traditional static techniques usually disassemble a program's binary image statically and build the control flow graph (CFG) from the assembly-level representation of the program. They are limited in precision because of difficulty in statically resolving indirect branches. Dynamic techniques, on the other hand, suffer from poor coverage and scalability. Hybrid techniques based on combined dynamic and symbolic path exploration can be used to improve code coverage, but still suffer from poor coverage and scalability because they rely on expensive constraint solving to generate alternative inputs to explore different control flow paths.

This paper presents the first practical technique that constructs precise control flow graphs from binaries. Our technique rests on the key observation that the possible targets of most indirect branches are independent of intermediate program states, and thus by systematically *forcing* a program's execution to explore both branches of each conditional, we can discover the program's precise control flow. Specifically, we run the program under analysis in a controlled virtual environment. At each conditional branch, we save the address of the path not taken, and force the execution to explore that path later. In essence, we leverage both dynamic execution to compute the targets of indirect branches (as in traditional dynamic CFG construction) and efficient (since it is forced) systematic exploration specifically targeting the problem of control flow construction from binaries. We also introduce effective optimizations and heuristics to make our dynamic forced execution scalable and practical. We have implemented our technique in a practical tool FXE for x86 binaries and performed detailed evaluation of its precision, generality, and performance. Our results show that FXE constructs highly precise CFGs and scales to real-world programs, significantly outperforming state-of-the-art alternatives.

1. INTRODUCTION

Many program analysis techniques, such as data flow analysis and data dependence analysis, rely on the existence of a control flow graph (CFG), a fundamental data structure representing all the control flow paths of a program that might be traversed during execution. By inspecting and analyzing CFGs, it is possible to perform program verification [6, 7, 16, 18], find software bugs [31], and generate test cases [15] automatically.

Although effective techniques exist to build precise CFGs from source code, none exists that can construct reliable CFGs from binaries because of various difficulties introduced by low-level features of machine code. However, such a technique is highly desirable because it is essential for performing program analysis on binaries, which is important for the following reasons. 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.

Both static and dynamic techniques have been proposed to construct CFGs from binaries. Static techniques work by analyzing the structure of binaries and parsing instruction opcodes. They may have good code coverage because they operate on the complete binary, but suffer from poor precision because it is difficult to resolve indirect branch targets statically. On the other hand, dynamic techniques primarily rely on monitored executions of the binary on a particular set of runs, and are unable to guarantee good code coverage. They also have poor scalability due to long-running loops. More recently, hybrid techniques based on concolic testing [8, 9, 14, 26] and multiple path exploration [22] have been proposed to improve path coverage of purely dynamic techniques. However, these approaches do not yet scale to large programs. First, they rely on expensive constraint solving to generate test inputs to drive execution down alternative paths. When constraints are unsolvable, the corresponding conditional branch target remains unexplored. Second, similar to dynamic CFG construction, they suffer from ineffective treatment of loops. CUTE [26] and KLEE [9] are two publicly available, state-of-the-art tools in this category. Our experience with these tools suggests that they are not yet suitable for practical CFG construction from binaries. For example, on even small test programs from the Siemens Test Suite [17], they did not achieve good coverage or they failed to terminate in reasonable amount of time.

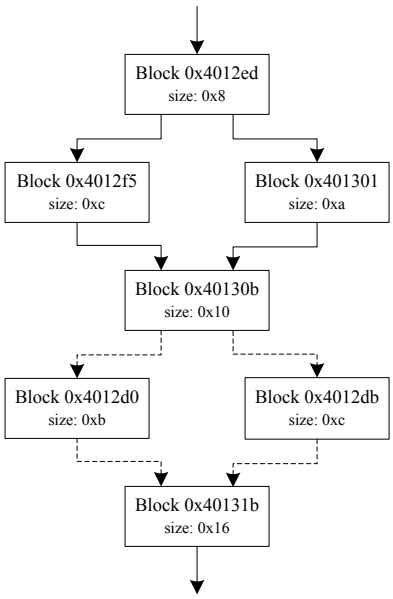
In this paper, we introduce a novel, effective technique that can automatically generate precise CFGs from binaries. Our approach is motivated by the observation that the targets of indirect branches are often independent of intermediate program states and can be properly resolved by systematically exploring both directions of each conditional branch. Instead of relying on expensive dynamic, symbolic execution as CUTE and KLEE do, our key insight is to use *forced execution* to efficiently explore both directions of every conditional branch. In particular, we dynamically track conditional branch instructions and identify branch points where we save the current program state, before allowing execution to proceed. Later, we return to the branch point, restore the saved program state, and force the execution to follow the other path by manipulating the

```
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.

CPU instruction pointer. In this way, we explore both directions at each branch point and resolve indirect branches dynamically to construct precise control flow graphs.

Our approach, based on dynamic forced execution, inherits the merits of dynamic CFG construction yet overcomes its disadvantages. On one hand, we can resolve indirect branch targets because we execute the code dynamically. On the other hand, we achieve high code coverage by systematically forcing the program execution along both directions of each branch point. Another limitation of dynamic CFG construction is that it takes time proportional to the number of executed instructions at run time. In other words, the analysis time is not bounded by code size, making the analysis inefficient, especially for programs with long-running loops. In our approach, we achieve efficiency by introducing an intraprocedural analysis that decides which instructions we need to execute at run time. Moreover, unlike previous hybrid approaches, we take advantage of the particular problem setting and do not rely on constraint solvers, allowing our technique to scale to realistic programs.

We have developed a prototype named FXE (**F**orced e**X**ecution **E**ngine) based on the QEMU [5] emulator. It works on x86 executables in the Microsoft Windows PE format. We have implemented a mechanism to save and restore snapshots for the executing process, and also force the execution to continue from a specific program point. To improve scalability, we use a backward reachability analysis to avoid unnecessary code re-execution. Our empirical results show that the proposed system is able to construct binary CFGs with high precision and coverage, is robust against different compilers and optimization options, and also scales to large programs. In particular, we show that FXE produces nearly ideal CFGs and substantially improves state-of-the-art alternatives such as the popular commercial disassembler IDA Pro. We believe our technique is a promising enabling technology for the analysis of binaries.

This paper makes the following main contributions:

- We propose a novel, practical technique based on forced execution to construct highly precise CFGs from binaries.
- We have designed and implemented a framework that analyzes Windows executables by executing them in a controlled virtual environment. Our system monitors the dynamic execution trace and generates the control flow graph containing all the executed basic blocks and control flow edges.
- We evaluate our system on non-trivial programs and show that we are able to effectively and efficiently construct CFGs for the binaries under analysis.

The remainder of this paper is structured as follows. The next section describes our approach using an illustrative example. Section 3 provides the details of our path exploration algorithm and explains key optimizations. In Section 4, we describe the design and implementation of FXE. In Section 5, we provide an empirical evaluation on the precision, coverage, generality, and performance of FXE. Section 6 surveys related work, and Section 7 concludes.

2. EXAMPLE

We first use a simple example to illustrate how FXE constructs control flow graphs from binaries. Consider the C program in Figure 1a. Note that although we present this example in C code, FXE works directly on binaries. The function pointer `foo` can point to either the function `add` or `mul`, depending on the values of `a` and `b` at run time. When FXE analyzes this program, it executes the binary and dynamically monitors the executed instructions. The assembly code for the example is given in Figure 1b.

In our example, the `test` function starts from the instruction at `0x4012ed`. For the purpose of illustration, suppose we invoke the function with `test(1,2)`. When FXE encounters the conditional branch instruction **`jge`** at `0x4012f3`, it predicts the branch target according to the semantics of the branch instruction and the current values of the CPU flags. This branch is not taken because $a < b$. FXE saves the address of the path not taken (*i.e.*, `0x401301`), snapshots the program state, and resumes execution from `0x4012f5`. The execution continues until the **`call`** instruction at `0x401319` is encountered. Since this is an indirect branch instruction, whose

target is given by the current value of the **eax** register, FXE retrieves the value from the **eax** register, and transfers the program control to the target accordingly. The program then continues to execute the `add` function at `0x4012d0`. When the **ret** instruction executes at the end of the function, the return address is retrieved from the program stack and execution is directed back to the call site to continue from `0x40131b`. Now, the **jge** conditional branch is *gray*, indicating that we have only followed one of its two directions. In order to explore the other path, FXE restores the program state, and manipulates the CPU instruction pointer to force the execution to take the other path from `0x401301`. By restoring the program state, it revokes all the side-effects introduced on the other path. In addition, FXE marks the conditional branch at `0x4012f3` as *black*, which means that both paths of this conditional branch have been explored. The **eax** register is then assigned the address of the `mul` function, and the `mul` function is explored after the indirect branch at `0x401319`. The path exploration continues until all the conditional branches in the program are marked black.

Figure 1c shows the control flow graph constructed for the example program, which is the ideal CFG we wish to construct. This example illustrates how our technique can precisely handle indirect branches. In the constructed CFG, instructions are partitioned into basic blocks, and the control flow between any two basic blocks is represented using a directed edge. We denote each block with its starting address (*i.e.*, the address of the first instruction of the block). Edges drawn in dashed lines stand for indirect control transfers.

3. 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.

3.1 Definitions and Assumptions

We first give a few basic definitions adapted to the binary setting.

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 instruction (*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).

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 $E \subseteq V \times V$ is the set of edges representing control flow between basic blocks. A *control flow edge* from block u to v is $e = (u, v) \in E$.

Next, we state a few key assumptions we have observed to hold in general over binaries to guide and motivate the design of FXE.

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.

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.

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.

3.2 Basic Algorithm

Our basic algorithm has three main components: 1) dynamic forced execution, 2) saving and restoring program states, and 3) handling exceptions.

3.2.1 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). 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.

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;
```

When all the conditional branches are marked black, FXE terminates the program execution since all the program paths have been explored. During program execution, each block ends with a control transfer instruction. If the execution later jumps into the middle of a previous block, that block needs to be split into two blocks according to the definition of a basic block. For performance reasons, we defer the block-splitting process to an offline analysis phase after dynamic execution. During offline analysis, FXE checks each block in the CFG whether it is a sub-block of another block by examining their starting addresses and sizes. If one block contains another block, FXE splits the larger block into two blocks. The `split_block` procedure at Line 35 implements splitting blocks.

In the course of forced execution, the program can often be in inconsistent states. This is because, instead of relying on constraint solvers to solve path constraints, FXE simply manipulates the CPU instruction pointer to control the program execution and guide the path exploration. However, this is not an issue for us. Since our goal is to build the control flow graph, we usually do not care whether the computation of the program is correct or not. Instead, we are only interested in the control flow targets. For direct branches, the control flow targets are given by the control transfer instructions, which should always be correct. For indirect branches, the control flow targets are usually independent of intermediate program states as described in the *Target Resolution* assumption. Even if the program is in an inconsistent state, our analysis will not be affected as long as we have the correct indirect branch targets. Therefore, it is usually safe to ignore the inconsistencies in the program states and continue the path exploration. For these reasons, we consider the forced execution approach quite suitable for the specific problem setting of binary CFG construction.

Since we dynamically execute a program to construct its control flow information, it may seem that the concrete input used to bootstrap the analysis process can affect the analysis result. However, this is not the case. Our observation is that, for most programs, the control flow graph we are trying to construct is an intrinsic static property of the program. This means the CFG is an invariant independent of any specific program input. Unlike traditional dynamic analysis where the control flow paths taken at runtime are largely determined by the specific input, our approach does not rely on concrete inputs to drive the execution along different paths. Instead, it systematically explores all program paths and is insensitive to any given specific input. In fact, we ignore any concrete input to the program and just use anything that happens to be in the memory when inputs are required. This is effectively the same as using random values for the inputs.

The benefit of using dynamic forced execution is twofold. First, we can conveniently resolve indirect branch targets, which is an intractable problem for static analyzers. As we execute the program dynamically, it is straightforward to retrieve the targets either from CPU registers or from specified memory locations. Second, we provide good code coverage that is difficult for dynamic approaches to achieve. With forced execution, we explore both directions of every conditional branch.

However, dynamic forced execution may also cause problems. First, the CPU and memory states are not consistently updated, which may lead to runtime exceptions. If these runtime exceptions are not properly handled, the program will crash, resulting in unexplored paths. Second, we might follow invalid program paths if the target of an indirect branch depends on corrupted program states. Finally, the program execution time is not bounded by the size of the code because of loops. In the worst case, the analysis may never terminate if we encounter infinite loops. We discuss our solutions to these problems in the following sections.

