Efficient Online Detection of Dynamic Control Dependence

Bin Xin   Xiangyu Zhang
Purdue University
Department of Computer Science
West Lafayette, Indiana 47906

ABSTRACT

Capturing dynamic control dependence is critical for many dynamic program analysis such as dynamic slicing, dynamic information flow, and data lineage computation. Existing algorithms are mostly a simple runtime translation of the static definition, which fails to capture certain dynamic properties by its nature, leading to inefficiency. In this paper, we propose a novel online detection technique for dynamic control dependence. The technique is based upon a new definition, which is equivalent to the existing one in the intraprocedural case but it enables an efficient detection algorithm. The new algorithm naturally and efficiently handles interprocedural dynamic control dependence even in presence of irregular control flow. Our evaluation shows that the detection algorithm slows down program execution by a factor of 2.57, which is 2.54 times faster than the existing algorithm that was used in prior work.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools, Tracing*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Algorithms, Measurement, Reliability

Keywords

dynamic control dependence, dynamic post-dominance, dynamic program slicing, dynamic information flow, and irregular control flow.

1. INTRODUCTION

Control dependence, an important concept in many program analysis, captures the effects of predicate statements on path selection and thus program behavior. Informally, a statement s statically control depends on a predicate statement p if p directly decides whether s gets executed. Control dependence is widely used in a lot of applications such as program slicing [18], program understanding [13], information flow analysis[10], compiler optimizations [6], and so on. Over years, researchers have been steadily pursuing the goal of improving the definition and computation of control dependence from well structured programs [6] to programs with arbitrary control flow [4, 15, 3, 14], and from intraprocedural to interprocedural [7, 16].

While the aforementioned concept is known as *static* control dependence, *dynamic control dependence*, introduced with the notion of dynamic slicing [8, 2], reveals the runtime effects of executed predicate instances on the program behavior *within a single execution*. According to a recent study [20] of the fault localization effectiveness of dynamic slicing, dynamic control dependence is critical in containing the root causes of many program failures. Besides dynamic slicing, dynamic control dependence is increasingly drawing attention in many other applications. For instance, dynamic information flow [12, 11] is a very effective technique to track information leak and prevent zero-day attacks. Handling dynamic control dependence has been known as a great challenge in dynamic information flow. Data lineage [19] captures the set of relevant input elements given a specific output element. It is invaluable in facilitating scientists in verifying computational results. The acquisition of precise lineage, in many cases, hinges on accurately handling dynamic control dependence. In [21], Zhang et al. propose a technique that can handle execution omission errors. The key component of the technique is about aligning two highly similar executions: one is the original execution and the other is generated by flipping one predicate instance in the original execution. With the technique proposed in this paper, a very efficient online alignment algorithm can be developed. Furthermore, execution alignment has applications in debugging [22], de-obfuscation, preventing time-channel attacks, etc.

Despite the importance of efficiently capturing dynamic control dependence, existing algorithms fall short in various aspects. Currently, there are two types of algorithms: *online* and *offline*. Offline algorithms have been popular in the past, for example, in the previous research of dynamic slicing [2, 23, 17]. These algorithms require collecting control flow traces, which becomes infeasible as the execution exceeds a certain length of time, usually a few seconds. Given a statement execution s_i, representing the ith instance of statement s, offline algorithms backward traverse the execution trace and identify the latest p_j such that s statically control depends on p, denoted by $s \xrightarrow{scd} p$. This procedure makes handling recursive functions intrinsically difficult.

The difficulty can be demonstrated by the example in Figure 1. The code is presented on the left while an execution trace is presented in the middle with different indents representing different call stack frames. As we can see, $2 \xrightarrow{scd} 1$ and $3 \xrightarrow{scd} 1$. Offline

code	**trace**
f(n) {	1_1
1: if (n>0) {	2_1
2: f(n-1);	1_2
3: s1;	2_2
}	1_3
4: s2;	4_1
}	3_1
	4_2
	3_2
	4_3

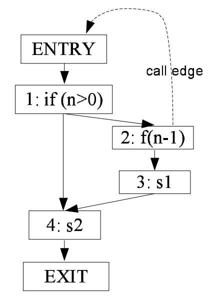


Figure 1: The Limitation of Offline Algorithms.

algorithms can easily recognize that $2_1 \xrightarrow{dcd} 1_1$, denoting 2_1 dynamically control depends on 1_1, and $2_2 \xrightarrow{dcd} 1_2$. Now assume we want to identify the controlling predicate of 3_2, backward traversal finds 1_3 while the correct answer is 1_1. The root cause is that the algorithm does not look for the latest execution of a static controlling predicate *in the same call stack frame*. Furthermore, simulating a call stack when traversing backward is difficult especially with the existence of irregular control flow such as `setjmp` and `longjmp`.

