Control Dependencies in Interpretive Systems

Babak Yadegari and Saumya Debray

Computer Science Department
University of Arizona
{babaky,debray}@cs.arizona.edu

Abstract. Interpreters and just-in-time (JIT) compilers are ubiquitous in modern computer systems, making it important to have good program analyses for reasoning about such systems. Control dependence, which plays a fundamental role in a number of program analyses, is an important contender in this regard. Existing algorithms for (dynamic) control dependence analysis do not take into account some important runtime characteristics of interpretive computations, and as a result produce results that may be imprecise and/or unsound. This paper describes a new notion of control dependence and analysis algorithm for interpretive systems. This significantly improves dynamic control dependence information, with corresponding improvements in client analyses such as dynamic program slicing and reverse engineering. To the best of our knowledge, this is the first proposal to reason about low-level dynamic control dependencies in interpretive systems in the presence of dynamic code generation and optimization.

1 Introduction

Interpretive systems—interpreters, often accompanied by dynamic code transformation via just-in-time (JIT) compilers, together with input programs—are ubiquitous in modern computer systems. Their ubiquity and complexity make it important to devise algorithms to reason about such systems. An especially important analysis in this context is that of *control dependence*, which specifies whether or not one statement in a program controls the execution of some other statement. Control dependence analysis plays a fundamental role in a number of important program analyses and applications, including code optimization [16], program parallelization [28,31], program slicing [35,24,33], and information flow analysis [14,27].

This paper is concerned with low-level reasoning about control dependencies during an execution of an interpretive system. There are several reasons for considering the system holistically in this manner. The first is that the behavior of such systems is intimately tied to the control flow logic of both the interpreter and the interpreted program as well as the optimization logic of the JIT compiler. Second, in the presence of JIT compilation, a complete and accurate understanding of the execution behavior of such a system necessarily requires a low-level analysis at the level of IR or machine instructions, including instructions that may be

dynamically generated or modified (cf. Balakrishnan's work showing that source-level reasoning may not suffice for understanding low-level program behavior [6], which applies even more strongly to dynamically modified code).

Unfortunately, classical notions of control dependence are not very helpful for such reasoning about the execution behavior of interpretive systems; interactions between the interpreted program, the interpreter, and the JIT compiler; and code generated or modified dynamically. As a result, existing control dependence analyses produce results that can be imprecise when reasoning about the low-level behavior of such systems, including in particular input-specific interactions between the interpreter and JIT compiler (such interactions can give rise to bugs that can be difficult to track down [34,10]). The widespread use of interpreters and interpretive systems makes this a highly problematic situation. This paper takes a step towards addressing this problem by proposing an improved approach to control dependence analysis for interpretive systems.

This paper makes the following technical contributions. First, it identifies an important shortcoming of existing control dependence analyses in the context of modern interpretive systems. Second, it extends the classical notion of control dependence to interpretive systems in a way that handles a wide class of dynamic optimizations cleanly and uniformly. Experiments using a prototype implementation of our ideas, built on top of a Python system equipped with an LLVM-based back-end JIT compiler[36], show that our notion of control dependence significantly improves the results for client analyses such as dynamic program slicing. To the best of our knowledge, this is the first proposal to reason about low-level dynamic control dependencies in interpretive systems in the presence of dynamic code generation and optimization.

The rest of the paper is organized as follows: Section 2 discusses some backgrounds and motivating example. Section 3 discusses terminology and notation as well as the traditional notion of the control dependency. Section 4 presents our ideas and algorithms to handle control dependencies in the interpreters. This is followed by implementation details in Section 5 with evaluations and experimental results in Section 6. Section 7 discusses related work followed by conclusions in Section 8.

2 Background and Motivation

2.1 Interpreters

An interpreter implements a virtual machine (VM) in software. Programs are expressed in the instruction set of the VM, and encoded in a manner determined by the interpreter's architecture (e.g., byte-code, direct-threaded). Each operation x in the VM's instruction set is processed within the interpreter using a fragment of code called the handler for x (written $handler(x)$). The interpreter uses a *virtual instruction pointer* (vip) to access successive VM instructions in the input program and a *dispatch* routine to transfer control to appropriate handler code.

An interpreter must accommodate all control flow behaviors possible for all of its input programs despite having a fixed control flow graph itself. This is

done by transforming control dependencies to data dependencies through vip and controlling the execution by appropriately updating the vip value for the next VM instruction that should be executed. Control then goes from the handler to the dispatch code, which uses the updated vipto fetch the appropriate next VM instruction. In other words, control flow in the input program is handled via updates to vipwithin the interpreter dependeing on the semantics of the executed byte-code. As a result, control dependencies in the input program are transformed into data dependencies through vipin the interpreter's execution. This also means that the handlers are no longer control dependent on each other, but rather are all control dependent on the dispatch code.

2.2 JIT Compilation

Just-in-time (JIT) compilers are widely used in conjunction with interpreters to improve performance by compiling selected portions of the interpreted program into (optimized) code at runtime.

While the specifics vary from one system to another, the general idea is to use runtime profiling to identify frequently-executed fragments of code, then—once one or more such fragments are considered to be "hot enough"—to apply optimization transformations to improve the code. The very first such optimization is to transform the code from an interpreted representation, such as byte code, into native code. Some JIT compilers support multiple levels of runtime optimization, where the dynamically created code can be subjected to additional optimization if this is deemed profitable [30]. We refer to such dynamically-generated native code as "JITted code.''

For our purposes, JIT compilation can be seen as a sequence of alternations between two phases: execution, where code from the input program—which may or may not include previously JITted code—is executed; and optimization, where code transformations are applied to hot code to improve its quality. We refer to each such optimization phase as a *round* of JIT compilation.

2.3 Motivating Example

Dynamic slicing is helpful for fault location in debugging [38]. For simplicity we consider a small interpreter containing an obvious bug; in practice, interpreters are usually much larger and more complex, and even more so when JIT compilation is included.