3.2.2 Saving and Restoring Program States

When FXE detects a conditional branch, it takes a snapshot of the program state including the values of CPU registers and contents of the virtual memory space used by the program. Even though we do not consistently maintain the program state, it is still necessary to save the program state at each conditional branch. This is because, by Assumption 2 (Target Resolution), indirect branch targets can be saved in CPU registers or memory locations. If we do not save these values, they could be overwritten while we explore one path of the branch, and we may use incorrect target values when we force the execution to follow the other path of the branch.

When FXE restores a program state, it loads the saved CPU register values and the contents of program memory address space to overwrite the current program state. In this way, it revokes all the side-effects introduced when executing the first path of the conditional branch. This means that FXE restores any value overwritten along the first path, including those that will serve as indirect branch

targets along the other path. This ensures that we follow both directions for each conditional branch conceptually in parallel, which means the exploration of both paths starts with the same state. By saving and restoring snapshots at branch points, we avoid introducing additional inconsistencies into the program state.

3.2.3 Exception Handling

During forced execution, runtime exceptions occur because of inconsistent program states. We need to properly intercept and handle all these exceptions. Otherwise, the program will terminate or crash, leaving paths unexplored.

We divide the runtime exceptions into two categories: *critical exceptions* and *non-critical exceptions*. When a critical exception occurs, FXE cannot continue exploring the current path. For instance, failure to read the memory location used as an indirect branch target constitutes a critical exception. In this case, FXE rewinds the execution to the nearest gray branch, and resumes execution to explore an unexplored path. On the other hand, a non-critical exception does not affect the program control flow, and FXE can continue to explore the current path. For example, if we encounter a data transfer instruction writing to a read-only memory location, it is a non-critical exception. In this case, FXE just skips the instruction that causes the exception, and resumes the program execution from the next instruction. Regardless of the type of an exception, FXE intercepts and handles the exception before the program crashes or terminates, so that it can continue with the path exploration.

3.3 Optimizations and Extensions

As discussed earlier, our basic technique may require long or unbounded analysis time. In this section, we describe some key optimizations of and extensions to our basic path exploration strategy to scale FXE to realistic programs.

3.3.1 Skipping Blocks

Example 1. Consider the following instruction sequence:

```
L1:  mov   eax, 0x0
L2:  mov   ecx, 0x1000
L3:  add   eax, ecx
L4:  dec   ecx
L5:  jns   L3
```

This is a loop that calculates the sum of all positive integers no more than `0x1000`. During execution, the instructions from L3 to L5 would be executed `0x1000` times. However, in terms of control flow, only the first and last loop iterations discover new instructions or control flow edges, while the other loop iterations do not contribute to the construction of the CFG. In order to construct the CFG, we only need to execute the block (L3–L5) in Example 1 once and simply skip it in the subsequent iterations of the loop. Now the question is: which blocks can we skip and how do we identify them? To answer this question, we begin with a few definitions.

Definition 3.4 (Indirect Branch Block). An *indirect branch block* is a basic block that ends with an indirect branch instruction.

During execution, an indirect branch block may jump to a new target every time it is executed. For instance, recall the example in Figure 1. The block at `0x40130b` is an indirect branch block, because it ends with a **`call`** instruction whose target is specified by the value of the **`eax`** register. When this block is executed, the program control may transfer to two different targets.

Definition 3.5 (Contributing Block). Given a CFG $G = (V, E)$, a block $b \in V$ is a *contributing block* if it is an indirect branch block, or it is on a control flow path leading to an indirect branch block:

$$b = b' \ \vee \ \exists\, v_0 = b, \ldots, v_n = b' \, [\forall 1 \leq i \leq n \ (v_{i-1}, v_i) \in E]$$

where b' is an indirect branch block.

We consider an indirect branch block a contributing block because it may directly contribute to the control flow with different targets. If a block is not an indirect branch block, it may still influence the value of an indirect branch target and thus indirectly contribute to the CFG when there is a path leading to an indirect branch block. Therefore, we also consider such blocks as contributing blocks that need to be re-executed. For a non-contributing block, we execute it once and skip it when we encounter it again in later exploration.

In order to identify contributing blocks, we conduct control flow analysis on the partial CFG upon the execution of an indirect branch block. We use the following notations for the presentation of our algorithm. Let $G_b = (V_b, E_b)$ denote the partial CFG when the block b is executed, where b is an indirect branch block. At the high level, we use a backward reachability algorithm to compute the set of blocks S_b containing every block $v \in V_b$ from which b is reachable. More formally, $S_b = \{\, v \in V_b \mid \exists v_0 = v, \ldots, v_n = b[\forall 1 \leq i \leq n \ (v_{i-1}, v_i) \in E_b] \,\}$. In fact, the computed block set S_b is an over-approximation that may contain unnecessary blocks. This is because that many blocks in S_b will not be re-executed if there are no back edges to these blocks in the CFG. In order to further improve efficiency, we perform an *intraprocedural* analysis to identify contributing blocks, which means that we limit the computation within the current function. To this end, we use a heuristic to locate function entry points. The first few instructions of a function are usually responsible for setting up a new stack frame. A function usually starts with the instructions **`push ebp`** and **`mov ebp, esp`**, followed by instructions that make room for function local variables[1]. We examine the beginning of each block to see whether the instructions implement such typical function prologues. Our backward intraprocedural reachability algorithm is presented in Algorithm 2.

Algorithm 2: **Backward Intraprocedural Reachability**

Output : BlockSet S_b
Input : Block b; PartialCFG G_b
LocalVar: BlockSet W, R, T; Block u, v;

```
1  S_b ← ∅;
2  W ← { b };
3  R ← { b };
4  T ← ∅;
5  while W ≠ ∅ do
6      foreach v ∈ W do
7          v.contrib ← true;
8          S_b ← S_b ∪ { v };
9          if v.function then continue;
10         foreach u ∈ V_b such that (u, v) ∈ E_b do
11             if u ∉ R then T ← T ∪ { u };
12             R ← R ∪ { u };
13     W ← T;
14     T ← ∅;
```

Initially, all blocks are marked as non-contributing. Every time an indirect branch block is executed, we apply Algorithm 2 to compute a set of blocks, and mark each block in the set as a contributing

[1]The **`enter`** instruction is also provided in the x86 architecture to set up a new stack frame for a procedure. However, this instruction is less commonly used by compilers.

block (Line 7). It is necessary to compute the set again even if the same indirect branch block has been executed before, because we work on partial CFGs, and the computed set may contain new blocks as the execution continues. At the beginning of the computation, the working set W starts with the indirect branch block b. The algorithm recursively tracks backward to include blocks leading to the ones in W. As an intraprocedural analysis, the backward tracking stops at each block that implements a function prologue (Line 9). During the computation, we need to keep track of all the marked blocks in a set R. If a block is marked during previous iterations of the loop, we do not add it to the working set (Line 11) to avoid infinite loops.

Even if we do not re-execute non-contributing blocks, we still need to take special care of loops whose blocks are all marked as contributing blocks. We devise a block quota scheme to address such cases. We assign a quota to each block, and dynamically adjust the quota for the blocks. We assign the block quota in a way that favors indirect branch blocks so that we can observe as many new branch targets as possible, while still limiting the number of times each block can be executed. To this end, we set each block's initial quota to one. Whenever a block is executed, its quota is decremented by one. Whenever we encounter an indirect branch block, we remember its indirect branch target. If we identify a new target for the block, we increase its quota by one so that it has another opportunity to execute. In addition, we also increase the quota of each block in the block set computed in the backward reachability algorithm. We only re-execute a block if it is a contributing block and still has unused quota. Otherwise, we rewind the execution to the nearest gray branch and explore a new path from that branch point.

Although we accelerate our dynamic path exploration by avoiding the re-execution of certain blocks, this also introduces some potential problems. Consider the scenario in Figure 2.

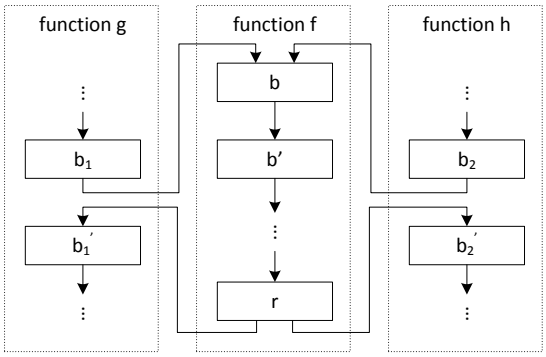


Figure 2: Handling Call Branches.

Function f is called from two other functions g and h with b_1 and b_2 as their call sites respectively. It returns to b_1' and b_2' from its return block r. During forced execution, the quota for certain blocks in f may be exhausted after f is invoked from g. In this case, the execution is rewound to explore other blocks in f, and it eventually returns to b_1' from r. However, when f is later invoked from h, the execution would be rewound, before reaching r, to the nearest conditional branch, which is outside f because all the paths in f have already been executed. The problem is that we would never explore the fall-through path starting from block b_2'. In addition, we would also miss the function return edge (r, b_2') in our CFG.

In order to explore the function fall-through path, we treat each function call branch in a similar way to how we handle conditional branches. We also add each call branch to the branch list in Algorithm 1, so that when the execution is rewound, the other unexplored path (*i.e.*, the fall-through path) of the call branch can be followed. To address the issue of finding the missing function return edge, we could remember the return block r for function f which has been explored, and manually connect it to b_2' when the execution is rewound. However, a function can have more than one return blocks, which complicates the problem. To mitigate this issue, we perform an intraprocedural forward reachability analysis on the partial CFG. More specifically, on each function call, we remember the call site (*i.e.*, caller) and the target function (*i.e.*, callee). Whenever the function is invoked in a different context (*i.e.*, from a different call site), we track forward on the partial CFG and reset the quota for all the blocks in the target function. We identify blocks within the function by looking for **`ret`** instructions which are considered function return points. In this way, each block within the function has more opportunities to be executed under different contexts, and the function is more likely to return to different call sites. Therefore, this technique helps find more return edges during forced execution.

3.3.2 Skipping External Code

Since our goal is to construct the CFG for the program under analysis, we skip the execution of all external functions, which further improves the scalability of our approach. For our analysis, external code (*e.g.*, a library) may influence the value of an index into a jump table, but we assume that it does not otherwise affect the computation of any indirect branch target in the program. By Assumption 3 (Call-Return), when a program invokes external code, the execution will eventually return to the caller. Therefore, when FXE detects a function call pointing to external code, it forces the execution to immediately return to the call site and continue along the fall-through path. However, we have observed that a few library functions (*e.g.*, `exit`, `abort`, `ExitProcess`) do not return to their original call sites. These functions are usually invoked when the program is about to terminate. We treat these functions specially and do not follow their fall-through paths.

3.3.3 Handling Jump Tables

Indirect branches are often used with jump tables where the target is retrieved according to an index into the table. A jump table usually contains an array of pointers to code that handles various cases based on the index. During our analysis, we want to discover and explore all possible targets in each jump table. To this end, we use a simple yet effective pattern matching algorithm to identify the starting addresses of jump tables at indirect branch instructions.

Example 2. The following code gives an example of jump table:

```
0x0046dff2:  cmp eax, 0xc
0x0046dff5:  ja  0x46e097
0x0046dffb:  jmp dword ptr ds:[eax*4+0x46e004]
0x0046e002:  ...
0x0046e004:  dd 0x46e0b6
0x0046e008:  dd 0x46e038
0x0046e00c:  ...
```

In Example 2, the target of the indirect branch at `0x46dffb` can be one of the twelve values given in the jump table starting at `0x46e004`. This is one of the few patterns that we detect to identify jump tables. In this example, we assume the address at `0x46e004` starts a jump table, and consecutively read target values from the table until discovering a value that falls out of the range of the code section. At each indirect branch where a jump table is detected, we also save the program state, and later restore the state before exploring each possible branch target.