The second category is online algorithms. An online detection algorithm was presented in [20]. The algorithm couples control dependence detection with call stack maintenance on the fly, which solves the problem caused by recursive functions. However, it introduces new problems. First, the coupling causes a lot of problems for library functions which are compiled by various compilers or even hand written. Secondly, it is not efficient in handling interprocedural dynamic control dependence. It may violate the desired semantics of dynamic control dependence in presence of irregular interprocedural function calls. Finally, it is suboptimal in terms of runtime overhead.

The limitations of the existing algorithms, both online and offline, are rooted at the fact that computation of dynamic control dependence has been taken for granted as a straightforward runtime rendering of the static concept. There exists evidence showing that such a simple rendering is not efficient due to the different goals and properties between static and dynamic control dependences. For example, in program slicing, the static notion of *correctness* dictates that a slice preserves the original semantics of the program [14, 18, 3, 4], which is a major concern in defining static control dependence. In contrast, many applications of dynamic control dependence such as fault localization and dynamic information flow pay more attention to the cause effect relation between a predicate execution and its dependents. Moreover, while statically a statement may have multiple controlling predicates, a statement instance always dynamically control depends on one predicate instance.

In this paper, we present a refined definition of dynamic control dependence. Based on the refined definition, an efficient online detection algorithm is developed. The key observation is that the nested structure of the language constructs that causes dynamic control dependence to occur is very analogous to the nested structure of calling contexts. Thereby, a stack based technique is developed to efficiently handle dynamic control dependence.

The rest of this paper is organized as follows. In Section 2, an existing algorithm is discussed. The proposed technique is described in Section 3. In Section 4, the technique is extended to handle interprocedural dynamic control dependence. Experimental results are presented in Section 5. Related work is discussed in Section 6 and conclusions are made in Section 7.

2. EXISTING ONLINE DETECTION

Existing online detection algorithms are discussed in this section. For simplicity, only intraprocedural execution is considered. In other words, we assume the entire run is the execution of one function. Dependence caused by interprocedural execution will be discussed in later sections.

To the best of our knowledge, dynamic control dependence has not been explicitly defined. However, researchers [17, 20, 5] have been following the below informal definition to compute dynamic control dependence.

Note that an execution defines a total order for all the executed statement instances assuming only sequential execution is considered. The ith instance of a statement x is represented by x_i, $x_i < y_j$ if and only if x_i is executed before y_j.

DEFINITION 1 (**Dynamic Control Dependence**). *An execution instance x_i dynamically control depends on another instance y_j if and only if*

(1) $y_j < x_i$;

(2) $x \xrightarrow{scd} y$;

(3) $\nexists z_k \; s.t. \; y_j < z_k < x_i \;\; \wedge \;\; x \xrightarrow{scd} z.$

Algorithm 1 presents the detection algorithm used in [20], which is based on Definition 1.

Algorithm 1 Dynamic Control Dependence Detection Based On Definition 1

```
Predicate ($p_i$)
{
    SHADOW($p$) = <get_current_timestamp(), $p_i$ >;
}

GetControlDep ($s_j$)
{
    $max$= 0;
    $inst$=NULL;
    for (each $p$ such that $s \xrightarrow{scd} p$ ) {
        if ($max <$ SHADOW($p$).first) {
            $max$= SHADOW($p$).first;
            $inst$= SHADOW($p$).second;
        }
    }
    return ($inst$)
}
```

If a predicate statement is executed, function *Predicate()* is called. A *shadow* variable is allocated for each predicate statement, which can be accessed by calling function *SHADOW()*, which allocates two words for each static predicate, referenced by fields *first* and *second*. Inside function *Predicate()*, the current timestamp and the current predicate instance are stored in these fields.

In order to detect dynamic control dependence, function *GetControlDep()* is called upon the entry of each basic block. Inside the function call, the algorithm scans through the shadow variables of all the predicates that the instruction statically control depends on and identifies the one with the largest timestamp. The corresponding predicate instance is returned as the dynamic controlling predicate of s_i.