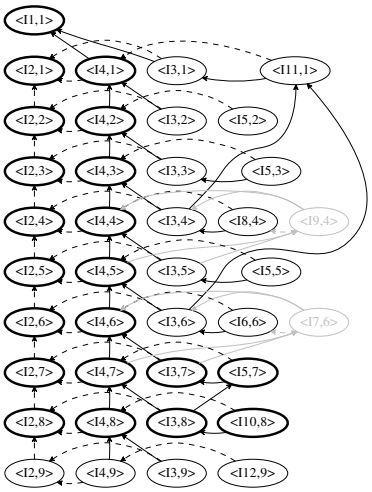
Consider the backward program slicing problem [24,2] posed for a simple interpreter, shown in Figure 1(b), executing the input program in Figure 1(a). Figure 1(c) shows the dynamic dependence graph (DDG) over the execution instances of the interpreter in Figure 1(b) with the input program in Figure 1(a) with an input other than 5. The nodes in the DDG graph represent execution instances of the statements in the interpreter program of Figure 1(b). Each node in the DDG graph contains a pair `<first,second>` where `first` corresponds to the statement in the interpreter code and `second` corresponds to the line number in the input program of Figure 1(a) that caused the `first` to be executed in the

```
1: n =
    input()
2: y = 1
3: x = 1
4: if n != 5
5:   y = 6
6: if n == 5
7:   x = 5
8: print x
9: halt
```

(a) An input program

```
I1: vip = 0;
I2: while(true)
I3:   op0, op1, dest =
        GetOperands(vip)
I4:   switch(InputPgm[vip++])
        case ASSIGNMENT:
I5:       *op1 = op0
          break
        case IF_EQ:
          /*== should be !=*/
I6:       if (op0 == op1)
I7:         vip = dest
          break
        case IF_NOT_EQ:
I8:       if (op0 == op1)
I9:         vip = dest
          break
        case PRINT:
I10:      print(op0)
          break
        case INPUT:
I11:      *op0 = input()
          break
        case HALT:
I12:      exit()
```

(b) A simple interpreter program



(c) DDG of the the interpreter in (b) interpreting input program (a)

Fig. 1. A motivating example

interpreter. Dashed edges in the graph show the control dependencies between statements in the code; solid edges show the data dependency and bold nodes represent the program statements that were included in the computed program slice. Gray nodes represent those statements that did not execute in this particular execution. We want to compute a slice with the criterion `(8, x)` of the input program in Figure 1.

There is a bug in the interpreter at $I6$ where the predicate for the conditional operator is wrong. Applying the slicing algorithm on the input program will correctly include lines 7, 6 and 1 in the slice because indeed the value of `x` at line 8 depends on all those statements. Suppose we want to do the slicing on machine level instructions executed by the interpreter, using dynamic program slicing algorithm [2] with the slicing criterion `(I10, op0)`. The algorithm starts from the node `<I10,8>` of the DDG that is the interpreter's code implementing the call to the `print` function of the input program. Nodes that are included in the program slice are made bold. According to the input program, node `<I6,6>` should be included because of a control dependency between lines 6 and 7 of the input program. However, this control dependence edge is missing in the DDG of the interpreter because `I5` is not control dependent on `I6` in the interpreter and so `<I6,6>` is not included in the computed slice.

Relevant slicing [3]—which includes predicates that did not affect the output but could have affected it were they evaluated differently—can improve slicing results where execution omission causes broken dependencies. Using relevant

slicing, node <I6,6> will be included in the slice because <I7,6> could have affected the output. However, for the same reason, node <I8,4> will also be included in the slice (the predicate at line 4 of the input program) because of node <I9,4>. So although relevant slicing helps including the missing statements in the slice computed by dynamic program slicing, it also includes irrelevant statements because of the data dependencies carried over the whole program by vipand this increases the size of the slice significantly for programs with larger sizes.

3 Terminology and Notation

Interpreters on modern computer systems often work in close concert with just-in-time (JIT) compilers to execute input programs.[1] To emphasize this, we refer to the combination of an interpreter and an associated JIT compiler as an *interpretive system*: an interpretive system with interpreter **I** and JIT compiler **J** is denoted by $\mathcal{I}(\mathbf{I}, \mathbf{J})$; where the interpreter and JIT compiler are clear from the context, or do not need to be referred to explicitly, we sometimes write the interpretive system as $\mathcal{I}$. The set of all possible execution traces of an interpretive system $\mathcal{I}$ on an input program P is denoted by $\mathcal{I}(P)$.

We assume a sequential model of execution. An *execution trace* for a program P is the sequence of (machine-level) instructions encountered when P is executed with some given input. A *dynamic instance* of an instruction x in an execution refers to x together with the runtime values of its operands at some specific point in the execution when x is executed. An instruction x in the static code for P may correspond to many different dynamic instances in an execution trace for P (e.g., if x occurs in a loop); where necessary to avoid confusion, we use positional subscripts such as x_m, to refer to a particular dynamic instance of an instruction x. We use $\prec$ to denote the sequential ordering on the instructions in a trace: thus, $x \prec y$ denotes that x is executed before y.

The idea of control dependence characterizes when one instruction controls whether or not another instruction is executed. This notion is defined formally as follows [16]:

Definition 1. Static Control Dependence: *y is* statically control dependent *on x in a given* control flow graph *G (written $y \overset{static(G)}{---\!\!\rightarrow} x$) if and only if:*

1. *there is a path π in G from x to y such that for every z on π ($z \neq x, y$), y post-dominates z; and*
2. *y does not post-dominate x in G.*

In addition to static control dependency which reasons about the program statically, Dynamic Control Dependence [37] reasons about control dependencies between program statements over a particular execution of the program.

[1] There may be additional software components in the runtime system, e.g., a profiler to identify hot code that should be JIT-compiled, a garbage collector, etc. For the purposes of this paper we focus on the interpreter and the JIT compiler.

Definition 2. Dynamic Control Dependence: *y_n is* dynamically control dependent *on x_m in a given execution trace and control flow graph G (written as $y_n \xrightarrow{dynamic(G)} x_m$) if and only if: (i) $x_m \prec y_n$; (ii) $y_n \xrightarrow{static(G)} x_m$; and (iii) for all z_k such that $x_m \prec z_k \prec y_n$ it is the case that $z_k \xrightarrow{static(G)} x_m$.*

The intuition here is that if y is control dependent on x, statically or dynamically, then one control flow successor of x always leads to y while the other may or may not lead to y.

Control dependence is, intuitively, a dynamic property, typically phrased informally as "one statement (or instruction) controlling whether another is executed." The reformulation of this dynamic property in terms of the structure of a static control flow graph, as in the definitions given above, rests on two implicit assumptions. The first, which we refer to as the *realizable paths assumption*, assumes that all "realizable" paths in a program—i.e., all paths subject to the constraint that procedure calls are matched up correctly with returns—are executable; or, equivalently, that either branch of any conditional can always be executed. This assumption, which Barth refers to as "precision up to symbolic execution" [8], is standard in the program analysis literature and is fundamental to sidestepping the undecidability problems arising from Rice's Theorem [20]. The second assumption is that the static control flow graph of the program contains all of the control flow logic of the computation.

