

Constructing Precise Control Flow Graphs from Binaries

Liang Xu Fangqi Sun Zhendong Su

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

Abstract

Third-party software is often distributed only in binary form. Precise and scalable binary analysis is an enabling technology for many software engineering and security techniques such as program verification, bug and vulnerability detection, and automated test case generation. One fundamental obstacle to performing binary analysis is the lack of precise control flow information. Existing techniques to construct the control flow of binaries are either static or dynamic. Traditional static techniques 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 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 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 effectiveness. 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 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 [30], and generate test cases [15] automatically.

Although effective techniques exist to build precise CFGs from source code, none constructs reliable CFGs from binaries because of various difficulties introduced by low-level features of machine code. However, such a technique 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 analyze the structure of binaries and parse instruction opcodes. They generally have good code coverage, 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 a binary in a finite set of runs, and are unable to guarantee code coverage. They also have poor scalability in the presence of long-running loops. Recently, hybrid techniques based on concolic testing [8, 9, 14, 24] 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 targets remain unexplored. Second, similar to dynamic CFG construction, they suffer from ineffective treatment of loops. CUTE [24] 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 binary CFG construction. For example, on even small test programs from the Siemens Test Suite [17], they either failed to achieve good coverage or did not terminate in a reasonable amount of time.

In this paper, we introduce a novel, effective technique that automatically generates precise CFGs from binaries. Our approach is motivated by the observation that indirect branch targets 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, our key insight is to use *forced execution* to explore both directions of every conditional branch. In particular, we dynamically track conditional branch instructions and identify each branch point where we save the current program state, before allowing the 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 CPU instruction pointer. In this way, we efficiently explore both directions at each branch point and resolve indirect branches dynamically to construct precise control flow graphs.

We have developed a prototype named FXE (Forced eXecution Engine) based on the QEMU [5] emulator. FXE works on x86 executables in the Windows PE format. For path exploration, we have implemented a mechanism to save and restore snapshots for an executing process, and also force the execution to continue from a

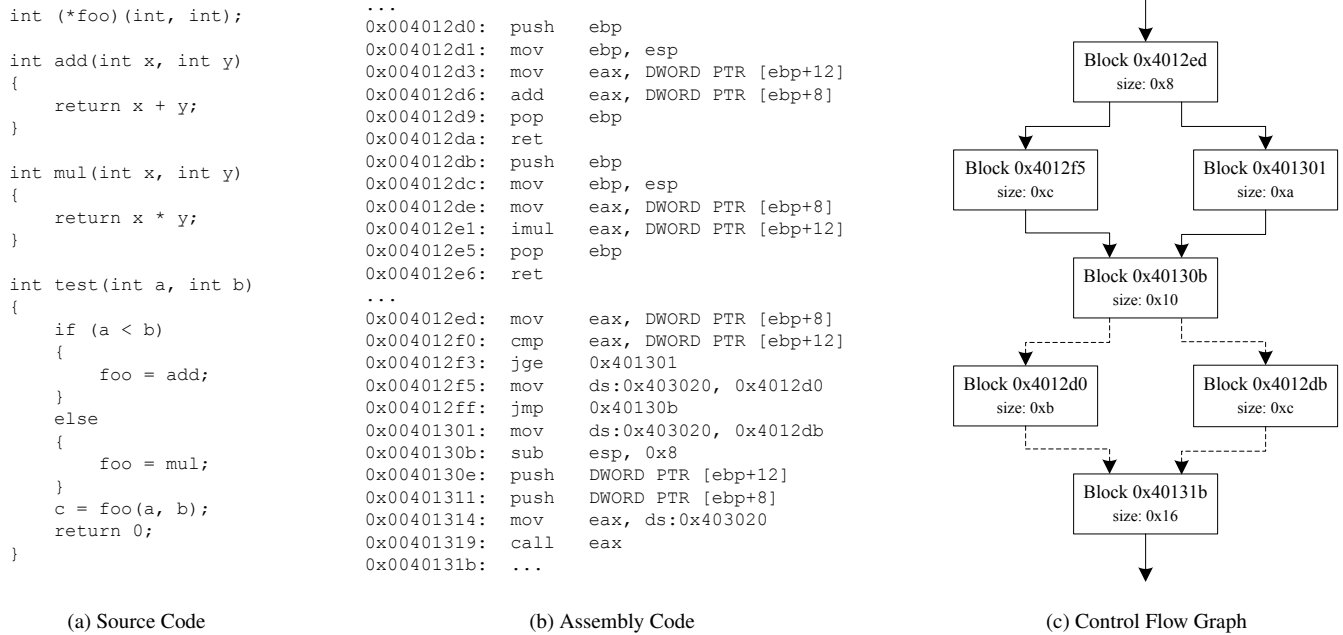


Figure 1. An Example of CFG Construction.

specific program point. To achieve 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 compiler optimization options, and scales to large programs. In particular, we show that FXE produces nearly ideal CFGs and substantially outperforms state-of-the-art alternatives such as the popular commercial disassembler IDA Pro. We believe our technique is a promising enabling technology for automated analysis of binaries.

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

2. Illustrating Example

We first use a simple example to illustrate how FXE constructs control flow graphs from binaries. Consider the C program in Figure 1a. Although we present this example in C code for illustration purposes, FXE works directly on binaries. The function pointer `foo` can point to either 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`. 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, which 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 CFG constructed for the example program, which is exactly the same as the ideal CFG we wish to construct. This example illustrates how our technique precisely handles 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. Formalization

This section formalizes the problem of CFG construction and presents our core approach. First, we introduce a simple language for low-level programs and give its concrete semantics. Our language is modeled after previous work by Kinder *et al.* [20]. We then describe a few assumptions, under which we prove that our approach constructs precise CFGs.

$$\begin{aligned}
\mathcal{E}[\![r := e]\!](\sigma) &:= \sigma[r \mapsto \mathcal{A}[\![e]\!](\sigma)][pc \mapsto \sigma(pc) + \Delta(r := e)] \\
\mathcal{E}[\![m(e_1) := e_2]\!](\sigma) &:= \sigma[m[\mathcal{A}[\![e_1]\!](\sigma)] \mapsto \mathcal{A}[\![e_2]\!](\sigma)][pc \mapsto \sigma(pc) + \Delta(m(e_1) := e_2)] \\
\mathcal{E}[\![\text{cjmp } b, e]\!](\sigma) &:= \begin{cases} \sigma[pc \mapsto \mathcal{A}[\![e]\!](\sigma)] & \text{if } \mathcal{B}[\![b]\!](\sigma) = \mathbf{true} \\ \sigma[pc \mapsto \sigma(pc) + \Delta(\text{cjmp } b, e)] & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\text{jmp } e]\!](\sigma) &:= \sigma[pc \mapsto \mathcal{A}[\![e]\!](\sigma)]
\end{aligned}$$

Figure 2. Semantic function for statements under normal execution.

3.1 A Simple Low-Level Language

We consider a simple low-level language, JMP, with the following syntactic categories:

- numbers \mathbf{N} , which is the set of integers \mathbb{Z} ,
- truth values $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$,
- arithmetic expressions \mathbf{Aexp} ,
- Boolean expressions \mathbf{Bexp} ,
- statements \mathbf{Stmt} .

In JMP, $\mathbf{R} = \{r_1, r_2, \dots, r_n\}$ denotes the set of processor registers. JMP also uses the program counter pc and a memory access operator $m[\cdot]$. A JMP program is a sequence of statements. A statement in JMP can be either:

- a register assignment $r := e$, where $r \in \mathbf{R}$ and $e \in \mathbf{Aexp}$, which assigns the value of e to the register r ;
- a memory assignment $m[e_1] := e_2$, where $e_1, e_2 \in \mathbf{Aexp}$, which assigns the value of e_2 to the memory location specified by e_1 ;
- a conditional jump statement $\text{cjmp } b, e$, where $b \in \mathbf{Bexp}$ and $e \in \mathbf{Aexp}$, which transfers control to the statement at the address specified by e iff b evaluates to \mathbf{true} ; or
- an unconditional jump statement $\text{jmp } e$, where $e \in \mathbf{Aexp}$, which transfers control to the statement at the address specified by e .

The statements of a JMP program are given by a finite set of addresses $\mathbf{A} \subseteq \mathbb{N}$. Every program in JMP has a unique starting address $\mathbf{entry} \in \mathbf{A}$. The mapping $\mathbf{A} \rightarrow \mathbf{Stmt}$ between addresses and statements is expressed as $[s]^a$, for $s \in \mathbf{Stmt}$ and $a \in \mathbf{A}$.

JMP is general and captures the important features of typical assembly languages. For instance, it supports indirect jumps, as the expressions used in the jump statements may involve processor registers or memory locations. Although JMP does not explicitly include call and `ret` statements, they can be implemented by manipulating the program counter and then jumping to the desired locations.

3.2 Semantics of JMP

Now we define the formal semantics of JMP. The semantics of the language, which describes how a JMP program behaves when we execute it, is defined in terms of states. With respect to a state, the program counter evaluates to a program address; an arithmetic expression evaluates to an integer value; and a Boolean expression evaluates to a truth value. Formally, a program state is given by $\sigma = (pc, v, m) \in \Sigma = \mathbf{A} \times \mathbf{V} \times \mathbf{M}$, where

- $pc \in \mathbf{A}$ is the value of the program counter;
- $v \in \mathbf{V} = \mathbf{R} \rightarrow \mathbf{N}$ maps process registers to integers; and
- $m \in \mathbf{M} = \mathbb{N} \rightarrow \mathbf{N}$ maps memory locations to integers.