3.3.4 Handling Callback and Thread Functions

FXE is able to identify blocks and control transfer edges that are reachable from the program entry point. However, some of the

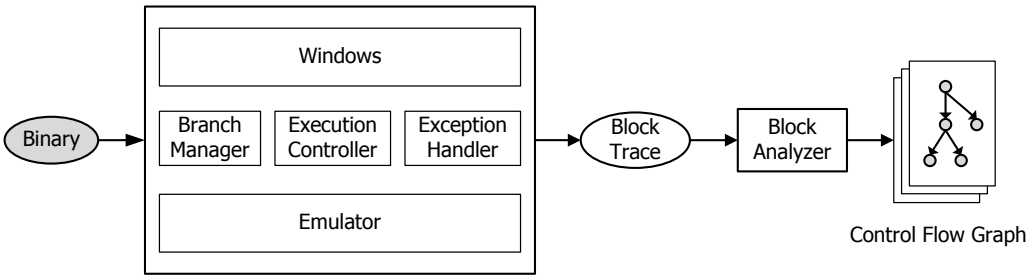


Figure 3: FXE Architecture Overview.

code may be in the form of callback functions that is only reachable from external code. A callback function is registered and later invoked when certain events occur. For example, the library function `atexit`, as suggested by its name, registers a callback function that will be invoked when the program terminates. Since FXE skips all external code, it will not execute these callback functions by following the program execution. To address this issue, FXE intercepts the registration of callback functions and retrieves their addresses from the arguments. It forces the execution to explore these callback functions after all other program paths are explored.

FXE can automatically detect callback function registration sites by examining the arguments of each function call. Since function arguments are usually saved onto the program stack immediately before the call, FXE scans through the basic block that ends with a function call, and looks for **`push`** or **`mov`** instructions that copy data to the stack. If the operand value falls within the range of code sections, and if the instructions at that address implement a typical function prologue, we assume that a callback function is registered.

Example 3. The `qsort` library function registers a callback function (*i.e.*, `comparator`) that compares two elements.

```
push 0x40bbc0                  ; comparator
push 0x10                      ; size
push dword ptr ss:[ebp-0x1c]   ; num
push dword ptr ss:[ebp-0x28]   ; base
call 0x41a6b0                  ; call qsort
```

In this example, when FXE executes the **`call`** instruction, it analyzes the preceding **`push`** instructions. Since the address `0x40bbc0` is within the code section, and it implements a function prologue, FXE identifies it as a callback function and explores it later.

Although FXE works well for single-threaded programs as it always analyzes the code for the main thread of the program, it is also desirable to be able to analyze multi-threaded programs. As we do not allow external code to be executed, the process under analysis cannot spawn child threads. In fact, FXE is able to discover the starting addresses of new threads as callback functions. This is because, when FXE intercepts function calls that are responsible for creating threads (*e.g.*, `beginthread`, `CreateThread`), it is able to extract the starting address of the new thread from the arguments that are copied onto the stack. In this way, we force the program to execute the code for the child threads later when all other program paths have been explored.

3.3.5 Eliminating Invalid Blocks

The forced execution approach may lead to invalid paths when indirect branch targets depend on incorrect values from corrupted CPU or memory states. We argue that the probability for this imprecision is low because the possible range for the executable code is trivial compared to the entire addressable memory space. For instance, in a 32-bit system, the memory address range is as large as 4 GB, while the size of a program is very small, usually not exceeding a few megabytes. Therefore, the probability of an incorrect target value to fall into the valid code range is insignificant. If, however, an incorrect value happens to be within the code range, FXE takes the following measures to detect such invalid paths.

First, if the target address does not contain valid instructions, or an invalid opcode exception occurs, FXE assumes that the indirect branch target is invalid, and rewinds execution to the nearest gray branch. Second, if FXE detects privileged or sensitive instructions (*e.g.*, **`hlt`**, **`in`**, **`out`**) in user code, it considers the block as invalid. Finally, during the offline analysis phase, if FXE detects overlapping blocks, one of the two blocks must be invalid. In this case, FXE uses some heuristics to decide which block is invalid. For instance, the block with more incoming control transfer edges is more likely to be valid.

4. IMPLEMENTATION

To demonstrate the feasibility of our approach, we have implemented our technique in a prototype tool FXE to construct control flow graphs for Windows PE executables. We first give a high-level overview of FXE, and then describe how each architectural component work to accomplish the analysis goals.

4.1 Architecture Overview

FXE is based on QEMU [5], a whole system emulator. We choose QEMU in our implementation mainly for two reasons. First, QEMU's binary translation works at the granularity of basic blocks[2], which is convenient for our CFG construction algorithm. We can also instrument each CPU instruction to identify specific control transfer instructions we are interested in. Second, with QEMU, we have full access to the emulated system's CPU and memory states, which makes forced execution possible. In our current prototype, QEMU emulates an x86 computer and runs a Microsoft Windows guest operating system on top of it.

Figure 3 illustrates the overall architecture of FXE. It takes an x86 Windows PE executable as an input, and generates a CFG output for the program. We perform the analysis in two phases: a dynamic execution phase and an offline analysis phase. During the dynamic execution phase, FXE loads and executes the binary program in the guest OS. It controls the program execution to explore different paths and monitors the dynamic execution trace. The second analysis phase is performed offline, in which FXE splits and validates blocks as necessary. Finally, FXE generates the CFG output at the end of this phase.

4.2 Branch Manager

The branch manager keeps track of all the branches during dynamic execution. It instruments instructions executed by the program and searches for conditional branch instructions. When it

[2]These blocks are called *Translated Blocks* in QEMU, which are slightly different from basic blocks.

detects such an instruction, it checks whether the branch has been executed before. If it is a new branch, it marks the branch as gray, and notifies the execution controller to take a snapshot of the current program state. Then, the branch manager predicts whether the branch is taken or not by emulating the semantics of the branch instruction using the values of the CPU flags. If the branch is taken, it saves the address of the fall-through instruction. Otherwise, it retrieves the address of the target from the operand of the branch instruction. Later, when the other path of the branch is explored, the branch manager receives notification from the execution controller and marks the branch as black.

4.3 Execution Controller

The execution controller, as the core component of FXE, decides how the program paths are explored. At each conditional branch, it takes a snapshot of the program state including the values of CPU registers and contents of the memory address space used by the process. We devised a mechanism to save and restore the memory state for the process under analysis. To this end, the execution controller identifies the active memory pages used by the program by traversing the process page directories and page tables in the kernel memory of the guest OS. When saving the memory snapshot, it only copies the contents of valid pages that belong to the process, and saves them into a file. Later, when the execution controller forces the execution down the alternative path, it restores the CPU state, loads the memory state from the saved file, and writes the values back to the guest memory. It then sets the CPU instruction pointer to point to the unexplored path, and requests the branch manager to mark the branch as black. The execution controller also performs the backward intraprocedural reachability analysis described in Section 3.3.1 to determine which blocks to re-execute.

4.4 Exception Handler

The exception handler is responsible for intercepting and handling runtime exceptions, which are mostly caused by inconsistent program states. We observed that memory access exceptions constitute a large part of such exceptions. In order to intercept illegal memory access, we hook the page fault handler module in QEMU and examine the virtual CPU state. The exception handler is able to intercept and handle the following exceptions:

- Null pointer dereference
- Accessing kernel memory from user mode
- Writing to memory that is read-only
- Accessing memory that is not committed
- Division by zero
- Invalid opcode
- General protection fault

When the exception handler detects an exception, it decides whether it is critical or not. In case of a critical exception where the execution can no longer continue along the current path, it requests the execution controller to restore the program state and rewind execution to the nearest gray branch. If it is a non-critical exception, it informs the execution controller to skip the current instruction that caused the exception and continues execution from the next instruction. In either case, the program can recover from exceptions, so that the path exploration can continue.

4.5 Block Analyzer

The block analyzer works offline when the dynamic execution phase is complete. It operates on the block trace recorded during program execution. For each block b_1, it checks whether there is another block b_2 such that b_2 is a sub-block of b_1 by comparing the starting addresses and sizes of the blocks. If such a block b_2 is found, it splits the block b_1 into two blocks and updates the corresponding edges. When the block analyzer detects two overlapping blocks, it uses some heuristics to decide which block is invalid. It then tracks backward in the CFG to find preceding blocks that lead to the invalid blocks. If a preceding block is not an indirect branch block, it is also considered invalid. The block analyzer removes the identified invalid blocks as well as the edges associated with those blocks, and finally provides the CFG output.

5. EMPIRICAL EVALUATION

We designed our evaluation of FXE to answer a few questions regarding its precision, generality, and scalability: 1) Can we trust the constructed CFGs, *i.e.*, how precise are they? 2) How general is FXE, *i.e.*, does it work with different compiler optimization options or even different compilers? 3) How effective is FXE compared to state-of-the-art alternatives? 4) Does it scale to large programs? Section 5.1 considers the first question and includes an evaluation of the soundness and completeness of FXE on a set of standard test programs. We show that FXE produces nearly ideal CFGs on these test programs. Section 5.2 considers the second question and demonstrates that compilers have limited impact on the FXE results. Section 5.3 considers the third question and shows that FXE produces significantly more precise results than the state-of-the-art commercial tool, IDA Pro, on a set of large standard benchmark programs. Finally, Section 5.4 considers the last question and shows that FXE scales well to large programs.

5.1 Soundness and Completeness

Before describing our evaluation of FXE's precision, we formally define the term precision in our evaluation setting in terms of soundness and completeness.

Definition 5.1 (Ideal CFG). Given a program P, its *ideal CFG* G is the union of all dynamic CFGs on P's input domain I:

$$G = \bigcup_{i \in I} \mathrm{CFG}(P, i)$$

where $\mathrm{CFG}(P, i)$ is the dynamic CFG for P on input i.

Definition 5.2 (Soundness and Completeness). Given the ideal CFG $G = (V, E)$ for a program P, the CFG $G' = (V', E')$ for P is *sound* iff $\forall v \in V'(v \in V)$ and $\forall e \in E'(e \in E)$. G' is *complete* iff $\forall v \in V(v \in V')$ and $\forall e \in E(e \in E')$.

Given the ideal CFG of a binary, we want the CFG constructed by FXE to be as close to the ideal CFG as possible. However, it is difficult (in fact undecidable) to compute the ideal CFG. Therefore, in practice, we cannot compare the CFG constructed by FXE with the ideal one, which is generally unavailable. For validation purpose, we can compare our result with a good approximation of the ideal CFG. One way to approximate the ideal binary CFG is to construct the CFG from source code. However, we do not use source-level CFGs in our evaluation mainly for two reasons. First, constructing CFGs from source code requires pointer analyses to understand function pointers – the main source of indirect branches in binaries. However, pointer analyses can be imprecise as they usually over-approximate points-to information. Second, it is difficult to map source-level CFGs to binary-level ones. Compilers usually introduce additional code into the binary even without any optimization. For instance, for a simple program that does nothing but directly returns, there are hundreds of basic blocks and tens of conditional branches in the compiled binary.

Program	LOC	Test Cases	Size (KB)						
			GCC -O0	GCC -O1	GCC -O2	GCC -O3	GCC -Os	MSVC	ICC
printtokens	726	4,130	26.3	26.1	25.5	26.4	24.3	9.5	15.0
printtokens2	570	4,115	22.3	21.4	20.7	20.6	20.3	6.5	13.0
replace	564	5,542	22.6	22.1	22.1	24.1	21.1	6.5	15.0
schedule	412	2,650	20.0	20.0	20.0	21.5	19.5	6.0	10.5
schedule2	374	2,710	20.3	20.3	20.3	23.3	19.6	6.0	11.0
tcas	173	1,608	20.1	19.6	20.1	20.1	19.6	5.0	8.0
totinfo	565	1,052	21.5	21.5	21.5	21.5	20.8	6.5	35.5

Table 1: Siemens Test Suite programs.

Program	$\|V_d\|$	$\|E_d\|$	$\|V_f\|$	$\|E_f\|$	$\frac{\|V_d \cap V_f\|}{\|V_f\|}$	$\frac{\|E_d \cap E_f\|}{\|E_f\|}$	$\frac{\|V_f \cap V_d\|}{\|V_d\|}$	$\frac{\|E_f \cap E_d\|}{\|E_d\|}$
printtokens	308	435	317	451	97.48%	95.12%	100.00%	98.39%
printtokens2	346	483	348	486	99.43%	98.56%	100.00%	99.17%
replace	329	435	351	481	94.02%	89.81%	100.00%	99.08%
schedule	216	277	220	282	99.09%	97.87%	100.00%	98.92%
schedule2	242	312	251	315	96.41%	95.87%	100.00%	96.79%
tcas	178	236	179	242	99.44%	97.52%	100.00%	100.00%
totinfo	253	313	267	336	95.51%	92.86%	100.00%	99.04%
Average					97.34%	95.37%	100.00%	98.77%

Table 2: Dynamic vs. FXE CFGs for the Siemens Test Suite.