The first of these two assumptions is arguably applicable to interpreters and interpretive systems. However, the second assumption does not hold in interpretive systems. There are two reasons for this: first, the logic of the input program necessarily influences the control flow behavior of an interpreter's computation; and second, JIT compilation can introduce new code at runtime whose control flow behavior is not accounted for in such definitions.

4 Control Dependence in Interpretive Systems

4.1 Semantic Control Dependency

To account for these aspects of interpretive computations that are problematic for the traditional notion of control dependency, we adapt the notion of control dependency in two ways. First, instead of considering all possible executions of the interpreter on all possible input programs—which is what we get from the traditional notion of control dependence applied to the control flow graph of the interpreter—we focus on all possible executions of the interpreter for a given input program being interpreted; this is helpful in our context because our goal is to improve the results of dynamic analyses of interpretive systems. Second, instead of tying our notion of control dependence to a fixed static control flow graph—an approach that does not work in the presence of dynamically generated code—we give a semantic definition in terms of execution traces.

Definition 3. Semantic Control Dependence: *Given an interpretive system $\mathcal{I}$ and an input program P, let x_m and y_n be dynamic instances of instructions*

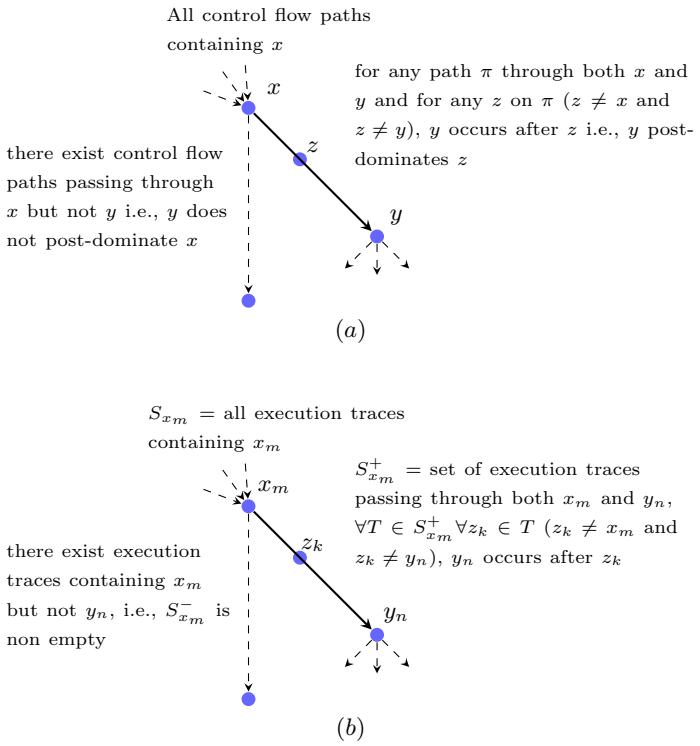


Fig. 2. Parallels between Static and Semantic Control Dependence; (a) Static control dependence; (b) Semantic control dependence

on some execution of $\mathcal{I}(P)$ *such that* $x_m \prec y_n$. *Let* $S_{x_m} \subseteq \mathcal{I}(P)$ *be the set of all execution traces in* $\mathcal{I}(P)$ *that contain the instruction instance* x_m. *Then* y_n *is* semantically control dependent *on* x_m *(written* $y_n \overset{semantic}{---\!\!\rightarrow} x_m$*) if and only if* S_{x_m} *can be partitioned into two nonempty sets* $S^+_{x_m}$ *and* $S^-_{x_m}$ *satisfying the following:*

1. $y_n \in S^+_{x_m}$ *and* $y_n \notin S^-_{x_m}$
2. $\forall z_k$ *such that* $x_m \prec z_k \prec y_n$, $z_k \in S^+_{x_m}$ *and* $z_k \notin S^-_{x_m}$

This definition parallels the traditional definition of control dependence, as shown in Figure 2. The notion of static control dependence (Definition 1), shown in Figure 2(a), uses the structure of the static control flow graph of the program to express the idea that, in the static program code, when we consider the control flow paths that contain x, some execution paths from x lead to y while others do not. Figure 2(b) uses the notion of execution traces—the dynamic analogue of control flow paths—to express the idea, in the dynamic program code, when

we consider the set of execution traces that contain x_m, some traces from x_m lead to y_n while others do not; here, $S^+_{x_m}$ denotes the set of execution traces that lead from x_m to y_n while $S^-_{x_m}$ denotes those that do not.

This notion of semantic control dependence differs from the traditional notion of static control dependence in two crucial respects. First, it takes into account both the interpretive system $\mathcal{I}$ and the input program P. Second, it is not tied to a fixed static control flow graph and therefore can be used for situations, such as with JIT compilers, where the code executed changes at runtime.

Definition 3 is not computable as stated and so cannot be embedded directly into an algorithm for computing control dependencies. Instead, we use this definition to reason about the soundness of our approach by showing that, considered in conjunction with the realizable paths assumption, it specifies exactly the set of control dependencies that is computed by our definition.

4.2 Interpretive Control Dependencies

We use a mechanism called instruction origin functions to incorporate control dependence information from the input program into the determination of control dependencies in the (low-level) execution trace. The essential intuition here is as follows. Suppose that, for each instruction Q in the input program, we can determine the set of instructions in the interpreter implementing $handler(Q)$, then the instruction origin function maps x to the input program instruction Q that x "originated from" if one exists; and $\perp$ (denoting "undefined") otherwise. More formally:

Definition 4. *Given an instruction x in the execution of an interpreter, an* instruction origin function Γ *is defined as:*

$$\Gamma(x) = \begin{cases} Q & \text{if there exists an instruction } Q \text{ in the input program} \\ & \text{such that } x \in handler(Q); \\ \perp & \text{otherwise} \end{cases}$$

Conceptually, Γ can be thought of in terms of labels associated with instructions in the input program that are propagated into the interpreter code, with the handler code in the interpreter getting labeled with the label of the instruction it handles. Γ is a many-to-one mapping of dynamic executed instances of instructions to static instructions in the input program. An instruction Q in the input program may be used in different places in the code, resulting in $handler(Q)$ getting executed multiple times. In this case Γ maps each executed interpreter instruction to the appropriate instruction in the input program. Moreover, if input program instruction Q causes $handler(Q)$ to be executed multiple times (e.g. a loop in the input program), they are all mapped to the same input instruction.