For a state $\sigma \in \Sigma$, $\sigma(pc)$ denotes the value of the program counter, $\sigma(r)$ denotes the value of a processor register $r \in \mathbf{R}$, and $\sigma(m[n])$ denotes the value of a memory location specified by $n \in \mathbb{N}$. We write $\sigma[\cdot \mapsto \cdot]$ for the new state obtained from σ by replacing the

value in the program counter, processor register or memory location with a new value. We now define the following semantic functions:

$$\begin{aligned}
\mathcal{A} &: \mathbf{Aexp} \rightarrow (\Sigma \rightarrow \mathbf{N}) \\
\mathcal{B} &: \mathbf{Bexp} \rightarrow (\Sigma \rightarrow \mathbf{T}) \\
\mathcal{E} &: \mathbf{Stmt} \rightarrow (\Sigma \rightarrow \Sigma).
\end{aligned}$$

The evaluation functions for arithmetic and Boolean expressions follow the standard denotational semantics in the literature [29], and we only show the definition of \mathcal{E} in Figure 2.

The function $\Delta : \mathbf{Stmt} \rightarrow \mathbb{N}$ calculates the size of a given statement (*i.e.*, instruction), which may vary with the architecture. For RISC architectures like PPC, this function yields a uniform length for all instructions, while for CISC architectures like x86, it may give different sizes to different instructions.

3.3 Control Flow Graph

Now we formally define the control flow graph of a program.

Definition 3.1 (Control Flow Graph). Given a program P , its *control flow graph* is a directed graph $G = (V, E)$, where V is the set of statements and $E \subseteq V \times V$ is the set of edges representing control flow between statements. A *control flow edge* from statement s_i to s_j is $e = (s_i, s_j) \in E$.

Using the concrete semantics of JMP, we describe how to construct a CFG in terms of program paths. Given a JMP program, a path π is a finite sequence of statements $(s_i)_{0 \leq i \leq n}$, such that

- σ_0 is the initial state where $\sigma_0(pc) = \mathbf{entry}$;
- $\forall i (0 \leq i < n), \sigma_{i+1} = \mathcal{E}[\![s_i]\!](\sigma_i)$.

For the program path π , we write $\mathcal{E}[\![\pi]\!](\sigma_0) = \sigma_n$ for the state reached by executing the sequence of statements in path π from the initial state σ_0 . Π denotes the set of all paths in a program.

Definition 3.2 (Path CFG). Given a path $\pi = (s_i)_{0 \leq i \leq n}$, the *path control flow graph (PCFG)* of π is given by $PCFG(\pi) = (V(\pi), E(\pi))$, where $V(\pi) = \{s_i \mid 0 \leq i \leq n\}$ and $E(\pi) = \{(s_i, s_{i+1}) \mid 0 \leq i < n\}$.

We define the ideal CFG based on path CFGs.

Definition 3.3 (Ideal CFG). Given a program, its *ideal control flow graph (ICFG)* is the union of the PCFGs of all paths:

$$ICFG = \bigcup_{\pi \in \Pi} PCFG(\pi),$$

where the \bigcup operator is defined as $G_1 \bigcup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$ for $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

3.4 Forced Execution

Now, we state a few key assumptions that guide and motivate our forced execution approach. We observe that these assumptions hold for binaries in general.

Assumption 1: Target Feasibility. *Both directions of a conditional jump are feasible.*

$\forall l \in \mathbf{A}$ with $[\text{cjmp } b, e]^l$, we assume

$$\begin{aligned}
\mathcal{F}[\![r := e]\!](\sigma) &:= \sigma[r \mapsto \mathcal{A}[\![e]\!](\sigma)][pc \mapsto \sigma(pc) + \Delta(r := e)] \\
\mathcal{F}[\![m(e_1) := e_2]\!](\sigma) &:= \sigma[m[\mathcal{A}[\![e_1]\!](\sigma)] \mapsto \mathcal{A}[\![e_2]\!](\sigma)][pc \mapsto \sigma(pc) + \Delta(m(e_1) := e_2)] \\
\mathcal{F}[\![\text{cjmp } b, e]\!](\sigma) &:= \begin{cases} \sigma[pc \mapsto \mathcal{A}[\![e]\!](\sigma)] \\ \sigma[pc \mapsto \sigma(pc) + \Delta(\text{cjmp } b, e)] \end{cases} \\
\mathcal{F}[\![\text{jmp } e]\!](\sigma) &:= \sigma[pc \mapsto \mathcal{A}[\![e]\!](\sigma)]
\end{aligned}$$

Figure 3. Semantic function for statements under forced execution.

- (i) $\exists \pi = (s_i)_{0 \leq i < m}$ s.t. $\sigma_m(pc) = l$ and $\mathcal{B}[\![b]\!](\sigma_m) = \mathbf{true}$ with $\sigma_m = \mathcal{E}[\![\pi]\!](\sigma_0)$,
- (ii) $\exists \pi' = (s'_j)_{0 \leq j < n}$ s.t. $\sigma'_n(pc) = l$ and $\mathcal{B}[\![b]\!](\sigma'_n) = \mathbf{false}$ with $\sigma'_n = \mathcal{E}[\![\pi']\!](\sigma'_0)$.

The paths π and π' could be the same program path with different initial states $\sigma_0 \neq \sigma'_0$ (e.g., different inputs).

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 that every remaining conditional branch can follow both its directions for the purpose of CFG construction.

Assumption 2: Target Resolution. *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.*

Given $\pi = (s_i)_{0 \leq i \leq n}$ where s_n is an indirect jump statement $\text{jmp } e$,

$$\begin{aligned}
\forall \sigma_0, \sigma'_0. (\forall i (0 \leq i < n), \mathcal{E}[\![s_i]\!](\sigma_i)(pc) = \mathcal{E}[\![s_i]\!](\sigma'_i)(pc)) \\
\Rightarrow \mathcal{A}[\![e]\!](\mathcal{E}[\![\pi]\!](\sigma_0)) = \mathcal{A}[\![e]\!](\mathcal{E}[\![\pi]\!](\sigma'_0)).
\end{aligned}$$

For ease of explanation, we consider indirect jumps to be unconditional, which is the case for architectures like x86. It would make no conceptual difference to include conditional indirect jumps.

This assumption basically means that the indirect branch target e is decided by the control flow path π leading to the branch, regardless of the initial program state (and thus other intermediate states). This holds true when the indirect branch target e has no data dependency on the initial state σ_0 or σ'_0 .

In practice, indirect branches usually serve the purpose of calling an external function through the import address table (IAT) or calling an internal function using a function pointer. In the first scenario, the target of the indirect branch is determined when the program is loaded and remains unchanged. For the second case, the function pointer is properly defined or assigned a valid address along a control flow path before the branch point, as is the case in Figure 1. However, indirect branch targets sometimes do have data dependency on the program states, and an exception to this assumption is the use of jump tables; we discuss how we handle jump tables in the next section.

Next, we describe how a program behaves under forced execution by defining its semantics. Under forced execution, the semantic functions for arithmetic and Boolean expressions remain the same as in normal execution. The semantic function for statements $\mathcal{F} : \mathbf{Stmt} \rightarrow (\Sigma \rightarrow \Sigma)$ is given in Figure 3.

The difference between \mathcal{F} and \mathcal{E} lies in the semantics of conditional jumps. In \mathcal{E} , the branch target is determined by the predicate, while in \mathcal{F} , the branch target is chosen non-deterministically so that both target directions can be executed. Under forced execution, a forced execution path $\hat{\pi}$ is a finite sequence of statements $\hat{\pi} = (s_i)_{0 \leq i \leq n}$, such that

- (i) $\hat{\sigma}_0$ is the initial state where $\hat{\sigma}_0 = \sigma_0$, $\hat{\sigma}_0(pc) = \mathbf{entry}$;
- (ii) $\forall i (0 \leq i < n), \hat{\sigma}_{i+1} = \mathcal{F}[\![s_i]\!](\hat{\sigma}_i)$.

For a JMP program, the statements are the same for normal execution and forced execution. However, the program states may be inconsistent under forced execution. We use $\hat{\sigma}$ to denote a possibly inconsistent program state. At the beginning of forced execution, the initial state $\hat{\sigma}_0$ is consistent and the same as σ_0 . Similar to a normal execution path, for a forced execution path $\hat{\pi}$, we write $\mathcal{F}[\![\hat{\pi}]\!](\hat{\sigma}_0) = \hat{\sigma}_n$ for the state reached by the forced execution of statements of path $\hat{\pi}$ from the initial state $\hat{\sigma}_0$. $\hat{\Pi}$ denotes the set of all forced execution paths of a program.

Definition 3.4 (Forced Path CFG). Given a forced execution path $\hat{\pi} = (s_i)_{0 \leq i \leq n}$, the *forced path control flow graph* (FPCFG) of $\hat{\pi}$ is $FPCFG(\hat{\pi}) = (\hat{V}(\hat{\pi}), \hat{E}(\hat{\pi}))$, where $\hat{V}(\hat{\pi}) = \{s_i \mid 0 \leq i \leq n\}$ and $\hat{E}(\hat{\pi}) = \{(s_i, s_{i+1}) \mid 0 \leq i < n\}$.

We define the forced CFG based on forced path CFGs.