The inefficiency of the algorithm partly lies in the search for the predicate instance with the largest timestamp, especially when the instruction has multiple static controlling predicates. More limitations are imposed by interprocedural dependence detection.

An example of this algorithm will be given and explained in a later section.

3. EFFICIENT DETECTION OF INTRAPROCEDURAL DYNAMIC CONTROL DEPENDENCE

While static control dependence is computed by static control flow analysis which considers all possible program paths, at runtime, only one program path is taken, the executed one. The key observation is that dynamic control dependence contexts take on a stack-like structure that is analogous to calling contexts.

3.1 Definitions

In order to better present the idea, we consider only intraprocedural control dependence in this section. The discussion in this section is general to both structural and unstructural control flow. The unstructural control flow can be caused by `break`, `return`, and `goto` statements. Interprocedural dynamic control dependence will be discussed in Section 5.

DEFINITION 2 (**Dynamic Post Dominance**).
A statement instance x_i dynamically post-dominates y_j, denoted by $x_i \xrightarrow{dpd} y_j$ if and only if $y_j < x_i$ and x statically strict post-dominates y, denoted by $x \xrightarrow{spd} y$, and there does not exist x_k such that $y_j < x_k < x_i$.

Based on Definition 2, a new definition of dynamic control dependence is given as follows. The definition is equivalent to Definition 1 if only intraprocedural execution is considered, but it leads to a more efficient detection algorithm.

DEFINITION 3 (**Dynamic Control Dependence (NEW)**).
An execution instance x_i dynamically control depends on the largest $y_j < x_i$ if and only if x_i dynamically post-dominates any z_k in between y_j and x_i but not y_j.

THEOREM 1 (**Equivalence**).
Definitions 3 and 1 are equivalent.

PROOF. Assume $x_i \xrightarrow{dcd} y_j$ based on Definition 1. Let z_l be the largest statement instance satisfying $y_j < z_l < x_i$ and x_i does not dynamically post-dominate z_l. In other words, x statically post-dominates all the statements in between z_l and x_i but not z. Therefore, $x \xrightarrow{scd} z$ according to the definition of static control dependence. This contradicts $x_i \xrightarrow{dcd} y_j$ based on condition (3) in Definition 1.
The other way of the equivalence can be proved similarly. □

In order to develop a detection algorithm, next we introduce the concept of *region*. A statement s is said to be a **branching point** (BP), denoted by $\hat{s}$, if and only if s has more than one successors in the CFG. Predicate statements and `switch` statements are examples of BPs. Let $IPD(\hat{s})$ be the *immediate static post dominator* of $\hat{s}$, and thus an immediate dynamic post-dominator of $\hat{s}$ has the form of $IPD(\hat{s})_m$, in which m is just a place holder, indicating it is a dynamic instance of $IPD(\hat{s})$.
A *region* is defined as follows.

DEFINITION 4 (**Region**).
Given an executed instance of a BP $\hat{s}_i$, let $IPD(\hat{s}_i)_m$ be the immediate dynamic post-dominator of $\hat{s}_i$, the region directed by $\hat{s}_i$, represented by $R(\hat{s}_i)$, is defined as:
$R(\hat{s}_i) = \langle x_j \mid \hat{s}_i < x_j < IPD(\hat{s})_m \rangle$
$\hat{s}_i$ is called the director of the region.

In other words, $R(\hat{s}_i)$ is an ordered set that is comprised of the executed statement instances in between $\hat{s}_i$ and the first instance of the immediate post-dominator of $\hat{s}$ that is executed since.

PROPERTY 1 (**Region**).
Given any $x_j \in R(\hat{s}_i)$, the upper bound of the region, $IPD(\hat{s})_m$, dynamically post-dominates x_j.

PROOF. Assume $IPD(\hat{s})_m$ does not dynamically post-dominate x_j, then
(i). $IPD(\hat{s})$ does not statically post-dominate x,
or *(ii).* there is a $IPD(\hat{s})_n$ such that $x_j \le IPD(\hat{s})_n < IPD(\hat{s})_m$.
Assume *(i)* is true, there must be a program path from x to $EXIT$, which does not include $IPD(\hat{s})$. Therefore, $\hat{s} \rightsquigarrow x \rightsquigarrow EXIT$ is a path from $\hat{s}$ to $EXIT$ and it does not contain $IPD(\hat{s})$. This is contradictory to the precondition of $IPD(\hat{s})$ post-dominating $\hat{s}$ —- (1).
Assume *(ii)* is true, then $IPD(\hat{s})_n$ should be the immediate dynamic post-dominator of $\hat{s}_i$, which contradicts the condition of $IPD(\hat{s})_m$ being the upper bound of $R(\hat{s}_i)$ —- (2).
The combination of (1) and (2) completes the proof. □

According to Definition 4, each branching point instance during an execution directs a region and thus there are usually many regions in an execution.