Given an execution trace of an interpretive system $\mathcal{I}$ on an input program P, in order to reason about the control flow influences due to the computational logic of both $\mathcal{I}$ and P, we define a notion of *interpretive control dependence*

("i-control-dependence" for short) that combines control dependence information from $\mathcal{I}$ and P and any code that may be dynamically created/modified by the JIT compiler (if any):

Definition 5. i-control-Dependence: *Given an interpretive system $\mathcal{I}$ and an input program P, let x_m and y_n be dynamic instances of instructions on some execution of $\mathcal{I}(P)$ with k rounds of JIT compilation such that $x_m \prec y_n$ and let $\mathbf{J}_i$ be the JITted code added at round i of JIT compilation. We define $y_n \xrightarrow{interp} x_m$ if either one of the following hold:*

1. *both x_m and y_n are in JITted code, i.e., $\exists i \leq k : x_m, y_n \in \mathbf{J}_i$, and $y_n \xrightarrow{dynamic(\mathbf{J}_i)} x_m$; or*
2. *at least one of x_m and y_n is not in JITted code, and either $y_n \xrightarrow{dynamic(\mathcal{I})} x_m$ or $\Gamma(y_n) \xrightarrow{static(P)} \Gamma(x_m)$.*

The notion of i-control-dependency uses the idea of dynamic control dependence to capture the control dependencies between interpreter instructions and JITted code. However, previously defined notions of control dependencies cannot be applied directly to capture the logic of the input program. To capture the control dependencies resulting from the control flow logic of the input program in both the interpreter code and JITted instructions, i-control-dependency uses the origin function. The intuition is that for input program instructions Q and R in P where $R \xrightarrow{static(P)} Q$, if there are instructions x and y in the execution trace such that $\Gamma(x_m) = Q$ and $\Gamma(y_n) = R$, then the execution of y_n depends on the execution of x_m and hence they are *semantically* control dependent but this can not be inferred by only looking at the CFG of the interpreter. The reason is that x_m and y_n belong to handlers of the interpreter which are not control dependent according to the interpreter's CFG.

Since JITted code executes natively without any interpretation, in the case where $x_m, y_n \in \mathbf{J}_i$, i-control-dependency uses the standard notion of control dependency (through the CFG of $\mathbf{J}_i$) to identify control dependencies between them. This allows i-control-dependency to handle situations where the JIT-compiler transformations may not preserve control dependence relationships in the input program (e.g., loop unrolling, loop permutation). However, if one of the instructions belongs to $\mathbf{I}$ and the other belongs to $\mathbf{J}_i$, it is still required to use Γ to look up control dependencies in the input program.

The reason for using the static control dependence definition for the input program P is that the function Γ maps instructions in the execution trace to static instructions in the input program, not their execution instances. Static control dependencies can be computed from the static CFG of the input program and given an execution trace in $\mathcal{I}(P)$, dynamic control dependencies for interpreter instructions can be computed using the CFG of the interpreter and Definition 2. Likewise for JITted instructions, dynamic control dependencies can be computed by the CFG of the created code at runtime.

The relationship $\overset{\text{dynamic}(\mathcal{I})}{-\!\!-\!\!-\!\!-\!\!\rightarrow}$ contains all the control dependencies in the interpreter code and the runtime generated JITted code that are computed using the standard notion of control dependency, i.e., $y_n \overset{\text{dynamic}(\mathcal{I})}{-\!\!-\!\!-\!\!-\!\!\rightarrow} x_m$ means that for x_m, $y_n \in \mathbf{I}$; or, for x, $y \in \mathbf{J}_i$, y_n is dynamically control dependent on x_m; the former using the CFG of the interpreter and the latter from CFG of the runtime generated code.

We next give a soundness result for the notion of i-control-dependence: namely, that under the realizable-paths assumption in program analysis, the notion of i-control-dependence is identical to that of semantic control dependence.

Lemma 1. *Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let x_m and y_n be instructions in an execution trace in $\mathcal{I}(P)$. Then, $y_n \overset{semantic}{-\!\!-\!\!-\!\!\rightarrow} x_m$ implies $y_n \overset{interp}{-\!\!-\!\!\rightarrow} x_m$.*

Proof. By induction on the number of rounds $k \geq 0$ of JIT compilation.

Lemma 2. *Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let x_m and y_n be dynamic instances of instructions in an execution trace in $\mathcal{I}(P)$. Then, $y_n \overset{interp}{-\!\!-\!\!\rightarrow} x_m$ implies $y_n \overset{semantic}{-\!\!-\!\!-\!\!\rightarrow} x_m$.*

Proof. From definition 5, $y_n \overset{\text{interp}}{-\!\!-\!\!\rightarrow} x_m$ implies that either (1) $y_n \overset{\text{dynamic}(\mathcal{I})}{-\!\!-\!\!-\!\!-\!\!\rightarrow} x_m$; (2) $y_n \overset{\text{dynamic}(\mathbf{J})}{-\!\!-\!\!-\!\!-\!\!\rightarrow} x_m$; or (3) $\Gamma(y_n) \overset{static(P)}{-\!\!-\!\!-\!\!\rightarrow} \Gamma(x_m)$. In the first two cases, the lemma follows from the definition of dynamic control dependence (Definition 2) applied to the code of the interpreter or the JITted code; In the second case, we use the definition of Γ to apply the definition of static control dependence to the input program.

The following result is now immediate.

Theorem 1. *Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let x_m and y_n be instructions in an execution trace in $\mathcal{I}(P)$. Then, $y_n \overset{interp}{-\!\!-\!\!\rightarrow} x_m$ if and only if $y_n \overset{semantic}{-\!\!-\!\!-\!\!\rightarrow} x_m$.*

5 Implementation

We have implemented a prototype of our control dependence analysis algorithm in the context of the *Unladen-swallow*implementation of Python [36], an open-source integration of a Python interpreter with a JIT compiler. *Unladen-swallow*uses the LLVM compiler framework [25] to dynamically map frequently executed code to LLVM-IR, which is then optimized and written out as JIT-compiled native code. *Unladen-swallow*was chosen because it is built up on two popular components, Python interpreter and LLVM compiler that are widely used both by researchers and in industry. Furthermore, *Unladen-swallow*provides mechanisms

to control the JIT compiler behavior from the input program, e.g. forcing the JIT compilation of a piece of code, that simplifies the evaluation.

To obtain the CFG of the interpreter, we disassemble[2] the interpreter (CPython) and reconstruct its control flow graph, from which we compute (intra-procedural) control dependencies in the interpreter and the JIT compiler. To simplify the implementation effort of our prototype, our current implementation treats the LLVM code generator as an external library that is not included in this control flow graph; this is simply for convenience and there is nothing precluding the inclusion of the LLVM libraries if so desired.

To identify control dependencies among the native code instructions, we use the CFG of the byte-code produced by the CPython compiler and mark the instructions in the execution trace. We instrumented the LLVM back-end JIT compiler to produce CFG of the compiled code that can be used to identify control dependencies among the JITted instructions.