Definition 3.5 (Forced CFG). Given a program, its *forced control flow graph* (FCFG) is the union of the FPCFGs of all forced execution paths:

$$FCFG = \bigcup_{\hat{\pi} \in \hat{\Pi}} FPCFG(\hat{\pi}).$$

We claim that given a program, under the target feasibility and target resolution assumptions, the forced CFG is precise with respect to the ideal CFG. We now formally define precision in our problem setting in terms of soundness and completeness.

Definition 3.6 (Soundness and Completeness). Given the ideal CFG $G = (V, E)$ for a program P , a CFG $G' = (V', E')$ for P is *sound* iff $V' \subseteq V$ and $E' \subseteq E$. G' is *complete* iff $V \subseteq V'$ and $E \subseteq E'$.

Theorem 3.1 Given a program in the JMP language, the forced CFG is both sound and complete.

Formally, given the ideal CFG $G = (V, E)$ and the forced CFG $\hat{G} = (\hat{V}, \hat{E})$ for program P , $\hat{V} = V$ and $\hat{E} = E$.

The proof is a straightforward induction on the length of paths using the two assumptions.

4. FXE

This section presents our CFG construction algorithm and describes key optimizations and extensions to scale it to realistic programs.

Section 3 defines the control flow graph of a program in terms of statements (*i.e.*, instructions). For the purpose of exposition, we recast it using basic blocks. A *basic block* is a maximal sequence of consecutive instructions with a single entry and a single exit. Each instruction in a basic block always dominates the ones after it in that block, and no other instruction can be executed between two sequential instructions in the same block. Therefore, for the rest of the paper, we denote a binary CFG as $G = (V, E)$, 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.

4.1 Basic Algorithm

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

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
11                   find_branch(inst) == NULL then
12                    branch = get_branch(inst);
13                    if branch.is_taken(branch) then
14                        branch.next_ip = ip + inst.length;
15                    else
16                        branch.next_ip = get_target(branch);
17                        branch.state = gray;
18                        save_cpu_mem_state(branch);
19                        add_to_branch_list(branch);
20                try
21                    ip = execute_inst(inst);
22                catch (exception)
23                    error_handler(ip, block, exception);
24            else
25                try
26                    ip = execute_block(block);
27                catch (exception)
28                    error_handler(ip, block, exception);
29            if branch = get_last_gray_branch() then
30                load_cpu_mem_state(branch);
31                ip = branch.next_ip;
32                branch.state = black;
33                current = get_branch_block(branch);
34            else
35                break;
36 split_block(cfg);
37 return cfg;
```

4.1.1 Dynamic Forced Execution

The core of our analysis is *dynamic forced execution*. We execute the program under analysis in a virtual environment. Under Assumption 1 (Target Feasibility), each conditional branch has two possible directions, so we control the execution of the program to explore both directions, 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. 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 is about to terminate, 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 path. When all the conditional branches are marked black, FXE terminates the exploration. During program execution, each block ends with a control transfer instruction. If the execution later jumps into the middle of another block, that block needs to be split into two blocks. This is implemented in the `split_block` procedure at Line 35. In essence, FXE explores program paths in a depth-first order.

In the course of forced execution, the program can often be in inconsistent states. This is because, instead of relying on constraint solvers, FXE simply manipulates the CPU instruction pointer to control the program execution and guide the path exploration. However, under our problem setting, we usually do not care about the correctness of the program computation. Instead, we are only interested in 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 Assumption 2 (Target Resolution). Even if the program is in inconsistent states, our analysis will not be affected as long as we have the correct indirect branch targets. For these reasons, we consider our forced execution approach 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 analysis result depends on the concrete input used to bootstrap the analysis. However, this is not the case. Our observation is that, for most programs, the CFG is an intrinsic static property that is independent of any specific input. Unlike traditional dynamic analysis where control flow paths taken at run time are largely determined by concrete inputs, our approach does not rely on specific inputs to drive the execution along different paths. Instead, we systematically explore all program paths. In fact, we do not feed any concrete input to the program and use anything that happens to be in the memory when inputs are required. This is equivalent to using random values for the inputs.

While we enjoy the benefits of conveniently resolving indirect branch targets, we need to address a few issues with forced execution. First, CPU and memory states are not consistently updated, which may lead to runtime exceptions. If these runtime exceptions are not properly handled, the program may 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 in the presence 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.

4.1.2 Saving and Restoring Program States

At each conditional branch, FXE takes a snapshot of the program state including the values of CPU registers and contents of the virtual memory 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 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 a branch, and we may end up with incorrect targets when we force the execution to follow the other path of the branch.

When FXE restores a program state, it basically revokes all the side-effects introduced while executing one path of the branch, so that the exploration of the other path starts with the same state. Conceptually, FXE follows both directions for each conditional

branch in parallel. By saving and restoring snapshots, we also avoid introducing additional inconsistencies into the program states.

4.1.3 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 may terminate or crash, leaving paths unexplored. Currently, we intercept and handle the following types of 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, and general protection fault.

We divide 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 continues exploring the current path. For example, a data transfer instruction writing to a read-only memory location 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.

4.2 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.

4.2.1 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 0×1000 . During dynamic execution, the instructions from L3 to L5 are executed 0×1000 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 this case, 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 4.1 (Indirect Branch Block). An *indirect branch block*, or simply *indirect block*, is a basic block that ends with an indirect branch instruction.

Definition 4.2 (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:

$$b = b' \vee \exists v_0 = b, \dots, v_n = b' [\forall 0 \leq i < n . (v_i, v_{i+1}) \in E]$$

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 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 which 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 b . We use a backward reachability algorithm to compute the set of blocks from which b is reachable. However, the computed block set is an over-approximation that may contain many unnecessary blocks. In order to improve efficiency, we perform an *intraprocedural* analysis to identify contributing blocks, limiting the computation within function boundaries. To this end, we use a heuristic to locate function entry points. A function usually starts with the instructions `push ebp` and `mov ebp, esp`. We examine the beginning of each block to see whether the instructions implement this typical function prologue.

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

We also need to take special care of loops, because it is possible that all the blocks in a loop are 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. 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 and we decrement the quota by one when a block is executed. For each indirect branch block, we keep track of its targets. Every time we identify a new target for an indirect branch 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 analysis. We only re-execute a block if it is a contributing block with unused quota. Otherwise, we rewind the execution to the nearest gray branch and explore another path.

The block quota scheme enables our approach to scale yet still cover as many paths as possible. Although we achieve efficiency by avoiding the re-execution of certain blocks, we may also miss some paths. For instance, if the quota for some blocks in function f is exhausted when f is called from one context, f will not return when it is called later from a different context. When this happens, we miss the function return edge and the fall-through path for the latter call branch. To address this issue, we add each call branch to the branch list in Algorithm 1 just like conditional branches, to ensure that the fall-through path can be explored later. To discover missing function return edges, we perform an intraprocedural forward reachability analysis on the partial CFG. More specifically, upon each function call, we remember the call site s and the target function f . Whenever f is invoked from a different call site s' , we track forward on the partial CFG and reset the quota for all the block within f . In this way, f has more opportunities to be executed under different contexts, and is more likely to return to different call sites.

4.2.2 Skipping External Code

Since our goal is to construct the CFG for the program under analysis, we skip the execution of all external functions to improve scalability without reducing coverage. For our analysis, external code (e.g., library functions) 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.

In compiler-generated binaries, most function calls return to their call sites [1]. Therefore, when FXE detects a function call pointing to external code, it forces the execution to immediately return to

the call site and continue execution along the fall-through path. However, we have observed that a few library functions (e.g., `exit`, `abort`) do not return to their original call sites. These functions are usually invoked when the program terminates. We treat these functions as special cases and do not follow their fall-through paths.

4.2.3 Handling Jump Tables

Indirect branches are sometimes used with jump tables. A jump table usually contains an array of pointers to code that handles various cases based on an index. During our analysis, we want to discover and explore all possible targets in each jump table efficiently. 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 tables:

```
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`, so we obtain the twelve targets consecutively from the table. This is one of the few patterns that we use to identify jump tables. 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.

4.2.4 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 callback functions are only reachable from external code. A callback function is registered and later invoked when certain events occur. For example, the `atexit` function, as suggested by its name, registers a callback function that will be invoked when the program terminates. Since FXE skips all external code, it may not execute these callback functions by following the 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 automatically detects 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.

Although FXE works well for a single-threaded program as it always analyzes the code for the main thread, it is also desirable to be able to analyze multi-threaded programs. As we do not execute

external library code, the process under analysis could not spawn child threads. However, FXE is able to discover threads as callback functions. This is because, when FXE intercepts function calls that are responsible for creating threads (e.g., `beginthread`), it is able to extract the starting address of the new thread from the arguments. In this way, we force the program to execute the code for the child threads later when all other program paths have been explored.

4.2.5 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. When this happens, FXE tries to detect and eliminate 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 user code, it considers the block as invalid. Finally, during an offline analysis phase, when FXE detects two 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.

5. Empirical Evaluation

We have implemented our technique in a prototype tool FXE to construct control flow graphs for Windows PE executables. FXE is based on QEMU [5], a whole system emulator. We chose QEMU in our implementation mainly for two reasons. First, QEMU’s binary translation works at the granularity of basic blocks, which is convenient for CFG construction. 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 Windows guest system on top of it.