As an alternative, for our evaluation we use programs with abundant inputs to closely (under-)approximate their ideal CFGs. We use the HR variants of the Siemens Test Suite [17] from the Aristotle Analysis System[3], which are also included in the Software-artifact Infrastructure Repository (SIR [13]). These programs all come with thousands of inputs and their binaries all contain indirect branches. Therefore, these programs are good candidates for our evaluation. Table 1 lists the test subjects, given for each subject its name, the number of lines of code, the number of test cases available, and binary file size.

We first compile the original version of the programs using GCC with default compiler options (optimization level zero). Then we run the executables using all the provided test cases in the same instrumented execution environment, but without forced execution, to generate dynamic CFGs. At the end, we merge all dynamic CFGs for the same program into $G_d = (V_d, E_d)$, which is an under-approximation of the ideal CFG. We also run these programs with FXE and construct $G_f = (V_f, E_f)$ for each program. In this experiment, we use a slightly different configuration for FXE so that it applies forced execution only on the part of the code that corresponds to the original program source code. This is done because the compiler generates initialization routines that are invoked when the program is started to initialize data before the `main` function is executed. The compiler also generates termination routines that are invoked when the program terminates[4]. However, these functions contain branches to handle errors and exceptions that never occur during the generation of dynamic CFGs. For a fair comparison, we do not apply forced execution on initialization and termination routines. In order to do this, we first identify the address of the `main` function for a given program. We run the program normally at first, and start forced execution at the beginning of the `main` function. We then revert to normal execution when the program reaches the termination routine.

To evaluate the soundness and completeness of FXE, we compare G_d and G_f. For soundness, we examine how many blocks and edges found by FXE are also included in the corresponding dynamic CFG. The columns $\frac{|V_d \cap V_f|}{|V_f|}$ and $\frac{|E_d \cap E_f|}{|E_f|}$ in Table 2 represent the percentage of blocks and edges in G_f which are confirmed valid by G_d. On average, G_d covers over 95% of the blocks and edges found by FXE. For those blocks and edges that are not covered, we postulate that they arise from one of the following two situations:

- The program contains code that cannot be executed by any input to the program. This usually means the program either contains dead code or code that will not execute on the current platform or in the current environment.
- The program contains code that is executed only under extremely rare conditions (*e.g.*, failure to allocate memory).

To confirm our hypothesis, we manually inspected all the additional blocks and edges. After examining the source code and the binaries, we were able to verify that all the blocks and edges not found by FXE fall into the above two categories. Although there are thousands of test cases for each test program, those inputs do not achieve 100% coverage. For each program, there is an `UNREACHABLE` file in the source directory that describes which part of the source code is not reachable. Our findings match the descriptions in these `UNREACHABLE` files.

Example 4. The following code from `tcas` gives an example of dead code. The code on the second line is unreachable because it requires two mutually contradictory expressions `need_upward_RA` and `need_downward_RA` to be true.

```
if (need_upward_RA && need_downward_RA)
  alt_sep = UNRESOLVED; // unreachable code
else if (need_upward_RA)
  alt_sep = UPWARD_RA;
else if (need_downward_RA)
  alt_sep = DOWNWARD_RA;
else
  alt_sep = UNRESOLVED;
```

[3] http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/
[4] http://gcc.gnu.org/onlinedocs/gccint/

Compiler (Option)	$\overline{\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}}$	$\overline{\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}}$	$\overline{\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}}$	$\overline{\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}}$
GCC -O0	97.34%	95.37%	100.00%	98.77%
GCC -O1	97.47%	94.76%	100.00%	98.88%
GCC -O2	97.66%	95.30%	100.00%	97.25%
GCC -O3	96.42%	91.72%	100.00%	98.36%
GCC -Os	97.53%	95.69%	100.00%	99.29%
MSVC	95.15%	89.45%	100.00%	95.26%
ICC	89.87%	81.73%	100.00%	98.15%

Table 3: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with different optimization options and different compilers.

The CFGs generated by FXE contain such dead code as illustrated in Example 4. However, we believe this is reasonable because building CFGs directly from source code also considers such code.

Example 5. The `malloc` function in the following code from the program `totinfo` almost always succeeds unless in extremely rare cases (*e.g.*, insufficient memory). The code inside the `if` conditional is never executed during dynamic CFG construction.

```
if ((xi = (double *)malloc(r *
     sizeof(double))) == NULL) {
  info = -4.0; // unreachable code
  ......
}
```

FXE also explores code typically not exercised by dynamic execution as described in Example 5. In this case, the code is actually reachable, although with very small probability. This is an advantage of forced execution over dynamic execution, as forced execution can explore code that is difficult to be covered in dynamic runs.

To evaluate the completeness of FXE, we examine how many blocks and edges in dynamic CFGs are covered by FXE. The columns $\frac{|V_f \cap V_d|}{|V_d|}$ and $\frac{|E_f \cap E_d|}{|E_d|}$ in Table 2 represent the percentage of blocks and edges in G_d which G_f covers. FXE achieves 100% block coverage for all the programs used in our experiments. On average, it covers 98.77% of the edges in dynamic CFGs. Upon close inspection, we were able to confirm that all the edges not covered by FXE are function return edges, which we explained in Section 3.3.1.

5.2 Generality

In this section, we evaluate the generality of FXE by measuring the effects of compilers on FXE's precision. Specifically, we aim to test whether FXE can successfully generate precise CFGs for programs that are compiled with different optimization options or even different compilers. For this, we use the same programs from the Siemens Test Suite.

For each test program, we first re-compile the source code using the same compiler (*i.e.*, GCC), but with different optimization options. Then we build dynamic CFGs with the same test inputs and merge them into a single CFG for each binary. Finally, we also generate CFGs with FXE, and compare them with the dynamic ones. We present the summarized results in Table 3 and provide detailed results in Appendix A.

Table 3 presents the average soundness and completeness results for binaries compiled under each optimization level of GCC. For instance, the column $\overline{\frac{|V_f \cap V_d|}{|V_d|}}$ represents the average block coverage of all seven test programs by FXE. The numbers from the first row correspond to the average numbers from the bottom of Table 2. As Table 3 shows, different compiler optimization options have negligible impact on our results, and FXE is able to consistently generate precise CFGs for all the binaries.

Besides different optimization options, we also evaluate FXE with different compilers. To this end, we re-compile the same test programs using two other popular compilers: Microsoft Visual C++ 12 (from Microsoft Visual Studio 2008) and Intel C++ Compiler Professional Edition 11.1. We then build dynamic CFGs for the binaries and compare them with the ones generated by FXE. The results are also summarized in Table 3. As the table shows, different compilers have limited impact on the precision of FXE results.

5.3 Comparison with IDA

In order to further evaluate the effectiveness and scalability of FXE, we performed experiments on CINT2000[5], the integer components of the SPEC CPU2000 benchmarks. SPECint 2000 contains twelve applications written in C or C++. We compiled these programs using GCC under MinGW[6] into Windows executables using default compiler options (optimization level zero) specified in the Makefiles. An overview of the SPECint benchmark programs is given in Table 4.

We analyzed the twelve benchmark programs with FXE and constructed CFGs from these binaries. Then we compared our results with `IDA Pro 5.0`[7], an interactive disassembler that is often considered the state-of-the-art commercial disassembler. As a disassembler, IDA Pro does not directly support the generation of CFGs. We have developed a plugin, *IDA CFG*, using the `IDA Pro SDK`, to build CFGs. IDA CFG works on top of the disassembly results from IDA Pro, and relies on IDA Pro to resolve branch targets. It starts from the program entry point, and follows the instructions until it encounters a control transfer instruction (CTI). If it is an unconditional branch, IDA CFG continues from the branch target. If the control transfer is a conditional branch, it continues at both branch targets. IDA CFG recursively follows all the paths that can be found statically by IDA Pro. In the case of indirect branches, IDA CFG is often unable to follow the targets because IDA Pro has limited capability in resolving indirect branch targets. Therefore, IDA CFG stops following the current path if the indirect branch target cannot be resolved. For call branches, IDA CFG continues from the instruction immediately following the `call` instruction. For a fair comparison, we enhance IDA CFG with the knowledge that certain library functions do not return, so that it would not follow these invalid fall-through paths. In addition, IDA CFG also stops exploring the current path upon encountering privileged instructions.

The results for FXE and a comparison of it to IDA are presented in Table 5. For each program, $G_i = (V_i, E_i)$ and $G_f = (V_f, E_f)$ represent the CFGs constructed by IDA and FXE respectively. The results show that FXE provides significantly more precise CFGs over

[5]http://www.spec.org/cpu2000/CINT2000/

[6]http://www.mingw.org/

[7]http://www.hex-rays.com/idapro/

Program	LOC	Size (KB)				
		GCC -O0	GCC -O1	GCC -O2	GCC -O3	GCC -Os
gzip	8,616	84	77	67	73	61
vpr	17,729	200	169	165	175	139
gcc	222,182	1,732	1,404	1,420	1,635	1,130
mcf	2,423	33	30	31	31	29
crafty	21,150	282	256	264	277	233
parser	11,391	142	127	132	175	112
eon	41,009	1,338	1,168	1,136	1,151	1,055
perlbmk	85,464	761	622	645	736	526
gap	71,430	662	492	484	517	398
vortex	67,220	711	644	663	683	495
bzip2	4,649	59	53	55	65	50
twolf	20,459	306	232	229	239	194

Table 4: SPECint benchmark programs.

Program	$\|V_d\|$	$\|E_d\|$	$\|V_i\|$	$\|E_i\|$	$\|V_f\|$	$\|E_f\|$	$\frac{\|V_f \cap V_i\|}{\|V_i\|}$	$\frac{\|E_f \cap E_i\|}{\|E_i\|}$	$\frac{\|V_f \setminus V_i\|}{\|V_i\|}$	$\frac{\|E_f \setminus E_i\|}{\|E_i\|}$
gzip	992	1,304	1,557	2,007	1,603	2,319	100.00%	100.00%	2.95%	15.55%
vpr	3,885	5,432	5,751	7,100	6,157	9,169	100.00%	100.00%	7.06%	29.14%
gcc	38,102	54,925	45,125	63,391	85,110	138,854	100.00%	100.00%	88.61%	119.04%
mcf	432	561	636	791	676	965	100.00%	100.00%	6.29%	22.00%
crafty	4,660	6,822	8,238	11,175	8,835	14,084	100.00%	100.00%	7.25%	26.03%
parser	4,393	6,329	5,968	7,513	6,018	8,917	100.00%	100.00%	0.84%	18.69%
eon	12,254	19,374	7,435	7,821	15,536	24,613	100.00%	100.00%	108.96%	214.70%
perlbmk	6,707	9,282	7,086	9,418	20,128	32,125	100.00%	100.00%	184.05%	241.10%
gap	7,990	12,534	4,930	6,435	32,539	53,235	100.00%	100.00%	560.02%	727.27%
vortex	11,077	14,621	16,640	22,875	19,625	30,797	100.00%	100.00%	17.94%	34.63%
bzip2	979	1,287	1,450	1,840	1,489	2,166	100.00%	100.00%	2.69%	17.72%
twolf	3,740	5,245	9,073	12,317	9,283	14,236	100.00%	100.00%	2.31%	15.58%
Average							100.00%	100.00%	82.41%	123.45%

Table 5: IDA vs. FXE CFGs for the SPECint benchmarks.

IDA, as FXE is able to discover more blocks and edges. The number of blocks and edges reported for each binary is provided in Table 5. The columns $\frac{|V_f \cap V_i|}{|V_i|}$ and $\frac{|E_f \cap E_i|}{|E_i|}$ denote the percentage of blocks and edges in the results for IDA that are covered by FXE. We observe that FXE is able to achieve 100% coverage on both blocks and edges over the results of IDA for all the programs.

Table 5 also shows the relative increase in the number of blocks and edges, using the results from IDA as the baseline. We can see that there is a greater increase in the number of edges than blocks. This is because, even though IDA Pro is able to find indirect branch blocks, it often misses the indirect branch edges when it cannot properly resolve the targets. In contrast, FXE is able to explore those indirect branch edges.