THEOREM 2 (**Region**).
Regions in an execution are either disjoint or nested.

PROOF. Assume there are $\hat{s}_i < \hat{t}_j$, and they direct the regions of $R(\hat{s}_i) = (\hat{s}_i, IPD(\hat{s})_m)$ and $R(\hat{t}_j) = (\hat{t}_j, IPD(\hat{t})_n)$. Let us further assume these two regions overlap but they are not nested, such that,
$\hat{t}_j \in R(\hat{s}_i)$; (i)
$IPD(\hat{s})_m \in R(\hat{t}_j)$; (ii)
According to Property 1, $IPD(\hat{s})_m$ dynamically post-dominates $\hat{t}_j$.
Therefore, $R(\hat{t}_j) = (\hat{t}_j, x \le IPD(\hat{s})_m)$, which is a contradiction. □

THEOREM 3 (**Dynamic Control Dependence**).
A statement instance x_i is dynamically control dependent on the director of the smallest enclosing region.

PROOF. Let $x_i \xrightarrow{dcd} y_j$ based on Definition 3 and $R(\hat{r}_k) = (\hat{r}_k, IPD(\hat{r})_m)$ be the smallest enclosing region of x_i, now we prove $y_j \equiv \hat{r}_k$.
Assume $y_j < \hat{r}_k$, $x_i \xrightarrow{dpd} \hat{r}_k$ according to Definition 3. There must be a $IPD(\hat{r})_n \le x_i < IPD(\hat{r})_m$ such that $R(\hat{r}_k) = (\hat{r}_k, IPD(\hat{r})_n)$, which is a contradiction. Therefore $\hat{r}_k \le y_j$ —- (1).
Assume $\hat{r}_k < y_j$. Apparently, y must be a branching point. Besides, $IPD(\hat{r})_m$ dynamically post-dominates y_j according to Property 1. As a consequence, region $(\hat{y}_j, IPD(\hat{y})_n \le IPD(\hat{r})_m)$ is a smaller region than $(\hat{r}_k, IPD(\hat{r})_m)$. This region must contain x_i. Otherwise, according to Definition 3, x_i dynamically post-dominates $IPD(\hat{y})_n$ since $x_i \xrightarrow{dcd} \hat{y}_j$, and thus x_i dynamically post-dominates $\hat{y}_j$. This leads to a contradiction to Definition 3. Therefore, $y_j \le \hat{r}_k$ —– (2).
Combine (1) and (2), $y_j \equiv \hat{r}_k$. □

The merit of region is that it induces a very efficient online detection algorithm. Moreover, it provides better solutions for interprocedural dynamic control dependence and easily handles unstructural interprocedural control flow caused by `longjmp`, `exit`, and so on. Another benefit of the new definition is that it can be easily extended to accommodate indirect control dependence introduced in [13].

3.2 Online Detection Algorithm

Theorem 2 discloses that regions are either disjoint or nested, which is critical for devising the detection algorithm. A stack structure akin to call stack can be applied to maintaining nested regions and consequently dynamic control dependences. We call this stack the *control dependence stack* (CDS).