We designed our evaluation of FXE to answer a few questions regarding its precision and scalability: 1) Can we trust the constructed CFGs, i.e., how precise are they? 2) How effective is FXE compared to state-of-the-art alternatives? 3) Does it scale to large programs? Section 5.1 considers the first question and includes an evaluation on the precision of FXE on a set of standard test programs. We show that FXE produces nearly ideal CFGs on these programs. Section 5.2 considers the second 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 benchmark programs. Section 5.3 considers the last question and shows that FXE scales well to large programs. In addition, we have also performed extensive experiments to evaluate the generality of FXE (i.e., is it robust against different compiler options or even different compilers?), and show that compilers have limited impact on the results. We present detailed generality evaluation results in the appendix.

5.1 Soundness and Completeness

In Section 3, we formally defined the notion of an ideal CFG (Definition 3.3) and what it means for a constructed CFG to be sound or complete (Definition 3.6). Now we evaluate the precision of FXE by measuring both its soundness and completeness.

Given the ideal CFG of a binary, we want the CFG constructed by FXE to be as close to the ideal one as possible. However, it is difficult (in fact undecidable) to compute the ideal CFG. Therefore, in practice, we cannot compare CFGs constructed by FXE with the ideal ones. For validation purpose, we 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

Program	LOC	Size(KB)	Test Cases	$ 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	726	26.3	4,130	308	435	317	451	97.48%	95.12%	100.00%	98.39%
printtokens2	570	22.3	4,115	346	483	348	486	99.43%	98.56%	100.00%	99.17%
replace	564	22.6	5,542	329	435	351	481	94.02%	89.81%	100.00%	99.08%
schedule	412	20.0	2,650	216	277	220	282	99.09%	97.87%	100.00%	98.92%
schedule2	374	20.3	2,710	242	312	251	315	96.41%	95.87%	100.00%	96.79%
tcas	173	20.1	1,608	178	236	179	242	99.44%	97.52%	100.00%	100.00%
totinfo	565	21.5	1,052	253	313	267	336	95.51%	92.86%	100.00%	99.04%
Average	-	-	-	-	-	-	-	97.34%	95.37%	100.00%	98.77%

Table 1. Dynamic vs. FXE CFGs for the Siemens Test Suite.

for two reasons. First, constructing CFGs from source code requires pointer analyses to understand function pointers. 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 as compilers usually introduce additional code into the binary even without optimization.

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 System¹, 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 LOC, the binary file size, and the number of test cases.

We first compile these 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 does not apply forced execution on initialization and termination routines generated by the compiler. Because these routines handle errors and exceptions that never occur during the dynamic CFG generation, we exclude them from forced execution for a fair comparison.

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 1 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. We manually inspected all the additional blocks and edges that are reported by FXE but missing in G_d . After examining the source code and the binaries, we found that all those blocks and edges fall into the following two categories:

- 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).

For each program, there is a file named UNREACHABLE in the source directory that describes which part of the source code is not

exercised by the test inputs. 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;

```

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 1 represent the percentage of blocks and edges in G_d that 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 confirmed that all the edges not covered by FXE are function return edges, which we explained in Section 4.2.1.

5.2 Comparison with IDA

To show the effectiveness of FXE compared to state-of-the-art alternatives, we performed experiments on CINT2000², the integer components of the SPEC CPU2000 benchmarks. SPECint 2000 contains twelve applications written in C or C++. We compiled these programs using GCC with default compiler options specified in the Makefiles.

¹ <http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/>

² <http://www.spec.org/cpu2000/CINT2000/>

Program	LOC	Size(KB)	$ 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	8,616	84	85	168	95	47	115	260	100.00%	100.00%	21.05%	453.19%
vpr	17,729	200	274	1,078	313	104	331	1,504	100.00%	100.00%	5.75%	1346.15%
gcc	222,182	1,732	1,507	9,434	1,625	552	2,482	21,154	100.00%	100.00%	52.74%	3732.25%
mcf	2,423	33	53	81	60	27	76	131	100.00%	100.00%	26.67%	385.19%
crafty	21,150	282	140	908	180	207	198	2,096	100.00%	100.00%	10.00%	912.56%
parser	11,391	142	318	1,117	349	99	368	1,351	100.00%	100.00%	5.44%	1264.65%
eon	41,009	1,338	1,086	5,667	814	87	1,384	4,741	100.00%	100.00%	70.02%	5349.43%
perlbmk	85,464	761	508	2,188	406	125	837	5,225	100.00%	100.00%	106.16%	4080.00%
gap	71,430	662	832	3,966	173	48	2,147	11,138	100.00%	100.00%	1141.04%	23104.17%
vortex	67,220	711	608	3,400	668	219	739	3,933	100.00%	100.00%	10.63%	1695.89%
bzip2	4,649	59	78	180	91	74	114	283	100.00%	100.00%	25.27%	282.43%
twolf	20,459	306	160	831	209	107	227	1,649	100.00%	100.00%	8.61%	1441.12%
Average	-	-	-	-	-	-	-	-	100.00%	100.00%	123.62%	3670.59%

Table 2. IDA vs. FXE CFGs (indirect) for the SPECint benchmarks.

We analyzed the twelve benchmark programs with FXE and constructed $G_f = (V_f, E_f)$ for each binary. Then we built $G_i = (V_i, E_i)$ for each program using IDA Pro 6.1³, which is an interactive disassembler that is considered the state-of-the-art commercial disassembler. IDA Pro does not directly support the generation of CFGs, so we developed a plugin, *IDA CFG*, 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. If it is an unconditional branch, IDA CFG continues from the branch target. If the control transfer is conditional, it explores both branch targets. IDA CFG recursively follows all the paths found by IDA Pro statically. 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 those invalid fall-through paths. In addition, IDA CFG also stops exploring the current path upon encountering privileged instructions.

Since one major advantage of FXE over IDA and other static analysis tools is its ability to resolve indirect branch targets, here we focus on the comparison of indirect blocks and edges in the CFGs.

The results for FXE and a comparison to IDA are presented in Table 2. For each program, V'_f and E'_f represent the sets of indirect blocks and edges in the CFG generated by FXE. Similarly, V'_i and E'_i represent the results from IDA. The columns $\frac{|V'_f \cap V'_i|}{|V'_i|}$ and $\frac{|E'_f \cap E'_i|}{|E'_i|}$ denote the percentage of indirect blocks and edges in the results from IDA that are covered by FXE. We observe that FXE is able to achieve 100% coverage on both indirect blocks and edges over the results of IDA for all the programs.

Table 2 also shows the relative increase in the number of indirect blocks and edges, using the results from IDA as the baseline. We can see that there is a much greater increase in the number of edges than blocks. This also shows that IDA Pro has very limited ability to resolve indirect branches.

For the SPECint benchmarks, we observe that most runtime exceptions during forced execution are non-critical (over 99% on average), and FXE was able to recover from them and continue

its exploration. This confirms our assumption that indirect branch targets usually do not depend on intermediate program states, thus corrupted states do not affect the program control flow in most cases.

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 columns $|V'_d|$ and $|E'_d|$ in Table 2 represent the numbers of indirect blocks and edges in each dynamic CFG respectively.

Table 3 shows the indirect block and edge coverage on the dynamic CFGs both by IDA and FXE. We observe that FXE consistently achieves better coverage than IDA for all the programs. For most programs, FXE finds all the indirect blocks included in the corresponding dynamic CFG. For other programs where FXE misses a few blocks, the coverage is still far better compared to IDA. This also indicates that many 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). Our optimizations also prevent FXE from finding some function return edges, as explained in Section 4.2.1.

To better understand limitations of FXE empirically, we performed a manual study on the missing indirect blocks and edges. Specifically, we divide the missing edges into two categories: return edges and non-return edges. We confirmed that most of the missing edges belong to the first category. The last column in Table 3 shows the coverage of indirect edges if FXE were able to find all return edges. For each program, $E''_d \subseteq E'_d$ represents the set of return edges in the dynamic CFG. For the non-return edges, we manually examined some of the missing ones and discovered a few common cases where FXE would miss them. We now illustrate these cases using some examples.

Example 6. The following code from `gcc` declares and initializes a global array of function pointers `bcc_gen_fctn`. Function `do_jump_for_compare` invokes certain functions pointed to by elements of this array. The `comparison` variable is a parameter that acts as an index and decides which function to call inside `do_jump_for_compare`.

³ <http://www.hex-rays.com/idaipro/>

Program	$ V'_i \cap V'_d $	$ E'_i \cap E'_d $	$ V'_f \cap V'_d $	$ E'_f \cap E'_d $	$ (E'_f \cap E'_d) \cup E'_d $
	$ V'_d $	$ E'_d $	$ V'_d $	$ E'_d $	$ E'_d $
gzip	91.76%	12.50%	100.00%	80.95%	100.00%
vpr	97.45%	6.68%	100.00%	77.55%	100.00%
gcc	72.53%	1.12%	99.87%	66.81%	98.55%
mcf	86.79%	22.22%	100.00%	96.30%	100.00%
crafty	95.00%	5.29%	100.00%	81.39%	100.00%
parser	97.17%	4.21%	100.00%	76.99%	99.82%
eon	44.29%	0.39%	83.89%	52.46%	97.46%
perlbmk	54.13%	2.29%	88.98%	58.55%	95.66%
gap	17.07%	0.55%	100.00%	16.94%	68.81%
vortex	94.57%	4.18%	99.84%	39.32%	99.56%
bzip2	92.31%	11.11%	100.00%	80.56%	100.00%
twolf	95.62%	6.02%	100.00%	90.01%	100.00%
Average	78.22%	6.38%	97.72%	68.15%	96.66%

Table 3. Coverage of dynamic CFGs (indirect) by IDA vs. FXE.