In order to further evaluate the quality of our results, we produce a dynamic CFG $G_d = (V_d, E_d)$ by running each benchmark program, and compare it against the CFGs constructed by IDA and FXE. We run these programs using the test input sets provided along with the SPEC2000 distribution. We allow the programs to run until normal termination and construct dynamic CFGs. Although each of the benchmark programs has three test input sets (*i.e.*, `test`, `train` and `ref`), we only run the benchmark programs on their smallest input sets because they do not terminate on their largest input sets within a reasonable amount of time — `mcf`, the smallest benchmark, already takes more than seven hours when running in the instrumented emulation environment. The second and third columns in Table 5 represent the numbers of blocks and edges in each dynamic CFG respectively. Table 6 shows the coverage of blocks and edges on the dynamic CFGs both by IDA and FXE. We observe that FXE achieves consistently better coverage than IDA for all the programs. In fact, for most programs, FXE is able to find all the blocks included in the corresponding dynamic CFG. Even for the programs that FXE misses a few blocks, the coverage is still far better compared to IDA. This also indicates that many of the additional blocks reported by FXE but missed by IDA are actually valid. The reason that FXE fails to find certain blocks is that some indirect branch targets may not be properly calculated due to our optimizations (*i.e.*, block skipping). In this case, the blocks at the target location and any associated edges are not explored by FXE. In addition, as discussed in Section 3.3.1, our optimization also prevents FXE from finding some of the function return edges.

One of the most significant advantages of FXE over IDA and other static analysis tools is its ability to resolve indirect branch targets. To quantify this benefit, we break down the results from Table 5 and focus on the indirect blocks and edges in the CFGs. Table 7 lists the number of indirect blocks and edges for each CFG. For example, the columns $|V'_d|$ and $|E'_d|$ represents the number of indirect blocks and edges in the dynamic CFG respectively. As the table shows, FXE is able to find all the indirect blocks and edges reported by IDA, and it also discovers many additional ones. In fact, IDA Pro cannot resolve most of the indirect branches and thus does not report any indirect edges for most of the programs. In this case, the relative increase is denoted as ∞ in the table. When we compute the average increase, we do not take into account the ∞ cases and

Program	$\frac{\lvert V_i \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_i \cap E_d \rvert}{\lvert E_d \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$	$\frac{\lvert V'_i \cap V'_d \rvert}{\lvert V'_d \rvert}$	$\frac{\lvert E'_i \cap E'_d \rvert}{\lvert E'_d \rvert}$	$\frac{\lvert V'_f \cap V'_d \rvert}{\lvert V'_d \rvert}$	$\frac{\lvert E'_f \cap E'_d \rvert}{\lvert E'_d \rvert}$
gzip	96.88%	84.89%	100.00%	97.55%	92.94%	0.00%	100.00%	80.95%
vpr	95.08%	76.38%	100.00%	95.54%	96.35%	0.00%	100.00%	77.55%
gcc	61.90%	52.11%	99.58%	93.97%	71.86%	0.00%	99.87%	66.81%
mcf	94.21%	81.64%	100.00%	99.47%	88.68%	0.00%	100.00%	96.30%
crafty	92.19%	79.73%	100.00%	97.52%	95.71%	0.00%	100.00%	81.41%
parser	99.23%	81.85%	100.00%	95.94%	97.48%	0.00%	100.00%	76.99%
eon	25.37%	14.63%	85.14%	76.80%	44.48%	0.04%	83.89%	52.50%
perlbmk	37.51%	28.01%	84.88%	78.75%	55.91%	0.14%	88.98%	58.55%
gap	29.77%	21.41%	100.00%	73.73%	17.19%	0.00%	100.00%	16.97%
vortex	89.28%	68.47%	99.99%	85.89%	87.99%	0.00%	99.84%	39.32%
bzip2	97.55%	84.30%	100.00%	97.28%	93.59%	0.00%	100.00%	80.56%
twolf	99.33%	83.74%	100.00%	98.42%	96.25%	0.00%	100.00%	90.01%
Average	76.53%	63.10%	97.47%	90.91%	78.20%	0.02%	97.72%	68.16%

Table 6: Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks.

Program	$\lvert V'_d \rvert$	$\lvert E'_d \rvert$	$\lvert V'_i \rvert$	$\lvert E'_i \rvert$	$\lvert V'_f \rvert$	$\lvert E'_f \rvert$	$\frac{\lvert V'_f \cap V'_i \rvert}{\lvert V'_i \rvert}$	$\frac{\lvert E'_f \cap E'_i \rvert}{\lvert E'_i \rvert}$	$\frac{\lvert V'_f \setminus V'_i \rvert}{\lvert V'_i \rvert}$	$\frac{\lvert E'_f \setminus E'_i \rvert}{\lvert E'_i \rvert}$
gzip	85	168	106	0	115	261	100.00%	100.00%	8.49%	∞
vpr	274	1,078	317	0	331	1,504	100.00%	100.00%	4.42%	∞
gcc	1,507	9,434	1,572	21	2,482	21,154	100.00%	100.00%	57.89%	100633.33%
mcf	53	81	67	0	76	131	100.00%	100.00%	13.43%	∞
crafty	140	909	186	0	198	2,096	100.00%	100.00%	6.45%	∞
parser	318	1,117	356	0	368	1,351	100.00%	100.00%	3.37%	∞
eon	1,086	5,672	871	18	1,384	4,741	100.00%	100.00%	58.90%	26238.89%
perlbmk	508	2,188	431	4	837	5,225	100.00%	100.00%	94.20%	130525.00%
gap	832	3,966	183	0	2,147	11,165	100.00%	100.00%	1073.22%	∞
vortex	608	3,400	619	0	739	3,934	100.00%	100.00%	19.39%	∞
bzip2	78	180	106	0	114	283	100.00%	100.00%	7.55%	∞
twolf	160	831	214	0	227	1,649	100.00%	100.00%	6.07%	∞
Average							100.00%	100.00%	112.78%	85799.07%

Table 7: IDA vs. FXE CFGs (Indirect) for the SPECint benchmarks.

Compiler Option	$\overline{\frac{\lvert V_f \cap V_i \rvert}{\lvert V_i \rvert}}$	$\overline{\frac{\lvert E_f \cap E_i \rvert}{\lvert E_i \rvert}}$	$\overline{\frac{\lvert V_f \setminus V_i \rvert}{\lvert V_i \rvert}}$	$\overline{\frac{\lvert E_f \setminus E_i \rvert}{\lvert E_i \rvert}}$	$\overline{\frac{\lvert V_i \cap V_d \rvert}{\lvert V_d \rvert}}$	$\overline{\frac{\lvert E_i \cap E_d \rvert}{\lvert E_d \rvert}}$	$\overline{\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}}$	$\overline{\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}}$
GCC -O0	100.00%	100.00%	82.41%	123.45%	76.53%	63.10%	97.47%	90.91%
GCC -O1	100.00%	99.94%	49.65%	71.24%	85.11%	70.34%	97.38%	89.76%
GCC -O2	100.00%	99.94%	30.17%	49.17%	85.60%	70.38%	95.31%	86.55%
GCC -O3	100.00%	99.98%	34.52%	54.17%	85.55%	71.86%	96.18%	88.02%
GCC -Os	100.00%	99.99%	49.43%	78.86%	85.38%	71.06%	97.47%	90.02%

Table 8: IDA vs. FXE CFGs for the SPECint benchmarks under different compiler optimization levels.

Compiler Option	$\overline{\frac{\lvert V'_f \cap V'_i \rvert}{\lvert V'_i \rvert}}$	$\overline{\frac{\lvert E'_f \cap E'_i \rvert}{\lvert E'_i \rvert}}$	$\overline{\frac{\lvert V'_f \setminus V'_i \rvert}{\lvert V'_i \rvert}}$	$\overline{\frac{\lvert E'_f \setminus E'_i \rvert}{\lvert E'_i \rvert}}$	$\overline{\frac{\lvert V'_i \cap V'_d \rvert}{\lvert V'_d \rvert}}$	$\overline{\frac{\lvert E'_i \cap E'_d \rvert}{\lvert E'_d \rvert}}$	$\overline{\frac{\lvert V'_f \cap V'_d \rvert}{\lvert V'_d \rvert}}$	$\overline{\frac{\lvert E'_f \cap E'_d \rvert}{\lvert E'_d \rvert}}$
GCC -O0	100.00%	100.00%	112.78%	85799.07%	78.20%	0.02%	97.72%	68.16%
GCC -O1	100.00%	98.04%	99.67%	4468.70%	82.47%	4.00%	97.41%	62.34%
GCC -O2	100.00%	98.19%	52.27%	3557.12%	82.43%	3.52%	93.97%	54.85%
GCC -O3	100.00%	98.91%	78.30%	4250.83%	81.50%	3.74%	94.98%	57.32%
GCC -Os	100.00%	98.11%	100.78%	9229.96%	82.91%	3.32%	96.95%	64.35%

Table 9: IDA vs. FXE CFGs (Indirect) for the SPECint benchmarks under different compiler optimization levels.

Program	Dynamic Time (s)			IDA Time (s)			FXE Time (s)		
	T_{d1}	T_{d2}	T_d	T_{i1}	T_{i2}	T_i	T_{f1}	T_{f2}	T_f
gzip	731.408	0.002	731.410	6.209	0.161	6.370	7.313	0.003	7.317
vpr	5,270.172	0.006	5,270.178	10.569	0.559	11.128	32.222	0.012	32.234
gcc	730.990	0.069	731.059	72.109	4.409	76.518	1,852.660	0.234	1,852.894
mcf	58.779	0.001	58.779	3.815	0.065	3.880	2.484	0.002	2.485
crafty	1,244.113	0.011	1,244.123	15.538	0.778	16.316	72.331	0.022	72.352
parser	1,401.539	0.009	1,401.548	9.648	0.546	10.194	32.439	0.014	32.453
eon	646.459	0.006	646.466	36.360	0.652	37.012	173.681	0.011	173.692
perlbmk	42.137	0.007	42.144	34.695	0.638	35.333	207.989	0.037	208.025
gap	371.997	0.013	372.010	31.793	0.459	32.252	303.579	0.286	303.865
vortex	2,702.597	0.006	2,702.603	32.589	1.594	34.182	203.363	0.040	203.403
bzip2	2,362.794	0.002	2,362.795	5.390	0.146	5.536	6.416	0.004	6.420
twolf	63.546	0.007	63.553	14.446	0.626	15.072	51.914	0.026	51.940

Table 10: Performance results.

only consider the concrete numbers.

Table 6 also shows the coverage of indirect blocks and edges from dynamic CFGs both by IDA and FXE. From the table, we can see that, compared to IDA, FXE has considerably better coverage on indirect blocks and drastically higher coverage on indirect edges.

For the SPECint benchmark programs, we also compile them using different optimization options when we compare our results with IDA Pro. Again, we only present the average results here. Table 8 shows the comparison on the overall CFGs, while Table 9 focuses on the indirect blocks and edges. As these tables suggest, FXE is robust to different compiler options and consistently achieves better results than IDA. For detailed results on each benchmark program, please refer to Appendix B.

In addition to performing a comparison with IDA Pro, it would have also been interesting to compare our approach to existing static analyzers that are able to generate binary CFGs such as CodeSurfer/x86 [3, 23]. However, we are not able to perform such a comparison because CodeSurfer/x86 is a proprietary platform and not publicly available. In fact, CodeSurfer/x86 relies on IDA Pro to disassemble binaries and build CFGs. It then refines the IDA Pro results to account for indirect branches using value-set analysis (VSA) [2, 24], which is a data-flow analysis that statically overapproximates the values for each variable at each program point. As such, we expect that it also suffers from the usual scalability and precision issues of static techniques.

5.4 Performance

Our goal is to construct complete CFGs for binaries. Therefore we do not set any time limit on FXE's exploration of each program path, nor do we limit its overall execution time. We expect our approach to be scalable because its analysis time is bounded by the size of the code under analysis. Table 10 shows the time for FXE and IDA to construct the CFG for each benchmark compiled using GCC with default optimization options. The analysis time for these programs under other optimization options is similar. The time given in column T_i includes the initial disassembly and analysis time by IDA Pro (T_{i1}) and the time IDA CFG takes to generate the CFG output (T_{i2}). The column T_f represents the total amount of time taken by FXE in both its execution phase (T_{f1}) and its offline analysis phase (T_{f2}). We also conducted experiments where we tried to run FXE on the same benchmark programs without certain optimizations such as block skipping. We observed a slight increase in the number of blocks and edges from the CFGs, but considerably longer analysis time. On some of the larger programs, the analysis did not terminate within reasonable amount of time. This empirically shows that the optimization techniques described in Section 3.3 can greatly improve the scalability of our analysis without any major sacrifice in the quality of the constructed CFGs. The processing times presented in the table are taken as the average of five runs on a Windows host machine with an Intel 1.6 GHz dual-core CPU and 2GB of RAM.