Usually debug information is enough to compute the Γ function to map the executed instructions to the VM byte-codes in the input program. Interpreters, as well as JIT compilers, need to keep some debug information that relates executed instructions to the source program statements to make step-by-step debugging possible. In order to do this, an interpreter needs to keep information about the source program to individual byte-codes, and so for each handler code, the information about the source is accessible through these debug information.

We have instrumented the CPython compiler used in *Unladen-swallow*to produce and emit control flow graph of the byte-coded input program. Moreover, specific parts in the interpreter handlers that implement control flow transfers in the input program were marked to map conditional jumps in the input program (bytecode) to the machine-level instructions. This helps identifying the actual predicates and control transfer instructions in the execution trace when a basic block in the input program is found control dependent on another one. This was a matter of simplicity and convenience, and is not in any way fundamental to the ideas presented here: other approaches, e.g., reverse engineering the byte code obtained from the front-end compiler, e.g. see [29], would have produced exactly the same results. The modified interpreter inserts markers at the beginning of each basic block where we can map instructions in the execution trace to basic blocks in the input program. This information combined with the mapping information allows us to find the control dependencies in the input program.

For the JIT compiler, on the other hand, we instrumented the LLVM back-end to generate the CFG of the dynamically generated native code. We cannot rely on debug information to extract control dependencies in the JITted code because the JIT compiler's transformations may not preserve control dependencies from the original code. The safest option is to have the JIT compiler dump the CFG of the compiled code and use that to identify control dependencies. We modified

[2] We currently use the `objdump` utility for disassembly, invoking it as '`objdump --disassemble --source`'; however, any other disassembler would work. The '`--source`' option allows us to identify control flow targets for indirect jumps corresponding to `switch` statements.

the LLVM compiler to dump the CFG for the native code that is produced in the last step. This did not require more than 20 lines of code since the compiler needs to produce the CFG anyway and we just dumped this information.

6 Experimental Results

We evaluated our idea of control dependency presented in this paper using *dynamic program slicing* by applying the i-control-dependency notion discussed in 4.2. Our experiments were performed on a machine with 2 $\times$ 2.60 GHz six-core Intel Xeon processors with 64GB of RAM running Ubuntu. We used a tracing tool built on top of Pin [26] to collect execution traces. The tracing tool records each instruction with some runtime information such as instruction address.

Two crucial properties of a desired program slice is to include the buggy parts of the code in the slice while keeping the slice size minimal. To evaluate the effects of our notion of control dependency, we apply dynamic program slicing algorithm where the control dependencies are computed using 1) i-control-dependence, and 2) standard control dependence. For the first approach, our dynamic slicing algorithm ignores the data dependencies caused by the vipof the interpreter.

Each experiment was done for two execution modes: *Pure Interpreter* and *Interpreter Plus JIT*. In Pure Interpreter mode, the input program was only interpreted whereas in the Interpreter Plus JIT execution mode, all or some part of the input program was JITted. Our hypothesis is that slices computed using standard notion of control dependencies may be incorrect due to missed control dependencies in the input program while being larger mainly due to spurious data dependencies though the vip. We used a prototype implementation described in Section 5 and ran experiments on a collection of Python programs including three samples resembling already known bugs both in the interpreter and the JIT compiler, as well as five Python scripts taken from standard Python libraries that are included in the Python distributed package.

6.1 Buggy Samples

From three buggy samples, two of them are reported in the Python bug tracking system at `http://bugs.python.org/`[3] and the third one is adapted from a reported bug in C# JIT compiler.[4] All the three samples have a common characteristic where the wrong behavior is because of a bug in the interpreter or the JIT compiler that is only triggered by a particular input program. We constructed examples of the bugs and executed them with the Python interpreter used in *Unladen-swallow*. For the Interpreter Plus JIT experiment, we annotated the code triggering the bug so as to force it to be JITted at runtime. The samples are representative of a class of issues that may arise due to the imprecision of the analysis discussed in this paper and share similar characteristics. The goal of

[3] Issues 4296 and 3720 can be found at `http://bugs.python.org/issue4296` and `http://bugs.python.org/issue3720` respectively.

[4] https://www.infoq.com/news/2015/07/NET46-bug2

the experiment is to use dynamic program slicing from the point in the program where the wrong behavior is observed and determine whether the buggy code (either in the interpreter or in the JIT compiler) is included in the slice or not.

Table 1 presents the slicing results for the all the three buggy samples. The first two rows show the slicing results for the Pure Interpreter execution mode. As it can be seen from the table, for the two issues we examined, the slicing algorithm based on control dependencies computed with traditional definitions failed to include the sources of the bugs in the program slice because of missing control dependencies in the input program that was necessary to accurately pin down the bug. In the contrary, using i-control-dependency for interpreters, slicing algorithm was able to realize precise dependencies which resulted in the sources of the bug being included in the computed slice.

	Issue No.	*Trace size*	*Std Control Dependency*		i-Control-Dependency	
Exec. Mode	*(bug id.)*	*(instrs)*	*Slice size*	*Bug found*	*Slice size*	*Bug found*
Pure Interpreter	python-3720	11,837	265 (2.23%)	×	192 (1.62%)	✓
	python-4296	6,028	318 (5.27%)	×	222 (3.68%)	✓
Interpreter+JIT	python-3720	248,816	258 (0.10%)	×	22,769 (9.15%)	✓
	python-4296	269,928	21,924 (8.12%)	×	19,861 (7.35%)	✓
	C#-1299	259,802	269 (0.10%)	×	20,608 (7.93%)	✓

Table 1. Program slicing results

The last three rows show the slicing results where the buggy code is JITted during the execution. For the two Python bugs, the actual bug is in the runtime prodcuced (JITted) code because the bug is in the interpreter, but the JITted code make calls to the interpreter. For the third sample, the bug is in the JIT compiler so an accurate slicing algorithm should include the compiler code that produces wrong result. As it can be seen from the table, the slicing algorithm using standard notion of control dependency fails to include those parts of the JITted code or the compilation step where the source of the bug is. This is because of missing control dependencies in the runtime generated JIT code. With standard control dependence analysis, the slicing algorithm is only able to identify data dependencies in the JITted code. i-control-dependency helps slicing algorithm to include the bug in the slice.

Table 1 also shows the size of the computed slices for both slicing algorithms. The sizes are given in both raw instruction numbers and normalized to the size of the code observed in the execution trace. Since we are computing dynamic program slice, the size of the static code that was observed in the execution trace was considered as the program size. It can be seen that the slice sizes using i-control-dependence are smaller. So although our definition adds more control dependencies coming from the input program, it prevents spurious and unnecessary data dependencies due to the vipto pollute the slicing results.