```

bcc_gen_fctn[(int) EQ] = gen_beq;
bcc_gen_fctn[(int) NE] = gen_bne;
...

static void do_jump_for_compare (rtx
    comparison, rtx if_false_label, rtx
    if_true_label) {
    ...
    if (bcc_gen_fctn[(int) GET_CODE
        (comparison)] != 0)
        emit_jump_insn ((*bcc_gen_fctn[(int)
            GET_CODE(comparison)])(if_true_label));
    ...
}

```

During forced execution, `do_jump_for_compare` may execute with a certain `comparison` value. However, when this function is called again later with a different `comparison` value, some blocks inside this function may have no quota left so the execution would skip to the next instruction following the call to `do_jump_for_compare`. In this case, a different function pointer target is not exercised and FXE would miss that indirect branch edge. More generally, for a function f , when one or more of its parameters influence function pointer calculation inside f , FXE may miss indirect branch targets due to optimization.

Example 7. Consider the following code snippet from the `gap` benchmark. `EvTab` and `PrTab` are global arrays that hold pointers to various functions. The elements of these arrays are assigned inside a loop.

```

for (type1 = T_LIST; type1 < T_REC; type1++) {
    EvTab[type1] = EvList;
    PrTab[type1] = PrList;
}

```

When FXE explores this program, the basic blocks in the loop body will be executed only once so only one element in each array is properly assigned. This is similar to Example 1 from Section 4.2.1, but in this case, most of the elements in these two arrays will not have valid function pointer values as the assignments are skipped due to our optimization. Since at least one element of each array has the correct pointer value, the target function may still be explored. However, when other elements of these arrays are used to invoke the target functions, FXE does not have valid branch targets and thus misses those indirect branch edges. Our manual inspection reveals that such loops are common in `gap`, which explains why FXE is able to find all the basic blocks in `gap` but misses some indirect branch edges. In general, FXE may miss indirect branch edges when function pointers are calculated or assigned inside loops.

In Section 5.4, we discuss possible ways to improve FXE, and we leave them for future work.

Another case where FXE misses indirect edges involves callback functions. Recall Example 3 from Section 4.2.4. After the program registers a callback function for `qsort`, our forced execution will skip the external library function and continue from the following instruction after the call. The registered callback function will not be explored until FXE has explored all the other program paths, so the edge from the call site to the callback function is missing. Although in this specific case, it is clear that the `comparator` function would be invoked right from `qsort`, in other cases, callback functions may be invoked at any arbitrary point (e.g., upon a special system event). Therefore, FXE may not properly connect the call site to the callback function.

In addition to the comparison with IDA Pro, it would also be interesting to compare our approach to existing state-of-the-art static analyzers like Jakstab [19] and CodeSurfer/x86 [3]. For Jakstab, we tried to run it on `mcf`, the smallest benchmark program in SPECint, but it either prematurely halted with incorrect results under default settings or failed to terminate within hours under the suggested settings. For CodeSurfer/x86, however, we are unable to perform such a comparison because it is a proprietary platform that is not yet publicly available. In fact, CodeSurfer/x86 relies on IDA Pro to disassemble binaries and build CFGs. It then refines the IDA results to account for indirect branches using a static value-set analysis (VSA) [2], 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.

5.3 Performance

In our experiments, we do not set any time limit on FXE’s exploration of each program path, nor do we limit its overall execution time. Table 4 shows the time for FXE and IDA to construct the CFG for each SPECint benchmark program. The time given in column T_i includes both the initial disassembly and analysis time by IDA Pro and the time IDA CFG takes to generate the CFG output. The column T_f represents the total amount of time taken by FXE in both its execution phase and its offline analysis phase. 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 a reasonable amount of time. This empirically shows that the optimization techniques described in Section 4.2 can greatly improve the scalability

Program	T_d (s)	T_i (s)	T_f (s)
gzip	511.884	1.256	5.891
vpr	3,796.509	2.517	22.677
gcc	522.627	54.370	1,219.144
mcf	41.964	0.769	1.806
crafty	915.324	4.522	52.766
parser	974.737	2.222	24.708
eon	299.991	12.573	122.962
perlbmk	48.896	31.653	150.452
gap	278.668	7.968	207.688
vortex	2,004.288	7.214	141.331
bzip2	1,754.205	1.235	5.190
twolf	48.858	3.885	36.741

Table 4. Performance results.

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 2.2 GHz dual-core CPU and 4GB of RAM.

Table 4 also provides the time (T_d) to generate the dynamic CFG by running each program with its test input. These numbers correspond to one dynamic run with a single test input. Although we may improve the efficiency by running directly on the hardware instead of in 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 code size, which makes it 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, the dynamic CFGs are far less complete.

During dynamic forced execution, a snapshot is taken at each branch point. In our experiments, the size of a program state is 1.24 MB on average, while the largest one is approximately 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 is around 350 MB, with 256 MB allocated for the guest system.

5.4 Discussions

In Section 4.2, we described a few optimization techniques, which may lead FXE to produce imprecise results for certain binaries. 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 of the program. 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. We also plan to investigate more effective approaches (*e.g.*, aided by machine learning) to dynamically tune the quota for each block.

As our empirical study in the previous section shows, FXE may miss certain blocks and indirect edges when function pointer calculation depends on function parameters. To mitigate this issue, we could remember the actual parameter values passed at function call sites, and reset the block quota within the target function when we detect different parameter values. This will help expose different function pointers used inside the function and we leave this improvement in our future work.

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

Finally, FXE relies on a simple pattern matching algorithm to detect jump tables that contain indirect branch targets. Although we are able to 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 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` utility⁴ 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, 25, 27] follow the program control flow by interpreting branch instructions. However, these techniques 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 the basic algorithms. Schwarz *et al.* [23] 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.* [26] 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, fail to produce precise control flow information. Our approach, however, is based on dynamic (forced) execution, and thus can better handle indirect branches and construct precise CFGs.

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 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 [28] 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

⁴ <http://www.gnu.org/software/binutils/manual/>

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. In this way, indirect blocks are more likely to be executed under different contexts, so that we can identify more indirect branch paths.

7. Conclusion