Algorithm 2 Detecting Intraprocedural Dynamic Control Dependence

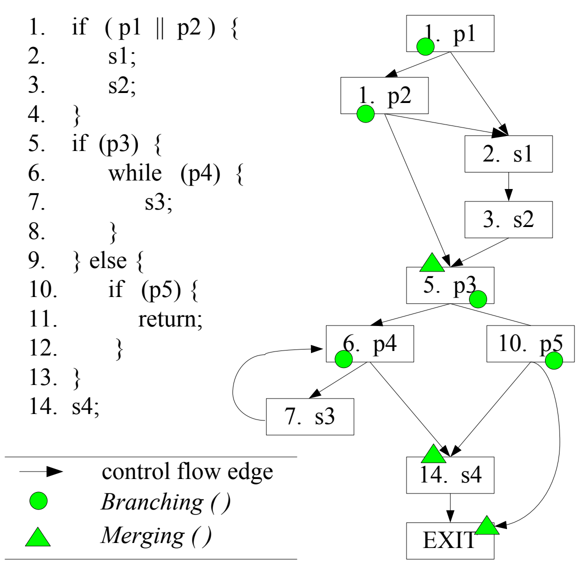
```
 1: Branching ($\hat{s}_i, IPD(\hat{s})$)
 2: {
 3:     if (CDS.top().second $\equiv IPD(\hat{s})$ {
 4:         CDS.top().first=$\hat{s}_i$;
 5:     } else {
 6:         CDS.push($< \hat{s}_i, IPD(\hat{s}) >$);
 7:     }
 8: }
 9: Merging ($t_j$)
10: {
11:     if (CDS.top().second $\equiv t$)
12:         CDS.pop();
13: }
```

Detecting region-based dynamic control dependence involves instrumentation at two kinds of statements – *branching points* (BPs) and *immediate post-dominators* (IPDs). Static control flow analysis is first applied to identify all the BPs and IPDs. As a BP is met during an execution, function *Branching( )* is called as illustrated in Algorithm 2. The first parameter is the current executed instance of the BP, the second parameter is the the IPD of the BP, the first instance of the IPD encountered serves as the termination point of the region directed by the BP.

Lines 3 and 4 present an important optimization of the algorithm. If two BP instances in the CDS share the same terminating static IPD, the two corresponding regions will be ended by the same IPD instance. Therefore, there is no need to maintain two entries in the CDS for the purpose of detecting dependence. Consequently, given a BP instance, the algorithm checks if it has the same terminating IPD as the top entry. If so, the top entry is simply replaced with the incoming BP instance. If it is not the case, at line 6, the algorithm simply pushes the current BP instance and the expected terminating IPD onto the CDS.

As an IPD is met, function *Merging ( )* is called. The function first checks if the IPD is the terminating IPD of the current region. As will be shown by an example, the first IPD met since the last BP may not be the IPD of the BP even though regions are always nested at a particular execution point. If it is the expected IPD, the top entry is popped from the CDS, which means the current region is left and the parent region is entered. The pop operation is implemented by simply substracting the stack pointer of CDS, which is very cheap.

Next, we use an example to demonstrate the algorithm. The program and its control flow graph are presented in Figure 2. Statements $s1$ and $s2$ have two controlling predicates – $p1$ and $p2$. Similarly, $s4$ has two controlling predicates, $p3$ and $p5$, because of the `return` at line 11. As shown by the figure, all the BPs are instrumented with calls to *Branching( )* and the IPDs are instrumented with calls to *Merging( )* . In contrast, if Algorithm 1 is used, BPs are instrumented by calls to *Predicate ( )* and the statements which



```
 1.  if  ( p1 || p2 ) {
 2.      s1;
 3.      s2;
 4.  }
 5.  if  (p3)  {
 6.      while  (p4)  {
 7.          s3;
 8.      }
 9.  } else {
10.      if  (p5) {
11.          return;
12.      }
13.  }
14.  s4;
```

Figure 2: An Example For Intraprocedural Dependence.

have more than one static controlling predicates are instrumented by calls to *GetControlDep ( )* .

Table 1 shows an execution trace of the example program and the instrumented executions for the two algorithms. For Algorithm 1, one extra assignment to the shadow variable is needed upon the execution of a predicate such as $\widehat{p1@1_1}$, $\widehat{p2@1_1}$, and $\hat{5}_1$. Since there are two predicates at statement 1, we use $p1@1$ and $p2@1$ to represent them. Comparisons of shadow variables are required on the execution of a statement which has more than one static controlling predicates such as 2_1, 3_1, 14_1.

For the region based Algorithm 2, a push onto the CDS or a replacement of the top entry is executed on the execution of a BP depending on whether the current executed BP instance having the same terminating IPD with the instance on top of the CDS. For instance, both $\widehat{p1@1_1}$ and $\widehat{p2@1_1}$ expect the same terminating IPD of 5, therefore at the second step of the execution, the top entry $< \widehat{p1@1_1}, 5 >$ is replaced with $< \widehat{p2@1_1}, 5 >$. This optimization is extremely important for the case of loops. At runtime, a loop creates as many nested regions as the number of executed iterations. For example, if the loop consisted of statements 6 and 7 gets executed for X times before it exits, it creates X entries on the CDS without the optimization. The CDS will overflow if the X is large enough. One important observation is that these regions will be terminated by the same IPD instance, and thus we only need to maintain the latest region. In other words, even though the loop may be executed for X times, there is always only one entry on the CDS.