Table 10 also provides the time (T_d) to generate the dynamic CFG by running each program in the emulator with its test input. These numbers correspond to one dynamic run with a single test input, and they confirm that dynamic CFG generation is quite expensive. Although we may improve the efficiency by running directly on the hardware instead of on an emulator, it would still be ineffective and inefficient for the following reasons. First, generating a complete dynamic CFG requires a comprehensive set of test inputs, which is usually unavailable in practice. Therefore, dynamic CFG generation may require unbounded number of runs to approximate the complete CFG. Second, the running time for dynamic execution is not bounded by the code size, which means the analysis is inefficient, especially for programs with long-running loops. As such, though the cost of dynamic CFG generation is comparable in some cases to that of FXE in Table 10, the dynamic CFGs are far less complete than the ones generated by FXE.

During dynamic forced execution, a snapshot is taken at each branch point that includes the virtual CPU state and process memory state. In our experiments, the size of a program state is approximately 850 KB on average, while the largest one is around 2 MB. There are 222 concurrent states on average, and the maximum is 1,400. These two numbers correspond to the average and the maximum depths of the search tree in our depth-first-search based path exploration. The total number of states is 11,472 on average, with a maximum of 56,879. In terms of memory usage, the peak usage is around 350 MB, with 256 MB allocated for the guest system.

5.5 Discussions

In Section 3.1, we make a few assumptions that may lead FXE to produce imprecise or incorrect results when analyzing certain binaries. For instance, we assume that each function call returns to its original call site. However, there are situations where the call returns to a different location or does not return at all. Although we identify a few uncommon library functions and treat them as special cases, it is still possible that the analyzed code contains functions which also violate this assumption. In such cases, we might follow invalid fall-through edges after these function calls. To mitigate this problem, FXE attempts to detect invalid blocks by identifying invalid instructions when it follows these paths.

Besides our assumptions, the optimization techniques described in Section 3.3 may also lead to imprecise analysis results. On one hand, we may miss function return edges because we skip certain blocks. On the other hand, if the skipped blocks contribute to the computation of indirect branch targets (*e.g.*, in a loop), we may end up with incorrect targets and thus fail to explore certain paths. This is also the reason why FXE does not cover all the blocks in dynamic CFGs for some SPECint benchmark programs (Table 6). One possible mitigation is to increase the initial block quota at the expense of longer analysis time so that blocks have more opportunities to be executed when higher-quality CFGs are desired.

In our current implementation, we skip external code assuming that it does not influence the control flow of the program under analysis. This may also limit the ability of our analysis on certain binaries. In order to account for the influence on control flow from external functions, we can also explore those functions or use their function summaries, which we leave as future work.

Finally, in our current implementation, we rely on a simple pattern matching algorithm to detect jump tables that contain indirect branch targets. Although we can detect many jump tables in this way, a more principled analysis may help us achieve better results. We plan to incorporate such techniques [11] in our future work.

6. RELATED WORK

Constructing control flow graphs from binaries is not a new problem, and most existing solutions are based on static binary analysis. Static analyzers usually involve a disassembly process where assembly code is extracted from the binary image. Based on the result of the disassembly phase, they then reconstruct high-level program information such as control flow graphs. Static disassembly techniques generally fall into two categories: linear sweep and recursive traversal. The GNU `objdump` utility[8] uses the linear sweep algorithm. It starts disassembly from the program entry point, and scans through the entire code section. The problem with this technique is that it tries to map all bytes in the code section into instructions, which leads to errors when it misinterprets data as instructions. To circumvent data embedded in the instruction stream, recursive disassemblers [12, 27, 29] follow the program control flow by interpreting branch instructions. However, they have difficulty in resolving indirect branch targets statically, and thus may fail to analyze parts of the program. To address this issue, speculative disassembly [10] is introduced to further analyze the unreachable code regions in case they contain indirect branch targets. In order to handle indirect branches, researchers have proposed a number of techniques to complement these basic algorithms. Schwarz *et al.* [25] present a hybrid approach by combining the linear sweep and recursive traversal algorithms. Cifuentes and Emmerik [11] propose a method to analyze jump tables using backward slicing and expression substitution. De Sutter *et al.* [28] describe a method using relocation information to handle the uncertainty of indirect control flow with hell nodes and hell edges. Kruegel *et al.* [21] propose a control flow analysis and statistical methods to disassemble obfuscated binaries. Kinder *et al.* [19, 20] reconstruct an over-approximation of control flow graph based on abstract interpretation by combining control and data flow analysis. Due to the inherent difficulty of static analysis to resolve indirect branch targets, these approaches, in general, cannot produce precise control flow information. Our approach, however, is based on dynamic (forced) execution, and thus does not have such problems with indirect branches.

Traditional dynamic analysis techniques cover only a small portion of program execution paths. To improve code coverage, Moser *et al.* [22] propose a dynamic taint analysis to explore multiple execution paths for malware analysis. It generates linear constraints at branch points where control flow decisions are based on tainted data. Their technique still suffers from poor path coverage in the presence of unsolvable constraints. In contrast, we explore program paths at all branch points, bypassing the computational limitation of constraint solvers. In this way, our approach covers more execution paths and scales better to large programs. For identifying rootkits, Limbo [30] uses forced sampled execution, a technique similar to our forced execution, to expose the execution behavior of kernel drivers. While the goal of Limbo is to detect kernel rootkits, ours is to construct precise CFGs. In order to traverse the control flow graph, Limbo applies flood emulation, which is a context-independent sampling approach. It statically sets a quota for each basic block, and also limits the total number of emulated instructions. Limbo ignores the block execution context and may miss program paths when the per-block quota is exhausted. Our approach performs an intraprocedural analysis to dynamically adjust the quota according to the execution context, so that we can identify more indirect branch paths.

[8] http://www.gnu.org/software/binutils/manual/

7. CONCLUSION

In this paper, 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. To make our basic technique scalable to large programs, we have also devised effective optimizations based on intraprocedural analysis to avoid unnecessary code re-execution. We have developed FXE, a prototype implementation of our technique, and conducted extensive evaluation on its effectiveness. Our experimental results show that FXE is both effective and scalable. FXE is able to construct nearly ideal control flow information and scales to large binaries. We believe that it provides a promising foundation for effective analysis of binaries. For future work, we plan to explore a number of interesting application domains of the technique, such as defect and vulnerability detection, software similarity analysis, and software piracy detection.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *Proceedings of International Conference on Compiler Construction*, pages 5–23, 2004.

[3] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 - A platform for Analyzing x86 Executables. In *Proceedings of International Conference on Compiler Construction*, pages 250–254, 2005.

[4] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Proceedings of International Conference on Verified Software: Theories, Tools and Experiments*, pages 202–213, 2007.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of USENIX Annual Technical Conference*, pages 41–46, 2005.

[6] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static Detection of Malicious Code in Executable Programs. *Int. J. of Req. Eng*, 2001.

[7] F. Besson, T. Jensen, D. L. Metayer, and T. Thorn. Model Checking Security Properties of Control Flow Graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 243–257, 2006.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[10] C. Cifuentes and M. V. Emmerik. UQBT: Adaptive Binary Translation at Low Cost. *IEEE Computer*, 33(3):60–66, 2000.

[11] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Science of Computer Programming*, 40(2–3):171–188, 2001.

[12] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[13] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of Programming Language Design and Implementation*, pages 213–223, 2005.

[15] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 53–62, 1998.

[16] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proceedings of SPIN Workshop on Model Checking Software*, pages 235–239, 2003.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of International Conference on Software Engineering*, pages 191–200, 1994.

[18] J. Jurjens and M. Yampolskiy. Code Security Analysis with Assertions. In *Proceedings of International Conference on Automated Software Engineering*, pages 392–395, 2005.

[19] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of International Conference on Computer Aided Verification*, pages 423–427, 2008.

[20] J. Kinder, F. Zuleger, and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 214–228, 2009.

[21] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of USENIX Security Symposium*, pages 255–270, 2004.

[22] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of IEEE Symposium on Security & Privacy*, pages 231–245, 2007.

[23] T. Reps and G. Balakrishnan. A Next-Generation Platform for Analyzing Executables. In *Proceedings of ASIAN Symposium on Programming Languages and Systems*, pages 212–229, 2005.

[24] T. Reps and G. Balakrishnan. Improved Memory-Access Analysis for x86 Executables. In *Proceedings of International Conference on Compiler Construction*, pages 16–35, 2008.

[25] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proceedings of Working Conference on Reverse Engineering*, pages 45–54, 2002.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, 2005.

[27] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary Translation. *Digital Technical Journal*, 4 (4), 1992.

[28] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1013–1019, 2000.

[29] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 23–30, 2000.

[30] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of Recent Advances in Intrusion Detection Symposium*, pages 219–235, 2007.

[31] X. Zhang, N. Gupta, and R. Gupta. Locating Faults Through Automated Predicate Switching. In *Proceedings of International Conference on Software Engineering*, pages 272–281, 2006.

APPENDIX

A. DETAILED RESULTS FOR THE SIEMENS TEST SUITE

In this section, we provide detailed generality evaluation results for FXE on the Siemens Test Suite programs. Table 11 through Table 14 present the soundness and completeness results for each program in the Siemens Test Suite compiled with different optimization levels of GCC (*i.e.*, -O1, -O2, -O3, and -Os). Table 15 and Table 16 show the results for each program compiled with Microsoft Visual C++ and Intel C++ Compiler respectively. These tables indicate that our technique is general and FXE is robust against different compilers and compiler options.

B. DETAILED RESULTS FOR THE SPECINT BENCHMARKS

In this section, we present detailed comparison results of IDA and FXE on the SPECint benchmarks binaries compiled under different compiler optimization options. Table 17 through Table 20 provide the comparison of overall CFGs for binaries compiled with different optimization levels of GCC. Table 21 through Table 24 focus on the comparison of indirect blocks and edges in the CFGs. Finally, Table 25 through Table 28 show the coverage of dynamic CFGs (overall for the left part and indirect for the right part) both by IDA and FXE under different optimization levels. These tables indicate that FXE is able to find almost all the blocks and edges found by IDA. We can also see that, when compared to the dynamic CFGs, FXE achieves consistently far better results than IDA on all the benchmark programs.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}$	$\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$
printtokens	287	410	293	425	98.29%	94.82%	100.00%	98.05%
printtokens2	310	437	312	440	99.36%	98.41%	100.00%	99.08%
replace	302	410	323	458	93.81%	88.65%	100.00%	98.78%
schedule	206	266	208	271	100.00%	98.89%	100.00%	100.00%
schedule2	225	295	233	300	96.57%	95.67%	100.00%	97.29%
tcas	168	222	168	227	100.00%	97.80%	100.00%	100.00%
totinfo	228	288	243	321	94.24%	89.10%	100.00%	98.96%
Average					97.47%	94.76%	100.00%	98.88%

Table 11: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -O1.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}$	$\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$
printtokens	249	346	253	348	98.81%	95.40%	100.00%	95.66%
printtokens2	260	358	260	355	100.00%	99.72%	100.00%	98.88%
replace	294	393	315	430	93.65%	88.60%	100.00%	96.69%
schedule	202	267	203	260	100.00%	99.23%	100.00%	96.25%
schedule2	224	298	237	300	94.94%	94.67%	100.00%	94.97%
tcas	167	219	167	222	100.00%	98.65%	100.00%	100.00%
totinfo	228	290	240	317	96.25%	90.85%	100.00%	98.28%
Average					97.66%	95.30%	100.00%	97.25%

Table 12: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -O2.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}$	$\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$
printtokens	287	387	302	444	97.02%	87.61%	100.00%	98.97%
printtokens2	247	338	248	338	99.60%	99.41%	100.00%	99.41%
replace	306	415	334	483	92.22%	85.92%	100.00%	99.52%
schedule	199	263	207	273	98.55%	95.24%	100.00%	96.96%
schedule2	269	368	301	421	91.36%	84.56%	100.00%	95.38%
tcas	150	195	150	198	100.00%	98.48%	100.00%	100.00%
totinfo	226	289	238	316	96.22%	90.82%	100.00%	98.27%
Average					96.42%	91.72%	100.00%	98.36%