In this paper, we have presented a novel, practical technique based on forced execution to extract precise control flow information from binaries. Using forced execution, our technique systematically and efficiently 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 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 the 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 the 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 the 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 the 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 the 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 the 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 the ACM SIGPLAN Conference on 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 the 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 the International Conference 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 the ACM/IEEE International Conference on Software Engineering*, pages 191–200, 1994.
- [18] J. Jurjens and M. Yampolskiy. Code Security Analysis with Assertions. In *Proceedings of the IEEE/ACM 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 the 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 the 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 the 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 the IEEE Symposium on Security and Privacy*, pages 231–245, 2007.
- [23] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proceedings of the Working Conference on Reverse Engineering*, pages 45–54, 2002.
- [24] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, 2005.
- [25] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary Translation. *Digital Technical Journal*, 4(4), 1992.
- [26] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. De-moen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1013–1019, 2000.
- [27] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 23–30, 2000.
- [28] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 219–235, 2007.
- [29] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.
- [30] X. Zhang, N. Gupta, and R. Gupta. Locating Faults Through Automated Predicate Switching. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 272–281, 2006.

Program	Size (KB)						
	GCC -O0	GCC -O1	GCC -O2	GCC -O3	GCC -Os	MSVC	ICC
printtokens	26.3	26.1	25.5	26.4	24.3	9.5	15.0
printtokens2	22.3	21.4	20.7	20.6	20.3	6.5	13.0
replace	22.6	22.1	22.1	24.1	21.1	6.5	15.0
schedule	20.0	20.0	20.0	21.5	19.5	6.0	10.5
schedule2	20.3	20.3	20.3	23.3	19.6	6.0	11.0
tcas	20.1	19.6	20.1	20.1	19.6	5.0	8.0
totinfo	21.5	21.5	21.5	21.5	20.8	6.5	35.5

Table A1. Summary of Siemens Test Suite program binaries.

APPENDIX

We present a detailed evaluation on the generality of FXE and provide some additional results in the appendix.

A. 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 generate precise CFGs for programs that are compiled with different optimization options or even different compilers. For this evaluation, we use the same programs from the Siemens Test Suite.

For each test program, we first compile the source code using the same compiler (*i.e.*, GCC), but with different optimization options. Then we build dynamic CFGs in the same way as described in Section 5.1. We also generate CFGs with FXE, and finally compare them with the dynamic ones. Table A1 gives a summary of the compiled binaries for each program.

Table A2 presents the average soundness and completeness results for binaries compiled under each optimization level of GCC.

For instance, the columns $\frac{|V_f \cap V_d|}{|V_d|}$ and $\frac{|E_f \cap E_d|}{|E_d|}$ represent the average block and edge coverage of all seven test programs by FXE. The numbers from the first row correspond to the average numbers from the bottom of Table 1 in Section 5.1.

Detailed results are also provided in the following tables. Table A3 through Table A6 present the soundness and completeness results for each program compiled with different optimization levels of GCC (*i.e.*, -O1, -O2, -O3, and -Os). Note that the results for optimization level zero is given previously in Table 1 in Section 5.1. From the numbers in these tables, we can see that different compiler optimizations have negligible impact on our results, and FXE is able to consistently generate precise CFGs for all the binaries in our experiment.

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 summarized in Table A2. More detailed results can be found in Table A7 and Table A8. The results from all these tables indicate that our technique is general and FXE is robust against different compilers.

B. Overall CFG Comparison with IDA

We presented a comparison of FXE and IDA Pro on the SPECint benchmark programs in Section 5.2, where we focused on indirect blocks and edges in the CFGs. In this section, we present the comparison results on the overall CFGs. For this experiment, we

use the same binaries compiled with GCC under optimization level zero as those used in Section 5.2.

We first compare the overall CFGs built by FXE with those constructed by IDA. The number of blocks and edges reported for each binary is provided in Table A9. 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 columns $\frac{|V_f \cap V_i|}{|V_i|}$ and $\frac{|E_f \cap E_i|}{|E_i|}$ denote the block and edge coverage by FXE over the CFGs constructed by IDA. The results show that FXE is able to find all the blocks and edges reported by IDA. As we expect, FXE discovers more blocks and edges than IDA, and the relative increase is given in the two columns $\frac{|V_f \setminus V_i|}{|V_i|}$ and $\frac{|E_f \setminus E_i|}{|E_i|}$ in Table A9.

To measure the quality of our results, we also compare the overall CFGs generated by IDA and FXE against the dynamic ones. Table A10 shows the coverage of dynamic CFGs both by IDA and FXE. We can see from the table that FXE covers all of the blocks and most of the edges in the corresponding dynamic CFGs on most of the programs. The results also indicate that, compared to IDA, FXE is able to achieve far better coverage for both blocks and edges on all the benchmark programs.

C. More Experiments on SPECint Benchmarks

In Section A, we showed that FXE is robust against compiler options with an evaluation on the Siemens Test Suite. To further demonstrate the effectiveness of FXE, we perform additional experiments and compare with IDA Pro on the larger SPECint benchmarks. Specifically, we compile the SPECint benchmark programs with different optimization options under GCC, and perform detailed comparison with IDA Pro. An overview of the compiled SPECint benchmark binaries is given in Table A11. We summarize the experimental results and present the average numbers under each optimization level in Table A12 and Table A13.

The detailed evaluation results are organized as follows. Table A14 through Table A17 provide the comparison of overall CFGs constructed by IDA and FXE under different optimization levels of GCC. Table A18 through Table A21 compare the overall block and edge coverage over dynamic CFGs by IDA and FXE. Next, Table A22 through Table A25 focus on the comparison of indirect blocks and edges in the CFGs. Finally, Table A26 through Table A29 show the coverage of dynamic CFGs both by IDA and FXE under different optimization levels.

These tables indicate that FXE is able to find almost all the blocks and edges reported by IDA. We observe that, when compared to dynamic CFGs, FXE consistently achieves far better results than IDA on all the benchmark programs. Results from these additional experiments also suggest that FXE is robust with respect to compiler optimization options.

Compiler	$\frac{ V_d \cap V_f }{ V_f }$	$\frac{ E_d \cap E_f }{ E_f }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	GCC -O0	97.34%	95.37%	100.00%
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%
MS Visual C++	95.15%	89.45%	100.00%	95.26%
Intel C++ Compiler	89.87%	81.73%	100.00%	98.15%

Table A2. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with different compilers and optimization options.

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	287	410	293
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 A3. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -O1.

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	249	346	253
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 A4. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -O2.

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	287	387	302
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 A5. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -O3.

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	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 A6. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with GCC -Os.

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	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 A7. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with MS Visual C++.