6.2 Python Library Samples

The second experiment involves five python scripts taken from widely used Python libraries found in their distribution package:

- *binhex*: is a module to encode and decode files in `binhex4` format
- *socket*: provides a low-level networking interface
- *zipfile*: is a module to manipulate files in `ZIP` format
- *StringIO*: provides a file-like class to read/write string buffers
- *HTMLParser*: is a simple `HTML` and `XML` parser library

All the above samples have significantly large and complex logic. We used these samples to show how commonly the slicing results are computed incorrectly when applied to an interpreter code or when JIT-compilation is involved. For each sample we manually marked some variables as the slicing criterion on the execution trace of the samples and carried out dynamic slicing algorithm starting from the slice criterion.

		Trace size	*Std Control Dependency*		*i*-Control-Dependency
Exec. Mode	*Program*	*(instrs)*	*Slice size*	*Missed Pred.*	*Slice size*
Pure Interpreter	*binhex*	37,565	4,798 (12.77%)	13	3,561 (09.47%)
	socket	47,929	6,307 (13.15%)	13	5,681 (11.85%)
	zipfile	63,268	9,636 (15.23%)	17	7,837 (12.38%)
	StringIO	20,458	2,458 (12.01%)	10	1,512 (07.39%)
	HTMLParser	104,192	21,027 (20.18%)	14	16,637 (15.96%)
Interpreter+JIT	*binhex*	303,031	27,348 (9.02%)	36	27,906 (09.20%)
	socket	342,634	34,655 (10.11%)	1	34,130 (09.96%)
	zipfile	337,070	35,215 (10.44%)	68	35,478 (10.52%)
	StringIO	295,877	28,013 (09.46%)	30	28,285 (09.55%)
	HTMLParser	354,758	41,578 (11.72%)	10	40,921 (11.53%)

Table 2. Program slicing results: pure interpreter

Table 2 presents the slicing results for the samples from the Python library. Missed predicates column, only given for the dynamic slicing using standard control dependency, shows the number of predicate instructions included in the slice computed using *i*-control-dependency but not included when standard control dependency was used. The inaccuracy is mainly due to situations similar to what was shown on Figure 1. For the Pure Interpreter case, missed predicate column indicates missing control dependencies in the input program that are reflected in the interpreter code, while for the Interpreter Plus JIT execution mode it indicates the missing predicates only in the JITted code. The larger the number of missed predicates is, the less accurate the computed slice is, because in addition to missing predicates in the slice, data dependencies to these predicates are also missing which increases the inaccuracy of the slice even more.

Similar to the analysis of buggy samples, the slice sizes are included in Table 2. As it can be seen from the table, the slicing algorithm based on *i*-control-dependency produces smaller slices even though it includes more control dependencies in the slice, which as mentioned before is mainly due to spurious data dependencies of the interpreter vip. Smaller slice size difference for the Interpreter Plus JIT execution mode is because JIT compilation optimizes away the spurious data dependencies through the vipmaking the resulting slices smaller.

The running time of the analysis mostly depends on the size of the execution trace . Running time for our three largest traces are 15.12, 37.61 and 201.23 minutes with trace sizes of 423,641,006, 696,851,223 and 1,886,810,614 instructions respectively. The trace sizes for the Interpreter+JIT case are significantly larger

than the Interpreter only execution mode which is the reason for the increase in the analysis running times.

7 Related Work

There is an extensive body of research on control and data dependence analysis, including: program representations for control and data dependencies [16,21]; frameworks for control dependence analysis [9]; handling control-flow features of modern program structures and reactive systems [13,4]; and efficient algorithms and representations for control dependence analysis [12,37]. None of these definitions or systems consider cases where the executing code can change dynamically due to JIT compiler optimization.

The issue of imprecision of analysis resulting from "overestimation" of control dependencies, which in the interpretive systems arises from the transformation of control dependencies in the original program to data dependencies through `vip` in the interpreter, has conceptual similarities with problems due to over-tainting that arise when dealing with implicit flows in the context of dynamic information flow analysis. The latter problem is discussed by Bao *et al.* [7] and Kang *et al.* [23], who propose algorithms for considering control dependencies selectively, i.e., disregarding dependencies that do not satisfy certain properties of interest. High-level conceptual parallels notwithstanding, the details of the problems are very different from those considered here, as are the proposed solutions.

There is a lot of work on analysis and optimization of interpreters and interpretive systems, but much of this focuses on individual components of interpretive systems—e.g., the input program [17,11], the interpreter [15,18], or the JIT compiler [5,19,1]. Research on partial evaluation has considered the effect of specializing interpreters with respect to their input programs and shown that this is essentially equivalent to compiling the input program [22,32].

8 Conclusion

Interpretive systems—the combination of interpreters and JIT compilers—are ubiquitous in modern software tools and applications. This ubiquity, combined with their complexity, makes it important to develop good algorithms for reasoning about their behavior, in particular with relation to control dependences. Unfortunately, existing algorithms fall short in this regard. This paper introduces a notion of "interpretive control dependence" that can be used to reason about computations of interpretive systems, including code dynamically generated by the JIT compiler. Experimental results show that this notion leads to significantly improved precision in client analyses such as dynamic slicing and CFG recovery.

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) under grants CNS-1115829, CNS-1145913, III-1318343, CNS-1318955, and CNS-1525820.