Table 13: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -O3.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}$	$\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$
printtokens	249	342	254	358	98.43%	95.53%	100.00%	99.71%
printtokens2	240	333	240	330	100.00%	100.00%	100.00%	99.10%
replace	302	396	325	436	93.23%	89.91%	100.00%	98.74%
schedule	202	254	204	256	100.00%	99.22%	100.00%	99.21%
schedule2	218	284	231	299	94.81%	94.65%	100.00%	99.30%
tcas	162	214	162	217	100.00%	98.62%	100.00%	100.00%
totinfo	228	283	240	308	96.25%	91.88%	100.00%	98.94%
Average					97.53%	95.69%	100.00%	99.29%

Table 14: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -Os.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}$	$\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$
printtokens	222	306	235	319	97.02%	90.60%	100.00%	92.48%
printtokens2	312	425	334	475	97.60%	87.79%	100.00%	94.82%
replace	238	340	258	381	93.02%	86.61%	100.00%	96.47%
schedule	157	212	168	230	93.45%	88.26%	100.00%	95.75%
schedule2	187	269	206	294	90.78%	85.37%	100.00%	93.31%
tcas	87	124	87	126	100.00%	98.41%	100.00%	100.00%
totinfo	160	216	171	229	94.15%	89.08%	100.00%	93.98%
Average					95.15%	89.45%	100.00%	95.26%

Table 15: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with MS Visual C++.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_d \cap V_f \rvert}{\lvert V_f \rvert}$	$\frac{\lvert E_d \cap E_f \rvert}{\lvert E_f \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$
printtokens	343	494	421	643	83.14%	72.94%	100.00%	93.52%
printtokens2	546	715	672	1,010	87.50%	73.86%	100.00%	98.46%
replace	294	394	341	505	88.27%	79.01%	100.00%	99.75%
schedule	195	253	225	305	90.22%	83.93%	100.00%	99.60%
schedule2	283	407	303	455	95.05%	87.25%	100.00%	96.56%
tcas	111	145	111	147	100.00%	98.64%	100.00%	100.00%
totinfo	261	349	325	472	84.92%	76.48%	100.00%	99.14%
Average					89.87%	81.73%	100.00%	98.15%

Table 16: Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with Intel C++ Compiler.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_i \rvert$	$\lvert E_i \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_f \cap V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \cap E_i \rvert}{\lvert E_i \rvert}$	$\frac{\lvert V_f \setminus V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \setminus E_i \rvert}{\lvert E_i \rvert}$
gzip	890	1,209	1,430	1,950	1,474	2,245	100.00%	100.00%	3.08%	15.13%
vpr	3,460	4,971	5,582	7,432	5,672	8,850	100.00%	100.00%	1.61%	19.08%
gcc	33,675	49,984	71,130	105,135	74,165	120,791	100.00%	99.98%	4.27%	14.89%
mcf	394	521	577	762	617	927	100.00%	100.00%	6.93%	21.65%
crafty	4,219	6,385	8,075	11,588	8,121	13,450	100.00%	99.88%	0.57%	16.07%
parser	3,854	5,817	5,327	7,311	5,377	8,607	100.00%	100.00%	0.94%	17.73%
eon	8,704	13,492	6,102	6,912	11,818	16,654	100.00%	100.00%	93.67%	140.94%
perlbmk	6,249	8,746	15,159	22,573	18,479	29,914	100.00%	99.52%	21.90%	32.52%
gap	7,004	11,604	4,604	6,430	25,574	40,703	100.00%	99.98%	455.47%	533.02%
vortex	10,463	13,908	16,842	24,024	17,484	27,699	100.00%	99.97%	3.81%	15.30%
bzip2	847	1,163	1,277	1,730	1,316	2,038	100.00%	100.00%	3.05%	17.80%
twolf	3,370	4,843	8,235	12,220	8,275	13,534	100.00%	100.00%	0.49%	10.75%
Average							100.00%	99.94%	49.65%	71.24%

Table 17: IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -O1.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_i \rvert$	$\lvert E_i \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_f \cap V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \cap E_i \rvert}{\lvert E_i \rvert}$	$\frac{\lvert V_f \setminus V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \setminus E_i \rvert}{\lvert E_i \rvert}$
gzip	899	1,227	1,495	2,010	1,539	2,266	100.00%	100.00%	2.94%	12.74%
vpr	3,467	4,994	5,513	7,415	5,552	8,585	100.00%	100.00%	0.71%	15.78%
gcc	33,826	50,404	70,673	102,266	72,455	117,001	100.00%	99.97%	2.52%	14.41%
mcf	423	549	615	782	655	951	100.00%	100.00%	6.50%	21.61%
crafty	4,345	6,805	8,380	11,773	8,420	13,645	100.00%	99.88%	0.48%	15.90%
parser	4,045	5,980	5,679	7,472	5,720	8,777	100.00%	100.00%	0.72%	17.47%
eon	7,148	9,975	5,828	6,769	10,266	14,895	100.00%	100.00%	76.15%	120.05%
perlbmk	6,276	8,763	15,797	22,772	28,708	44,112	100.00%	99.41%	81.73%	93.71%
gap	7,124	11,851	4,779	6,421	13,543	21,713	100.00%	100.00%	183.39%	238.16%
vortex	10,752	14,118	18,624	25,319	19,307	29,044	100.00%	100.00%	3.67%	14.71%
bzip2	893	1,224	1,392	1,841	1,431	2,154	100.00%	100.00%	2.80%	17.00%
twolf	3,627	5,114	8,656	12,501	8,696	13,559	100.00%	100.00%	0.46%	8.46%
Average							100.00%	99.94%	30.17%	49.17%

Table 18: IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -O2.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_i \rvert$	$\lvert E_i \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_f \cap V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \cap E_i \rvert}{\lvert E_i \rvert}$	$\frac{\lvert V_f \setminus V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \setminus E_i \rvert}{\lvert E_i \rvert}$
gzip	953	1,309	1,620	2,210	1,769	2,627	100.00%	100.00%	9.20%	18.87%
vpr	3,652	5,199	6,040	8,357	6,079	9,508	100.00%	100.00%	0.65%	13.77%
gcc	37,447	55,018	84,439	124,221	87,188	140,938	100.00%	99.96%	3.26%	13.46%
mcf	432	567	618	802	658	965	100.00%	100.00%	6.47%	20.32%
crafty	4,604	7,252	8,730	12,325	8,770	14,401	100.00%	99.89%	0.46%	16.84%
parser	4,881	7,310	7,353	10,408	7,394	11,751	100.00%	100.00%	0.56%	12.90%
eon	7,177	10,014	5,811	6,769	10,266	14,912	100.00%	100.00%	76.66%	120.30%
perlbmk	6,709	9,221	19,457	28,746	35,647	55,212	100.00%	99.89%	83.21%	92.07%
gap	7,430	12,353	5,300	7,223	17,340	29,047	100.00%	100.00%	227.17%	302.15%
vortex	10,825	14,202	18,994	25,953	19,648	29,511	100.00%	100.00%	3.44%	13.71%
bzip2	921	1,254	1,465	2,032	1,504	2,364	100.00%	100.00%	2.66%	16.34%
twolf	3,649	5,129	8,855	12,839	8,895	14,036	100.00%	100.00%	0.45%	9.32%
Average							100.00%	99.98%	34.52%	54.17%

Table 19: IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -O3.

Program	$\lvert V_d \rvert$	$\lvert E_d \rvert$	$\lvert V_i \rvert$	$\lvert E_i \rvert$	$\lvert V_f \rvert$	$\lvert E_f \rvert$	$\frac{\lvert V_f \cap V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \cap E_i \rvert}{\lvert E_i \rvert}$	$\frac{\lvert V_f \setminus V_i \rvert}{\lvert V_i \rvert}$	$\frac{\lvert E_f \setminus E_i \rvert}{\lvert E_i \rvert}$
gzip	896	1,198	1,443	1,893	1,487	2,169	100.00%	100.00%	3.05%	14.58%
vpr	3,442	4,939	5,375	7,046	5,414	8,301	100.00%	99.91%	0.73%	17.81%
gcc	34,030	49,977	69,332	99,999	71,696	114,082	100.00%	99.98%	3.41%	14.08%
mcf	409	525	595	739	635	910	100.00%	100.00%	6.72%	23.14%
crafty	4,253	6,283	8,042	11,153	8,089	13,036	100.00%	100.00%	0.58%	16.88%
parser	4,045	5,802	5,609	7,196	5,657	8,446	100.00%	100.00%	0.86%	17.37%
eon	7,803	11,244	6,020	6,932	10,608	15,589	100.00%	100.00%	76.21%	124.88%
perlbmk	6,437	8,940	15,362	21,711	27,009	41,357	100.00%	99.99%	75.82%	90.49%
gap	6,904	11,333	4,634	6,154	24,077	42,095	100.00%	100.00%	419.57%	584.03%
vortex	10,515	13,937	18,140	25,006	18,643	29,051	100.00%	99.98%	2.77%	16.18%
bzip2	875	1,172	1,358	1,756	1,398	2,044	100.00%	100.00%	2.95%	16.40%
twolf	3,537	5,000	8,239	11,607	8,279	12,821	100.00%	100.00%	0.49%	10.46%
Average							100.00%	99.99%	49.43%	78.86%

Table 20: IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -Os.

Program	$\lvert V'_d\rvert$	$\lvert E'_d\rvert$	$\lvert V'_i\rvert$	$\lvert E'_i\rvert$	$\lvert V'_f\rvert$	$\lvert E'_f\rvert$	$\frac{\lvert V'_f \cap V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \cap E'_i\rvert}{\lvert E'_i\rvert}$	$\frac{\lvert V'_f \setminus V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \setminus E'_i\rvert}{\lvert E'_i\rvert}$
gzip	85	166	106	0	115	247	100.00%	100.00%	8.49%	∞
vpr	276	1,046	324	29	333	1,329	100.00%	100.00%	2.78%	4482.76%
gcc	1,487	9,221	2,183	2,598	2,436	14,508	100.00%	99.00%	11.59%	458.43%
mcf	56	81	69	0	78	122	100.00%	100.00%	13.04%	∞
crafty	140	843	189	132	199	1,943	100.00%	89.39%	5.29%	1371.97%
parser	313	1,024	352	0	364	1,243	100.00%	100.00%	3.41%	∞
eon	876	3,106	602	23	1,082	2,206	100.00%	100.00%	79.73%	9491.30%
perlbmk	506	2,066	649	1,497	840	4,329	100.00%	92.72%	29.43%	189.18%
gap	829	3,955	185	46	2,076	4,841	100.00%	100.00%	1022.16%	10423.91%
vortex	607	3,314	684	153	742	2,992	100.00%	95.42%	8.48%	1855.56%
bzip2	79	179	107	0	115	265	100.00%	100.00%	7.48%	∞
twolf	161	741	219	17	228	1,288	100.00%	100.00%	4.11%	7476.47%
Average							100.00%	98.04%	99.67%	4468.70%

Table 21: IDA vs. FXE CFGs (Indirect) for the SPECint benchmarks compiled with GCC -O1.

Program	$\lvert V'_d\rvert$	$\lvert E'_d\rvert$	$\lvert V'_i\rvert$	$\lvert E'_i\rvert$	$\lvert V'_f\rvert$	$\lvert E'_f\rvert$	$\frac{\lvert V'_f \cap V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \cap E'_i\rvert}{\lvert E'_i\rvert}$	$\frac{\lvert V'_f \setminus V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \setminus E'_i\rvert}{\lvert E'_i\rvert}$
gzip	82	173	106	0	116	209	100.00%	100.00%	9.43%	∞
vpr	207	981	271	29	279	1,156	100.00%	100.00%	2.95%	3886.21%
gcc	1,601	9,858	2,564	2,520	2,743	15,157	99.96%	98.65%	6.98%	501.47%
mcf	59	84	77	0	86	126	100.00%	100.00%	11.69%	∞
crafty	171	1,228	243	130	252	1,959	100.00%	89.23%	3.70%	1406.92%
parser	342	1,106	414	0	424	1,262	100.00%	100.00%	2.42%	∞
eon	703	2,764	558	23	869	1,625	100.00%	100.00%	55.73%	6965.22%
perlbmk	501	2,075	748	1,399	1,550	5,517	100.00%	90.35%	107.22%	294.35%
gap	907	4,210	216	45	1,098	3,381	100.00%	100.00%	408.33%	7413.33%
vortex	624	3,282	1,095	143	1,178	3,030	100.00%	100.00%	7.58%	2018.88%
bzip2	75	194	103	0	111	270	100.00%	100.00%	7.77%	∞
twolf	179	768	262	17	271	1,032	100.00%	100.00%	3.44%	5970.59%
Average							100.00%	98.19%	52.27%	3557.12%

Table 22: IDA vs. FXE CFGs (Indirect) for the SPECint benchmarks compiled with GCC -O2.