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	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 A8. Dynamic vs. FXE CFGs for the Siemens Test Suite compiled with Intel C++ Compiler.

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	1,508	2,000	1,603	2,318	100.00%	100.00%	6.30%	15.90%
vpr	3,885	5,432	6,017	7,599	6,157	9,169	100.00%	100.00%	2.33%	20.66%
gcc	38,102	54,925	47,094	66,588	85,110	138,854	100.00%	100.00%	80.72%	108.53%
mcf	432	561	594	767	676	965	100.00%	100.00%	13.80%	25.81%
crafty	4,658	6,818	8,318	11,499	8,835	14,084	100.00%	100.00%	6.22%	22.48%
parser	4,393	6,329	5,924	7,559	6,018	8,917	100.00%	100.00%	1.59%	17.97%
eon	12,243	19,355	7,084	7,487	15,536	24,613	100.00%	100.00%	119.31%	228.74%
perlbmk	6,707	9,282	6,779	9,112	20,128	32,125	100.00%	100.00%	196.92%	252.56%
gap	7,990	12,534	4,882	6,429	32,539	53,208	100.00%	100.00%	566.51%	727.62%
vortex	11,077	14,621	18,599	25,755	19,625	30,796	100.00%	100.00%	5.52%	19.57%
bzip2	979	1,287	1,398	1,861	1,489	2,166	100.00%	100.00%	6.51%	16.39%
twolf	3,740	5,245	9,078	12,434	9,283	14,236	100.00%	100.00%	2.26%	14.49%
Average							100.00%	100.00%	84.00%	122.56%

Table A9. IDA vs. FXE CFGs for the SPECint benchmarks.

Program	$\frac{ V_i \cap V_d }{ V_d }$	$\frac{ E_i \cap E_d }{ E_d }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	gzip	96.67%	86.43%	100.00%
vpr	99.31%	81.06%	100.00%	95.54%
gcc	63.17%	53.34%	99.58%	93.97%
mcf	93.75%	84.67%	100.00%	99.47%
crafty	92.23%	80.48%	100.00%	97.52%
parser	99.18%	82.57%	100.00%	95.94%
eon	25.30%	14.69%	85.16%	76.81%
perlbmk	36.50%	27.72%	84.88%	78.75%
gap	29.75%	21.57%	100.00%	73.72%
vortex	97.58%	75.98%	99.99%	85.89%
bzip2	97.34%	85.78%	100.00%	97.28%
twolf	99.28%	84.67%	100.00%	98.42%
Average	77.51%	64.91%	97.47%	90.91%

Table A10. Coverage of dynamic CFGs by IDA vs. FXE.

Program	Size (KB)				
	GCC -O0	GCC -O1	GCC -O2	GCC -O3	GCC -Os
gzip	84	77	67	73	61
vpr	200	169	165	175	139
gcc	1,732	1,404	1,420	1,635	1,130
mcf	33	30	31	31	29
crafty	282	256	264	277	233
parser	142	127	132	175	112
eon	1,338	1,168	1,136	1,151	1,055
perlbmk	761	622	645	736	526
gap	662	492	484	517	398
vortex	711	644	663	683	495
bzip2	59	53	55	65	50
twolf	306	232	229	239	194

Table A11. Summary of SPECint benchmark program binaries.

Compiler Option	$\frac{ V_f \cap V_i }{ V_i }$	$\frac{ E_f \cap E_i }{ E_i }$	$\frac{ V_f \setminus V_i }{ V_i }$	$\frac{ E_f \setminus E_i }{ E_i }$	$\frac{ V_i \cap V_d }{ V_d }$	$\frac{ E_i \cap E_d }{ E_d }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	GCC -O0	100.00%	100.00%	84.00%	122.56%	77.51%	64.91%	97.47%
GCC -O1	100.00%	100.00%	52.79%	72.53%	85.00%	71.31%	97.38%	89.79%
GCC -O2	100.00%	100.00%	33.80%	51.71%	85.38%	71.13%	95.31%	86.50%
GCC -O3	100.00%	100.00%	37.58%	56.01%	85.42%	72.55%	96.20%	87.98%
GCC -Os	100.00%	100.00%	52.55%	80.39%	85.28%	71.97%	97.47%	90.05%

Table A12. IDA vs. FXE CFGs for the SPECint benchmarks under different compiler optimization levels.

Compiler Option	$\frac{ V'_f \cap V'_i }{ V'_i }$	$\frac{ E'_f \cap E'_i }{ E'_i }$	$\frac{ V'_f \setminus V'_i }{ V'_i }$	$\frac{ E'_f \setminus E'_i }{ E'_i }$	$\frac{ V'_i \cap V'_d }{ V'_d }$	$\frac{ E'_i \cap E'_d }{ E'_d }$	$\frac{ V'_f \cap V'_d }{ V'_d }$	$\frac{ E'_f \cap E'_d }{ E'_d }$
	GCC -O0	100.00%	100.00%	123.62%	3670.59%	78.22%	6.38%	97.72%
GCC -O1	100.00%	99.72%	111.37%	1244.65%	81.90%	10.30%	97.41%	62.50%
GCC -O2	100.00%	99.99%	60.59%	1055.72%	81.70%	9.03%	93.97%	54.68%
GCC -O3	100.00%	100.00%	88.49%	1169.19%	80.92%	9.80%	95.00%	57.49%
GCC -Os	100.00%	100.00%	112.07%	2396.97%	82.30%	9.15%	96.95%	64.48%

Table A13. IDA vs. FXE CFGs (indirect) for the SPECint benchmarks under different compiler optimization levels.

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	890	1,209	1,381	1,942	1,474	2,245	100.00%	100.00%	6.73%	15.60%
vpr	3,460	4,971	5,536	7,481	5,672	8,859	100.00%	100.00%	2.46%	18.42%
gcc	33,675	49,984	71,099	105,519	74,165	120,793	100.00%	99.99%	4.31%	14.48%
mcf	394	521	535	742	617	928	100.00%	100.00%	15.33%	25.07%
crafty	4,219	6,385	8,030	11,725	8,121	13,450	100.00%	100.00%	1.13%	14.71%
parser	3,854	5,817	5,283	7,334	5,377	8,607	100.00%	100.00%	1.78%	17.36%
eon	8,695	13,476	5,742	6,566	11,818	16,654	100.00%	100.00%	105.82%	153.64%
perlbmk	6,249	8,746	15,111	22,256	18,479	29,915	100.00%	100.00%	22.29%	34.41%
gap	7,004	11,604	4,557	6,412	25,574	40,702	100.00%	99.98%	461.20%	534.78%
vortex	10,463	13,908	16,800	24,052	17,484	27,699	100.00%	99.97%	4.07%	15.16%
bzip2	847	1,163	1,226	1,751	1,316	2,038	100.00%	100.00%	7.34%	16.39%
twolf	3,370	4,843	8,189	12,271	8,275	13,534	100.00%	100.00%	1.05%	10.29%
Average							100.00%	100.00%	52.79%	72.53%

Table A14. IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -O1.

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	899	1,227	1,447	1,984	1,539	2,266	100.00%	100.00%	6.36%	14.21%
vpr	2,733	3,929	5,462	7,482	5,552	8,585	100.00%	100.00%	1.65%	14.74%
gcc	33,826	50,404	70,550	102,297	72,455	117,033	100.00%	100.00%	2.70%	14.41%
mcf	423	549	573	760	655	951	100.00%	100.00%	14.31%	25.13%
crafty	4,345	6,805	8,335	11,920	8,420	13,645	100.00%	100.00%	1.02%	14.47%
parser	4,045	5,980	5,619	7,491	5,720	8,777	100.00%	100.00%	1.80%	17.17%
eon	7,140	9,961	5,350	6,268	10,266	14,894	100.00%	100.00%	91.89%	137.62%
perlbmk	6,276	8,763	15,415	21,955	28,708	44,112	100.00%	100.00%	86.23%	100.92%
gap	7,124	11,851	4,726	6,393	13,543	21,713	100.00%	100.00%	186.56%	239.64%
vortex	10,752	14,118	18,582	25,396	19,307	29,037	100.00%	100.00%	3.90%	14.34%
bzip2	893	1,224	1,323	1,797	1,431	2,154	100.00%	100.00%	8.16%	19.87%
twolf	3,627	5,114	8,607	12,561	8,696	13,559	100.00%	100.00%	1.03%	7.95%
Average							100.00%	100.00%	33.80%	51.71%

Table A15. IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -O2.

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	953	1,309	1,572	2,185	1,664	2,470	100.00%	100.00%	5.85%	13.04%
vpr	3,659	5,210	5,989	8,445	6,079	9,509	100.00%	100.00%	1.50%	12.60%
gcc	37,447	55,018	84,210	124,157	87,188	140,938	100.00%	100.00%	3.54%	13.52%
mcf	432	567	576	780	658	965	100.00%	100.00%	14.24%	23.72%
crafty	4,604	7,252	8,685	12,476	8,770	14,423	100.00%	100.00%	0.98%	15.61%
parser	4,881	7,310	7,308	10,450	7,394	11,751	100.00%	100.00%	1.18%	12.45%
eon	7,169	10,000	5,324	6,252	10,266	14,911	100.00%	100.00%	92.82%	138.50%
perlbmk	6,709	9,221	18,902	27,720	35,648	55,207	100.00%	100.00%	88.59%	99.16%
gap	7,430	12,353	5,249	7,216	17,340	29,067	100.00%	100.00%	230.35%	302.81%
vortex	5,161	6,745	18,952	26,029	19,648	29,506	100.00%	100.00%	3.67%	13.36%
bzip2	921	1,254	1,402	1,993	1,504	2,364	100.00%	100.00%	7.28%	18.62%
twolf	3,649	5,129	8,809	12,906	8,895	14,036	100.00%	100.00%	0.98%	8.76%
Average							100.00%	100.00%	37.58%	56.01%

Table A16. IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -O3.

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	896	1,198	1,401	1,877	1,487	2,169	100.00%	100.00%	6.14%	15.56%
vpr	3,442	4,939	5,329	7,135	5,414	8,301	100.00%	100.00%	1.60%	16.34%
gcc	31,923	46,362	69,207	100,088	71,696	114,089	100.00%	100.00%	3.60%	13.99%
mcf	409	525	553	716	635	910	100.00%	100.00%	14.83%	27.09%
crafty	4,253	6,283	7,997	11,280	8,089	13,031	100.00%	100.00%	1.15%	15.52%
parser	4,045	5,802	5,551	7,221	5,657	8,446	100.00%	100.00%	1.91%	16.96%
eon	7,797	11,232	5,680	6,611	10,608	15,590	100.00%	100.00%	86.76%	135.82%
perlbmk	6,437	8,940	15,259	21,647	27,009	41,355	100.00%	100.00%	77.00%	91.04%
gap	6,904	11,333	4,581	6,126	24,077	42,101	100.00%	100.00%	425.58%	587.25%
vortex	10,515	13,937	18,098	25,050	18,643	29,051	100.00%	100.00%	3.01%	15.97%
bzip2	875	1,172	1,295	1,717	1,398	2,046	100.00%	100.00%	7.95%	19.16%
twolf	3,537	5,000	8,190	11,655	8,279	12,821	100.00%	100.00%	1.09%	10.00%
Average							100.00%	100.00%	52.55%	80.39%

Table A17. IDA vs. FXE CFGs for the SPECint benchmarks compiled with GCC -Os.

Program	$\frac{ V_i \cap V_d }{ V_d }$	$\frac{ E_i \cap E_d }{ E_d }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	gzip	96.52%	85.61%	100.00%
vpr	99.22%	80.19%	100.00%	93.66%
gcc	97.46%	81.96%	99.60%	92.72%
mcf	93.15%	84.45%	100.00%	98.66%
crafty	99.36%	88.66%	100.00%	97.53%
parser	99.07%	82.31%	100.00%	95.58%
eon	27.57%	17.14%	86.20%	77.10%
perlbmk	80.96%	66.96%	85.04%	78.79%
gap	31.90%	22.58%	97.74%	71.00%
vortex	98.64%	76.35%	100.00%	84.40%
bzip2	96.93%	84.35%	100.00%	95.87%
twolf	99.20%	85.13%	100.00%	95.02%
Average	85.00%	71.31%	97.38%	89.79%