References

1. Adl-Tabatabai, A.R., Cierniak, M., Lueh, G.Y., Parikh, V.M., Stichnoth, J.M.: Fast, effective code generation in a just-in-time Java compiler. In: Proc. PLDI '98. pp. 280–290 (Jun 1998)
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: Proc. PLDI '90. pp. 246–256 (Jun 1990)
3. Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.: Incremental regression testing. In: ICSM. vol. 93, pp. 348–357. Citeseer (1993)
4. Amtoft, T., Androutsopoulos, K., Clark, D., Harman, M., Li, Z.: An alternative characterization of weak order dependence. Information Processing Letters 110(21), 939–943 (2010)
5. Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F.: A survey of adaptive optimization in virtual machines. Proceedings of the IEEE 93(2), 449–466 (2005)
6. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. Ph.D. thesis, Computer Science Department, University of Wisconsin, Madison (2007)
7. Bao, T., Zheng, Y., Lin, Z., Zhang, X., Xu, D.: Strict control dependence and its effect on dynamic information flow analyses. In: Proc. 19th ISSTA. pp. 13–24 (2010)
8. Barth, J.M.: A practical interprocedural data flow analysis algorithm. Communications of the ACM 21(9), 724–736 (1978)
9. Bilardi, G., Pingali, K.: A framework for generalized control dependence. ACM SIGPLAN Notices 31(5), 291–300 (1996)
10. Chen, H., Cutler, C., Kim, T., Mao, Y., Wang, X., Zeldovich, N., Kaashoek, M.F.: Security bugs in embedded interpreters. In: Proceedings of the 4th Asia-Pacific Workshop on Systems. p. 17. ACM (2013)
11. Clausen, L.R.: A java bytecode optimizer using side-effect analysis. Concurrency: Practice and Experience 9(11), 1031–1045 (1997)
12. Cytron, R., Ferrante, J., Sarkar, V.: Compact representations for control dependence. In: Proc. PLDI '90. pp. 337–351 (1990)
13. Danicic, S., Barraclough, R.W., Harman, M., Howroyd, J.D., Kiss, A., Laurence, M.R.: A unifying theory of control dependence and its application to arbitrary program structures. Theoretical Computer Science 412(49), 6809–6842 (2011)
14. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
15. Ertl, M.A., Gregg, D.: The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism 5, 1–25 (2003)
16. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9(3), 319–349 (1987)
17. Franz, M.: Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In: Mobile Object Systems: Towards the Programmable Internet, LNCS, vol. 1222, pp. 263–276. Springer (1997)
18. Gagnon, E., Hendren, L.: Effective inline-threaded interpretation of java bytecode using preparation sequences. In: Compiler Construction. pp. 170–184. Springer (2003)
19. Gal *et al.*, A.: Trace-based just-in-time type specialization for dynamic languages. In: Proc. 30th SIGPLAN Conference on Programming Language Design and Implementation. pp. 465–478 (2009)

20. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (1979)
21. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(1), 26–60 (1990)
22. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
23. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: Dta++: Dynamic taint analysis with targeted control-flow propagation. In: NDSS (2011)
24. Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters 29(3), 155–163 (1988)
25. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization. pp. 75–86 (2004)
26. Luk *et al.*, C.K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. ACM Conference on Programming Language Design and Implementation. pp. 190–200 (June 2005)
27. Masri, W., Podgurski, A., Leon, D.: Detecting and debugging insecure information flows. In: ISSRE 2004. pp. 198–209 (2004)
28. Midkiff, S.P.: Automatic parallelization: an overview of fundamental compiler techniques. Synthesis Lectures on Computer Architecture 7(1), 1–169 (2012)
29. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: Proc. 2009 IEEE Symposium on Security and Privacy (May 2009)
30. Smith, J., Nair, R.: Virtual machines: versatile platforms for systems and processes. Elsevier (2005)
31. Srinivasan, V., Reps, T.: Partial evaluation of machine code. In: ACM SIGPLAN Notices. vol. 50, pp. 860–879. ACM (2015)
32. Thibault, S., Consel, C., Lawall, J.L., Marlet, R., Muller, G.: Static and dynamic program compilation by interpreter specialization. Higher-Order and Symbolic Computation 13(3), 161–178 (2000)
33. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
34. Wang *et al.*, X.: Jitk: a trustworthy in-kernel interpreter infrastructure. In: Proc. USENIX conference on Operating Systems Design and Implementation. pp. 33–47 (2014)
35. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering 10(4), 352–357 (Jul 1984)
36. Wouters, T., Yasskin, J., Winter, C.: unladen-swallow: A faster implementation of python, `https://code.google.com/p/unladen-swallow/`
37. Xin, B., Zhang, X.: Efficient online detection of dynamic control dependence. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 185–195. ACM (2007)
38. Zhang, X., Gupta, N., Gupta, R.: A study of effectiveness of dynamic slicing in locating real faults. Empirical Software Engineering 12(2), 143–160 (2007)

A Appendix. Proofs of Theorems

Lemma 1. *Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let x_m and y_n be instructions in an execution trace in $\mathcal{I}(P)$. Then, $y_n \overset{semantic}{\longrightarrow} x_m$ implies $y_n \overset{interp}{\longrightarrow} x_m$.*

Proof. By induction on the number of rounds $k \geq 0$ of JIT compilation.

Base case. The base case is for $k = 0$, i.e., when there is no JIT compilation. The proof in this case is by contradition. Suppose that $y_n \overset{\text{semantic}}{\longrightarrow} x_m$ and $y_n \overset{\text{interp}}{\not\longrightarrow} x_m$. From Definition 5, $y_n \overset{\text{interp}}{\not\longrightarrow} x_m$ is equivalent to (1) $y_n \overset{\text{static}(\mathcal{I})}{\not\longrightarrow} x_m$; and (2) $\Gamma(y_n) \overset{\text{static(P)}}{\not\longrightarrow} \Gamma(x_m)$. We consider these two cases separately:

Case 1. For (i), based on Definition 1 if $y_n \overset{\text{static}(\mathbf{I})}{\not\longrightarrow} x_m$, then x_m is post-dominated by y_n in the interpreter $\mathbf{I}$, which means that in every execution trace of $\mathcal{I}(P)$, either both instructions x_m and y_n or none of them should be executed. Combining the realizable paths assumption with Definition 3, this means that $y_n \overset{\text{semantic}}{\not\longrightarrow} x_m$.

Case 2. If $y_n \overset{\text{static(P)}}{\not\longrightarrow} x_m$, then $\Gamma(I)$ is post-dominated by $\Gamma(y_n)$ in the input program P. Given the realizable paths assumption, this means that for every execution of the $\mathcal{I}(P)$, either both $\Gamma(x_m)$ and $\Gamma(y_n)$ (and hence x_m and y_n) or none of them should be executed. This simply means that $y_n \overset{\text{interp}}{\not\longrightarrow} x_m$ and according to (1) above, $y_n \overset{\text{semantic}}{\not\longrightarrow} x_m$.

It follows from this that $y_n \overset{\text{interp}}{\not\longrightarrow} x_m$ implies that $y_n \overset{\text{semantic}}{\not\longrightarrow} x_m$. This contradicts the hypothesis that $y_n \overset{\text{semantic}}{\longrightarrow} x_m$ and $y_n \overset{\text{interp}}{\not\longrightarrow} x_m$. Thus the theorem holds.