Program	$\lvert V'_d\rvert$	$\lvert E'_d\rvert$	$\lvert V'_i\rvert$	$\lvert E'_i\rvert$	$\lvert V'_f\rvert$	$\lvert E'_f\rvert$	$\frac{\lvert V'_f \cap V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \cap E'_i\rvert}{\lvert E'_i\rvert}$	$\frac{\lvert V'_f \setminus V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \setminus E'_i\rvert}{\lvert E'_i\rvert}$
gzip	71	157	96	0	109	224	100.00%	100.00%	13.54%	∞
vpr	171	904	217	29	225	1,137	100.00%	100.00%	3.69%	3820.69%
gcc	1,459	9,894	2,363	2,765	2,583	16,078	99.96%	98.59%	9.31%	481.48%
mcf	55	78	71	0	80	120	100.00%	100.00%	12.68%	∞
crafty	169	1,362	238	148	247	2,181	100.00%	90.54%	3.78%	1373.65%
parser	208	1,022	258	0	268	1,300	100.00%	100.00%	3.88%	∞
eon	696	2,744	549	23	856	1,611	100.00%	100.00%	55.92%	6904.35%
perlbmk	428	1,931	679	1,413	1,552	5,712	100.00%	97.74%	128.57%	304.25%
gap	902	4,309	200	45	1,571	5,628	100.00%	100.00%	685.50%	12406.67%
vortex	593	3,266	1,032	143	1,113	2,899	100.00%	100.00%	7.85%	1927.27%
bzip2	54	157	73	0	81	289	100.00%	100.00%	10.96%	∞
twolf	159	719	233	17	242	1,171	100.00%	100.00%	3.86%	6788.24%
Average							100.00%	98.91%	78.30%	4250.83%

Table 23: IDA vs. FXE CFGs (Indirect) for the SPECint benchmarks compiled with GCC -O3.

Program	$\lvert V'_d\rvert$	$\lvert E'_d\rvert$	$\lvert V'_i\rvert$	$\lvert E'_i\rvert$	$\lvert V'_f\rvert$	$\lvert E'_f\rvert$	$\frac{\lvert V'_f \cap V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \cap E'_i\rvert}{\lvert E'_i\rvert}$	$\frac{\lvert V'_f \setminus V'_i\rvert}{\lvert V'_i\rvert}$	$\frac{\lvert E'_f \setminus E'_i\rvert}{\lvert E'_i\rvert}$
gzip	76	149	94	0	103	228	100.00%	100.00%	9.57%	∞
vpr	183	972	223	35	231	1,247	100.00%	82.86%	3.59%	3462.86%
gcc	1,246	8,410	1,809	1,612	1,984	12,684	99.94%	98.76%	9.67%	686.85%
mcf	55	79	69	0	78	128	100.00%	100.00%	13.04%	∞
crafty	139	850	189	115	199	1,946	100.00%	100.00%	5.29%	1592.17%
parser	303	990	338	0	350	1,199	100.00%	100.00%	3.55%	∞
eon	564	3,213	519	23	697	2,008	100.00%	100.00%	34.30%	8630.43%
perlbmk	433	1,967	555	816	1,079	5,058	100.00%	99.75%	94.41%	519.85%
gap	806	3,844	180	20	2,007	9,969	100.00%	100.00%	1015.00%	49745.00%
vortex	600	3,311	675	147	725	3,549	100.00%	95.92%	7.41%	2314.29%
bzip2	72	165	96	0	105	245	100.00%	100.00%	9.38%	∞
twolf	158	794	215	17	224	1,188	100.00%	100.00%	4.19%	6888.24%
Average							100.00%	98.11%	100.78%	9229.96%

Table 24: IDA vs. FXE CFGs (Indirect) for the SPECint benchmarks compiled with GCC -Os.

Program	$\frac{\lvert V_i \cap V_d\rvert}{\lvert V_d\rvert}$	$\frac{\lvert E_i \cap E_d\rvert}{\lvert E_d\rvert}$	$\frac{\lvert V_f \cap V_d\rvert}{\lvert V_d\rvert}$	$\frac{\lvert E_f \cap E_d\rvert}{\lvert E_d\rvert}$	$\frac{\lvert V'_i \cap V'_d\rvert}{\lvert V'_d\rvert}$	$\frac{\lvert E'_i \cap E'_d\rvert}{\lvert E'_d\rvert}$	$\frac{\lvert V'_f \cap V'_d\rvert}{\lvert V'_d\rvert}$	$\frac{\lvert E'_f \cap E'_d\rvert}{\lvert E'_d\rvert}$
gzip	96.74%	84.04%	100.00%	97.11%	92.94%	0.00%	100.00%	78.92%
vpr	99.28%	78.98%	100.00%	93.48%	97.83%	2.20%	100.00%	69.02%
gcc	97.42%	81.80%	99.60%	92.74%	94.49%	11.78%	99.87%	62.35%
mcf	93.65%	80.23%	100.00%	98.46%	89.29%	0.00%	100.00%	90.12%
crafty	99.41%	88.07%	100.00%	97.53%	95.71%	12.22%	100.00%	81.26%
parser	99.12%	81.85%	100.00%	95.58%	97.44%	0.00%	100.00%	74.90%
eon	27.67%	17.04%	86.19%	77.10%	37.33%	0.06%	82.53%	39.50%
perlbmk	80.99%	66.61%	85.04%	78.80%	81.62%	17.91%	89.13%	57.89%
gap	31.92%	22.42%	97.74%	71.00%	17.37%	0.83%	97.35%	18.94%
vortex	98.66%	76.01%	100.00%	84.40%	95.72%	3.02%	100.00%	34.55%
bzip2	97.17%	82.72%	100.00%	95.87%	93.67%	0.00%	100.00%	73.18%
twolf	99.26%	84.25%	100.00%	95.02%	96.27%	0.00%	100.00%	67.48%
Average	85.11%	70.34%	97.38%	89.76%	82.47%	4.00%	97.41%	62.34%

Table 25: Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O1.

Program	$\frac{\lvert V_i \cap V_d\rvert}{\lvert V_d\rvert}$	$\frac{\lvert E_i \cap E_d\rvert}{\lvert E_d\rvert}$	$\frac{\lvert V_f \cap V_d\rvert}{\lvert V_d\rvert}$	$\frac{\lvert E_f \cap E_d\rvert}{\lvert E_d\rvert}$	$\frac{\lvert V'_i \cap V'_d\rvert}{\lvert V'_d\rvert}$	$\frac{\lvert E'_i \cap E'_d\rvert}{\lvert E'_d\rvert}$	$\frac{\lvert V'_f \cap V'_d\rvert}{\lvert V'_d\rvert}$	$\frac{\lvert E'_f \cap E'_d\rvert}{\lvert E'_d\rvert}$
gzip	96.77%	83.78%	100.00%	95.44%	91.46%	0.00%	100.00%	67.63%
vpr	99.28%	80.38%	100.00%	93.47%	97.10%	2.34%	100.00%	66.77%
gcc	97.39%	80.67%	98.89%	90.41%	93.94%	10.88%	97.69%	55.43%
mcf	94.09%	80.69%	100.00%	98.36%	89.83%	0.00%	100.00%	89.29%
crafty	99.42%	83.12%	100.00%	91.96%	96.49%	8.22%	100.00%	55.46%
parser	99.21%	81.00%	99.83%	94.25%	97.66%	0.00%	99.42%	69.62%
eon	32.75%	22.98%	81.28%	65.80%	40.26%	0.07%	76.96%	25.33%
perlbmk	81.58%	66.86%	92.59%	83.02%	79.44%	17.01%	93.41%	52.92%
gap	31.36%	21.65%	71.32%	52.62%	17.20%	0.78%	60.42%	13.44%
vortex	98.68%	76.56%	99.85%	83.85%	95.83%	2.99%	99.68%	31.02%
bzip2	97.31%	82.35%	100.00%	95.18%	93.33%	0.00%	100.00%	69.59%
twolf	99.31%	84.55%	100.00%	94.25%	96.65%	0.00%	100.00%	61.72%
Average	85.60%	70.38%	95.31%	86.55%	82.43%	3.52%	93.97%	54.85%

Table 26: Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O2.

Program	$\frac{\lvert V_i \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_i \cap E_d \rvert}{\lvert E_d \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$	$\frac{\lvert V'_i \cap V'_d \rvert}{\lvert V'_d \rvert}$	$\frac{\lvert E'_i \cap E'_d \rvert}{\lvert E'_d \rvert}$	$\frac{\lvert V'_f \cap V'_d \rvert}{\lvert V'_d \rvert}$	$\frac{\lvert E'_f \cap E'_d \rvert}{\lvert E'_d \rvert}$
gzip	96.96%	86.02%	100.00%	96.49%	90.14%	0.00%	100.00%	70.70%
vpr	99.32%	82.63%	100.00%	94.71%	96.49%	2.54%	100.00%	69.58%
gcc	97.38%	82.14%	98.82%	90.88%	93.21%	11.53%	97.60%	54.72%
mcf	94.21%	82.36%	100.00%	98.59%	89.09%	0.00%	100.00%	89.74%
crafty	99.46%	82.53%	100.00%	92.90%	96.45%	8.59%	100.00%	62.19%
parser	99.34%	85.61%	99.86%	94.88%	96.15%	0.00%	99.04%	64.19%
eon	32.28%	22.74%	81.20%	65.95%	39.22%	0.07%	76.58%	25.26%
perlbmk	81.41%	68.69%	98.88%	89.20%	78.97%	18.38%	97.20%	52.05%
gap	30.86%	21.47%	75.59%	54.62%	15.74%	0.77%	69.62%	12.93%
vortex	98.69%	76.81%	99.85%	83.80%	95.62%	3.00%	99.66%	30.04%
bzip2	97.39%	85.73%	100.00%	97.13%	90.74%	0.00%	100.00%	77.07%
twolf	99.31%	85.55%	100.00%	97.11%	96.23%	0.00%	100.00%	79.42%
Average	85.55%	71.86%	96.18%	88.02%	81.50%	3.74%	94.98%	57.32%

Table 27: Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O3.

Program	$\frac{\lvert V_i \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_i \cap E_d \rvert}{\lvert E_d \rvert}$	$\frac{\lvert V_f \cap V_d \rvert}{\lvert V_d \rvert}$	$\frac{\lvert E_f \cap E_d \rvert}{\lvert E_d \rvert}$	$\frac{\lvert V'_i \cap V'_d \rvert}{\lvert V'_d \rvert}$	$\frac{\lvert E'_i \cap E'_d \rvert}{\lvert E'_d \rvert}$	$\frac{\lvert V'_f \cap V'_d \rvert}{\lvert V'_d \rvert}$	$\frac{\lvert E'_f \cap E'_d \rvert}{\lvert E'_d \rvert}$
gzip	96.76%	85.31%	100.00%	97.41%	92.11%	0.00%	100.00%	79.19%
vpr	99.27%	80.34%	100.00%	95.46%	96.72%	2.37%	100.00%	76.95%
gcc	97.19%	82.30%	99.74%	93.46%	93.66%	7.61%	99.84%	62.44%
mcf	93.89%	80.76%	100.00%	99.62%	89.09%	0.00%	100.00%	97.47%
crafty	99.41%	87.73%	100.00%	97.33%	95.68%	11.88%	100.00%	80.24%
parser	99.21%	82.42%	100.00%	95.95%	97.36%	0.00%	100.00%	76.26%
eon	30.58%	20.78%	81.48%	68.13%	48.23%	0.06%	72.70%	33.92%
perlbmk	80.47%	66.49%	90.79%	84.07%	81.29%	14.79%	94.46%	61.46%
gap	32.68%	22.88%	97.81%	71.40%	17.25%	0.23%	96.77%	19.48%
vortex	98.65%	76.04%	99.83%	85.74%	95.67%	2.90%	99.67%	40.44%
bzip2	97.14%	84.04%	100.00%	95.90%	91.67%	0.00%	100.00%	70.91%
twolf	99.29%	83.68%	100.00%	95.78%	96.20%	0.00%	100.00%	73.43%
Average	85.38%	71.06%	97.47%	90.02%	82.91%	3.32%	96.95%	64.35%

Table 28: Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -Os.