Table A18. Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O1.

Program	$\frac{ V_i \cap V_d }{ V_d }$	$\frac{ E_i \cap E_d }{ E_d }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	gzip	96.55%	84.84%	100.00%
vpr	99.01%	81.09%	100.00%	92.87%
gcc	97.43%	80.81%	98.89%	90.41%
mcf	93.62%	84.34%	100.00%	98.36%
crafty	99.38%	83.76%	100.00%	91.96%
parser	99.16%	81.54%	99.83%	94.25%
eon	31.54%	22.32%	81.27%	65.79%
perlbmk	81.55%	67.16%	92.59%	83.02%
gap	31.33%	21.79%	71.32%	52.62%
vortex	98.66%	76.83%	99.85%	83.84%
bzip2	97.09%	83.33%	100.00%	95.18%
twolf	99.26%	85.75%	100.00%	94.25%
Average	85.38%	71.13%	95.31%	86.50%

Table A19. Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O2.

Program	$\frac{ V_i \cap V_d }{ V_d }$	$\frac{ E_i \cap E_d }{ E_d }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	gzip	96.75%	87.01%	100.00%
vpr	99.26%	84.11%	100.00%	94.76%
gcc	97.38%	82.27%	98.82%	90.88%
mcf	93.75%	85.89%	100.00%	98.59%
crafty	99.41%	83.15%	100.00%	93.17%
parser	99.30%	86.06%	99.86%	94.88%
eon	31.08%	22.08%	81.20%	65.94%
perlbmk	81.26%	68.92%	98.88%	89.19%
gap	30.83%	21.61%	75.59%	54.64%
vortex	99.52%	76.13%	100.00%	83.04%
bzip2	97.18%	86.68%	100.00%	97.13%
twolf	99.26%	86.72%	100.00%	97.11%
Average	85.42%	72.55%	96.20%	87.98%

Table A20. Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O3.

Program	$\frac{ V_i \cap V_d }{ V_d }$	$\frac{ E_i \cap E_d }{ E_d }$	$\frac{ V_f \cap V_d }{ V_d }$	$\frac{ E_f \cap E_d }{ E_d }$
	gzip	96.54%	86.48%	100.00%
vpr	99.22%	81.86%	100.00%	95.46%
gcc	97.33%	82.79%	99.72%	93.65%
mcf	93.40%	84.19%	100.00%	99.62%
crafty	99.37%	88.33%	100.00%	97.33%
parser	99.16%	83.11%	100.00%	95.95%
eon	30.49%	20.94%	81.48%	68.11%
perlbmk	80.44%	66.79%	90.79%	84.06%
gap	32.65%	23.02%	97.81%	71.39%
vortex	98.63%	76.41%	99.83%	85.74%
bzip2	96.91%	84.90%	100.00%	96.08%
twolf	99.24%	84.86%	100.00%	95.78%
Average	85.28%	71.97%	97.47%	90.05%

Table A21. Coverage of dynamic CFGs by IDA vs. FXE for the SPECint benchmarks compiled with GCC -Os.

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	85	166	95	46	115	247	100.00%	100.00%	21.05%	436.96%
vpr	276	1,046	315	131	333	1,338	100.00%	100.00%	5.71%	921.37%
gcc	1,487	9,221	2,176	3,021	2,436	14,510	100.00%	99.70%	11.95%	380.30%
mcf	56	81	62	31	78	123	100.00%	100.00%	25.81%	296.77%
crafty	140	843	181	322	199	1,943	100.00%	100.00%	9.94%	503.42%
parser	313	1,024	345	76	364	1,243	100.00%	100.00%	5.51%	1535.53%
eon	876	3,102	549	91	1,082	2,206	100.00%	100.00%	97.09%	2324.18%
perlbmk	506	2,066	640	1,238	840	4,330	100.00%	99.92%	31.25%	249.76%
gap	829	3,955	175	81	2,076	4,840	100.00%	100.00%	1086.29%	5875.31%
vortex	607	3,314	677	232	742	2,992	100.00%	96.98%	9.60%	1189.66%
bzip2	79	179	93	74	115	265	100.00%	100.00%	23.66%	258.11%
twolf	161	741	210	121	228	1,288	100.00%	100.00%	8.57%	964.46%
Average							100.00%	99.72%	111.37%	1244.65%

Table A22. IDA vs. FXE CFGs (indirect) for the SPECint benchmarks compiled with GCC -O1.

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	82	173	99	32	116	209	100.00%	100.00%	17.17%	553.12%
vpr	167	796	262	154	279	1,156	100.00%	100.00%	6.49%	650.65%
gcc	1,601	9,858	2,554	2,688	2,743	15,189	100.00%	99.93%	7.40%	465.07%
mcf	59	84	70	29	86	126	100.00%	100.00%	22.86%	334.48%
crafty	171	1,228	235	330	252	1,959	100.00%	100.00%	7.23%	493.64%
parser	342	1,106	405	88	424	1,262	100.00%	100.00%	4.69%	1334.09%
eon	703	2,761	489	88	869	1,624	100.00%	100.00%	77.71%	1745.45%
perlbmk	501	2,075	727	1,070	1,550	5,517	100.00%	100.00%	113.20%	415.61%
gap	907	4,210	206	77	1,098	3,381	100.00%	100.00%	433.01%	4290.91%
vortex	624	3,282	1,088	271	1,178	3,023	100.00%	100.00%	8.27%	1015.50%
bzip2	75	194	91	34	111	270	100.00%	100.00%	21.98%	694.12%
twolf	179	768	253	133	271	1,032	100.00%	100.00%	7.11%	675.94%
Average							100.00%	99.99%	60.59%	1055.72%

Table A23. IDA vs. FXE CFGs (indirect) for the SPECint benchmarks compiled with GCC -O2.

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	71	157	89	33	106	213	100.00%	100.00%	19.10%	545.45%
vpr	171	906	208	175	225	1,138	100.00%	100.00%	8.17%	550.29%
gcc	1,459	9,894	2,352	2,998	2,583	16,078	100.00%	100.00%	9.82%	436.29%
mcf	55	78	64	29	80	120	100.00%	100.00%	25.00%	313.79%
crafty	169	1,362	230	352	247	2,203	100.00%	100.00%	7.39%	525.85%
parser	208	1,022	250	95	268	1,300	100.00%	100.00%	7.20%	1268.42%
eon	696	2,741	480	88	856	1,610	100.00%	100.00%	78.33%	1729.55%
perlbmk	428	1,931	650	1,124	1,552	5,706	100.00%	100.00%	138.77%	407.65%
gap	902	4,309	190	95	1,571	5,648	100.00%	100.00%	726.84%	5845.26%
vortex	295	1,658	1,025	270	1,113	2,894	100.00%	100.00%	8.59%	971.85%
bzip2	54	157	65	37	81	289	100.00%	100.00%	24.62%	681.08%
twolf	159	719	224	137	242	1,171	100.00%	100.00%	8.04%	754.74%
Average							100.00%	100.00%	88.49%	1169.19%

Table A24. IDA vs. FXE CFGs (indirect) for the SPECint benchmarks compiled with GCC -O3.

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	76	149	87	35	103	228	100.00%	100.00%	18.39%	551.43%
vpr	183	972	214	177	231	1,247	100.00%	100.00%	7.94%	604.52%
gcc	1,201	7,665	1,802	1,835	1,984	12,691	100.00%	100.00%	10.10%	591.61%
mcf	55	79	62	28	78	128	100.00%	100.00%	25.81%	357.14%
crafty	139	850	181	295	199	1,941	100.00%	100.00%	9.94%	557.97%
parser	303	990	331	92	350	1,199	100.00%	100.00%	5.74%	1203.26%
eon	564	3,210	468	96	697	2,009	100.00%	100.00%	48.93%	1992.71%
perlbmk	433	1,967	545	880	1,079	5,056	100.00%	100.00%	97.98%	474.55%
gap	806	3,844	170	51	2,007	9,975	100.00%	100.00%	1080.59%	19458.82%
vortex	600	3,311	668	242	725	3,549	100.00%	100.00%	8.53%	1366.53%
bzip2	72	165	86	30	105	247	100.00%	100.00%	22.09%	723.33%
twolf	158	794	206	121	224	1,188	100.00%	100.00%	8.74%	881.82%
Average							100.00%	100.00%	112.07%	2396.97%

Table A25. IDA vs. FXE CFGs (indirect) for the SPECint benchmarks compiled with GCC -Os.

Program	$\frac{ V'_i \cap V'_d }{ V'_d }$	$\frac{ E'_i \cap E'_d }{ E'_d }$	$\frac{ V'_f \cap V'_d }{ V'_d }$	$\frac{ E'_f \cap E'_d }{ E'_d }$
	gzip	91.76%	12.05%	100.00%
vpr	97.46%	8.03%	100.00%	69.89%
gcc	94.55%	12.50%	99.87%	62.27%
mcf	87.50%	28.40%	100.00%	91.36%
crafty	95.00%	16.84%	100.00%	81.26%
parser	97.12%	2.73%	100.00%	74.90%
eon	37.10%	0.77%	82.53%	39.43%
perlbmk	81.42%	19.41%	89.13%	57.84%
gap	17.25%	1.31%	97.35%	18.94%
vortex	95.55%	4.50%	100.00%	34.55%
bzip2	92.41%	11.17%	100.00%	73.18%
twolf	95.65%	5.94%	100.00%	67.48%
Average	81.90%	10.30%	97.41%	62.50%

Table A26. Coverage of dynamic CFGs (indirect) by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O1.

Program	$\frac{ V'_i \cap V'_d }{ V'_d }$	$\frac{ E'_i \cap E'_d }{ E'_d }$	$\frac{ V'_f \cap V'_d }{ V'_d }$	$\frac{ E'_f \cap E'_d }{ E'_d }$
	gzip	90.24%	8.09%	100.00%
vpr	95.81%	9.55%	100.00%	64.82%
gcc	94.00%	11.45%	97.69%	55.45%
mcf	88.14%	25.00%	100.00%	89.29%
crafty	95.91%	11.89%	100.00%	55.46%
parser	97.37%	2.98%	99.42%	69.62%
eon	38.83%	1.01%	76.96%	25.24%
perlbmk	79.24%	18.31%	93.41%	52.92%
gap	17.09%	1.19%	60.42%	13.44%
vortex	95.67%	4.17%	99.68%	30.99%
bzip2	92.00%	6.70%	100.00%	69.59%
twolf	96.09%	8.07%	100.00%	61.72%
Average	81.70%	9.03%	93.97%	54.68%

Table A27. Coverage of dynamic CFGs (indirect) by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O2.

Program	$\frac{ V'_i \cap V'_d }{ V'_d }$	$\frac{ E'_i \cap E'_d }{ E'_d }$	$\frac{ V'_f \cap V'_d }{ V'_d }$	$\frac{ E'_f \cap E'_d }{ E'_d }$
	gzip	88.73%	8.92%	100.00%
vpr	95.91%	11.15%	100.00%	69.87%
gcc	93.21%	12.24%	97.60%	54.72%
mcf	87.27%	26.92%	100.00%	89.74%
crafty	95.86%	11.97%	100.00%	63.66%
parser	95.67%	3.33%	99.04%	64.19%
eon	37.79%	1.02%	76.58%	25.21%
perlbmk	78.50%	19.89%	97.20%	51.99%
gap	15.63%	1.18%	69.62%	13.00%
vortex	97.97%	4.22%	100.00%	31.00%
bzip2	88.89%	8.28%	100.00%	77.07%
twolf	95.60%	8.48%	100.00%	79.42%
Average	80.92%	9.80%	95.00%	57.49%

Table A28. Coverage of dynamic CFGs (indirect) by IDA vs. FXE for the SPECint benchmarks compiled with GCC -O3.

Program	$\frac{ V'_i \cap V'_d }{ V'_d }$	$\frac{ E'_i \cap E'_d }{ E'_d }$	$\frac{ V'_f \cap V'_d }{ V'_d }$	$\frac{ E'_f \cap E'_d }{ E'_d }$
	gzip	90.79%	10.07%	100.00%
vpr	96.17%	10.19%	100.00%	76.95%
gcc	93.92%	8.47%	99.83%	62.99%
mcf	87.27%	24.05%	100.00%	97.47%
crafty	94.96%	16.47%	100.00%	80.24%
parser	97.03%	4.14%	100.00%	76.26%
eon	47.87%	0.81%	72.70%	33.83%
perlbnk	81.06%	16.22%	94.46%	61.41%
gap	17.12%	0.68%	96.77%	19.46%
vortex	95.50%	4.50%	99.67%	40.44%
bzip2	90.28%	6.67%	100.00%	72.12%
twolf	95.57%	7.56%	100.00%	73.43%
Average	82.30%	9.15%	96.95%	64.48%

Table A29. Coverage of dynamic CFGs (indirect) by IDA vs. FXE for the SPECint benchmarks compiled with GCC -Os.