Inductive case. Assume that the theorem holds after k rounds of JIT-compilation and consider the program after $k+1$ rounds of JIT-compilation. Suppose that the $(k+1)^{st}$ round of JIT-compilation causes a set of new instructions $\mathbf{J}$to be added to the interpretive system.[5] Suppose that $y_n \overset{\text{semantic}}{\longrightarrow} x_m$; we want to establish that this implies that $y_n \overset{\text{interp}}{\longrightarrow} x_m$. Depending on which parts of the code x_m and y_n belong to, we have the following cases:

1. $I \notin \mathbf{J}$ and $y_n \notin \mathbf{J}$, i.e., neither I nor y_n were added in the $(k+1)^{st}$ round of JIT compilation. It follows from the induction hypothesis that $y_n \overset{\text{interp}}{\longrightarrow} I$ in this case.
2. $x_m \notin \mathbf{J}$ and $y_n \in \mathbf{J}$:

[5] Some instructions may be removed as well, e.g., because they are dead or unreachable. However, instructions that are removed in this way will not be considered for any subsequent control dependence queries, so we do not consider them.

(a) $\Gamma(x_m) \neq \bot$ and $\Gamma(y_n) \neq \bot$: this means that x_m and y_n originated in the code for instruction handlers in the interpreter. $y_n \overset{\text{interp}}{-\not\rightarrow} x_m$ therefore means that $\Gamma(y_n) \overset{\text{static(P)}}{-\not\rightarrow} \Gamma(x_m)$. This means that $\Gamma(x_m)$ is post-dominated by $\Gamma(y_n)$ in the input program. From the realizable paths assumption, this means that $y_n \overset{\text{semantic}}{-\not\rightarrow} x_m$. This is a contradiction. We therefore conclude that $y_n \overset{\text{interp}}{\longrightarrow} x_m$.

(b) $\Gamma(x_m) = \bot$ and $\Gamma(y_n) \neq \bot$: since $x_m \notin \mathbf{J}$, it belongs to the interpreter code. For the interpreter to transfer control to the JIT-compiled code, there must be an instruction K in the interpreter that transfers control to the JIT-compiled code. Given Definition 3 and the realizable paths assumption, this implies that y_n post-dominates K. Suppose $y_n \overset{\text{interp}}{-\not\rightarrow} x_m$, this means that $x_m \overset{\text{interp}}{-\not\rightarrow} K$ so for every execution of $\mathcal{I}(P)$, either x_m, K and subsequently y_n (because y_n post-dominates K) or none of x_m, K and y_n are executed together. This is contradictory to the hypothesis where $y_n \overset{\text{semantic}}{\longrightarrow} x_m$ thus $y_n \overset{\text{interp}}{\longrightarrow} x_m$ holds.

3. $x_m \notin \mathbf{J}$ and $y_n \in \mathbf{J}$: this is similar to the previous case.
4. $x_m \in \mathbf{J}$ and $y_n \in \mathbf{J}$This implies that $\Gamma(x_m) \neq \bot$ and $\Gamma(y_n) \neq \bot$. $\Gamma(y_n) \overset{\text{static(P)}}{-\not\rightarrow} \Gamma(x_m)$. This means that $\Gamma(x_m)$ is post-dominated by $\Gamma(y_n)$ in the input program, which means that $y_n \overset{\text{semantic}}{-\not\rightarrow} x_m$. This is a contradiction. We therefore conclude that $y_n \overset{\text{interp}}{\longrightarrow} x_m$.

Lemma 2. *Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let x_m and y_n be dynamic instances of instructions in an execution trace in $\mathcal{I}(P)$. Then, $y_n \overset{interp}{\longrightarrow} x_m$ implies $y_n \overset{semantic}{\longrightarrow} x_m$.*

Proof. From definition 5, $y_n \overset{\text{interp}}{\longrightarrow} x_m$ implies that either (1) $y_n \overset{\text{dynamic}(\mathcal{I})}{\longrightarrow} x_m$; (2) $y_n \overset{\text{dynamic}(\mathbf{J})}{\longrightarrow} x_m$; or (3) $\Gamma(y_n) \overset{static(P)}{\longrightarrow} \Gamma(x_m)$. In the first two cases, the lemma follows from the definition of dynamic control dependence (Definition 2) applied to the code of the interpreter or the JITted code; In the second case, we use the definition of Γ to apply the definition of static control dependence to the input program. Based on the Definition 5, $y_n \overset{\text{interp}}{\longrightarrow} x_m$ implies that: (1) $y_n \overset{static(\mathcal{I})}{\longrightarrow} x_m$; or (2) $\Gamma(x_m) \overset{static(P)}{\longrightarrow} \Gamma(y_n)$. We consider each case separately:

Case 1. $y_n \overset{static(\mathcal{I})}{\longrightarrow} x_m$**.** From the definition of static control dependence (Definition 1), we can conclude two facts:

– From one of the successors of x_m—call it namely C_{y_n}—execution will eventually reach y_n; and for all instructions K on paths from x_m to y_n through C_{y_n} such that $K \neq x_m, y_n$, y_n post-dominates K. Let $S^+_{x_m}$

be the set of execution traces of $\mathcal{I}(P)$ that include x_m and C_{y_n}: these executions necessarily include y_n, and since y_n post-dominates all of the other instructions K that lie on paths from x_m to y_n through C_{y_n}, it follows that y_n occurs after K on each execution trace along these paths. Finally, from the realizable paths assumption, $S^+_{x_m}$ is non-empty.

- y_n may not be executed for the successor(s) of x_m other than C_{y_n}. Let $S^-_{x_m}$ be the set of execution traces that do not include C_{y_n}. It follows that some of these traces do not include y_n.

This is equivalent to the definition of semantic control dependency. It follows that $x_m \xrightarrow{\text{semantic}} y_n$.

Case 2. Suppose that $\Gamma(x_m) = {x_m}'$ and $\Gamma(y_n) = {y_n}'$ (and so $x_m \in \Gamma^{-1}({x_m}')$ and $y_n \in \Gamma^{-1}({y_n}')$) in the input program. Based on Definition 1, $\Gamma(x_m) \xrightarrow{static(P)} \Gamma(y_n)$ means that in the input program P, for one of the successors of ${x_m}'$, namely $C_{{y_n}'}$, the execution will eventually execute ${y_n}'$, and ${y_n}'$ is not executed for other successors of ${x_m}'$. Let $S^+_{x_m}$ be the set of execution traces that include instructions in $\Gamma^{-1}(C_{{y_n}'})$ and hence include $\Gamma^{-1}({y_n}')$ and so y_n because ${y_n}'$ post-dominates $C_{{y_n}'}$ in the input program. Similarly, $S^-_{x_m}$ be the set of execution traces that do not include $\Gamma^{-1}(C_{{y_n}'})$ and hence do not include $\Gamma^{-1}({y_n}')$ and y_n. This is equivalent to the definition of semantic control dependency and so we have $x_m \xrightarrow{\text{semantic}} y_n$.

So for both cases $y_n \xrightarrow{\text{interp}} x_m$ implies $y_n \xrightarrow{\text{semantic}} x_m$ under the realizable paths assumption.