Upon the execution of an IPD, Algorithm 2 first checks if the executed IPD is the expected terminating point of the current region. If it is, the top entry is popped. Popping the top entry in the case of a loop, such as the one in our example, has the effect of popping X nested regions at a time. Note that it is possible the executed IPD is not the terminating IPD of the current region. For instance, assume the execution takes the path $5_1 \rightarrow 10_1 \rightarrow 14_1$, Statement 14 is an IPD, which post-dominates BP 6. However, the current region is directed by 10_1, which will be terminated by $EXIT$. Therefore, the execution of 14_1 has no effect on the CDS.

In comparison with Algorithm 1, the cost of pushing an entry is similar to that of setting the shadow variables and the cost of

Table 1: Instrumented Runs for Algorithms 1 and 2

Trace	Algo 1 instrumentation	Algo 2	
		instrumentation	CDS
$p1@1_1$	$**S(p1) =< 1, p1@1_1 >$	push($< \widehat{p1@1_1}, 5 >$)	[$< \widehat{p1@1_1}, 5 >$]
$p2@1_1$	$S(p2) =< 2, p2@1_1 >$	replace top with ($< \widehat{p2@1_1}, 5 >$)	[$< \widehat{p2@1_1}, 5 >$]
2_1	$(S(p1).first > S(p2).first)$? $p1@1_1 : p2@1_1$	-	same as above
3_1	$(S(p1).first > S(p2).first)$? $p1@1_1 : p2@1_1$	-	same as above

5_1	$S(p3) =< 5, 5_1 >$	pop()
		push($< \hat{5}_1, EXIT >$)
			
			[$< \hat{5}_1, EXIT >$]

6_1	$S(p4) =< 6, 6_1 >$	push($< \hat{6}_1, 14 >$)	[$< \hat{5}_1, EXIT > \mid < \hat{6}_1, 14 >$]
7_1	-	-	as above
6_2	$S(p4) =< 8, 6_2 >$	replace top with ($< \hat{6}_2, 14 >$)	[$< \hat{5}_1, EXIT > \mid < \hat{6}_2, 14 >$]
14_1	$(S(p3).first > S(p5).first)$? $5_1 : NULL$	pop()	[$< \hat{5}_1, EXIT >$]
$EXIT_1$	-	pop()	[]

** $S(x)$ denotes the shadow variable of predicate x; $p1@1$ denotes the $p1$ sub-statement of 1 .

the pop operation is trivial. Algorithm 1 incurs extra overhead of accessing and comparing of timestamps in the case of a statement control depending on multiple predicates. In contrast, the nested structure of regions in Algorithm 2 efficiently handles the problem and thus requires no additional operations.

The acquisition of dynamic control dependence for each executed statement is omitted from the computation Table 1. For Algorithm 1, the dynamic depended predicate instance can be retrieved by looking at the shadow variable of the predicate if there is only one such predicate. Otherwise, it is the winner of the timestamp comparison. For Algorithm 2, the dynamic depended predicate is the director of the current region, which can be easily retrieved by looking at the top entry of the CDS. For example in Table 1, $\hat{6}_2 \xrightarrow{dcd} \hat{6}_1$ because $< \hat{6}_1, 14 >$ is the top entry at the moment that $\hat{6}_2$ gets executed.

4. DYNAMIC CONTROL DEPENDENCE IN INTERPROCEDURAL EXECUTION

In previous sections, we assume the whole execution is within a function. In practice, interprocedural control flow gives rise to interprocedural dynamic control dependence at runtime. It is not clear from previous work [17, 5] how interprocedural dynamic control dependence is handled. Part of the reason is that interprocedural dynamic control dependence is not as important as other type of dependences such as interprocedural data dependence in the application of dynamic slicing. However, it is no longer the case in the applications of dynamic information flow, data lineage, and dynamic data race detection. For example, missing a dynamic interprocedural control dependence may lead to leak of confidential information. In this section, we discuss computation of dynamic control dependence.

4.1 Detecting Dependence For Regular Interprocedural Control Flow

There exist two types of interprocedural control flow – *regular* and *irregular* flow. Regular interprocedural control flow is stemmed by normal `call` and `return` instructions, which maintain the LIFO property of the call stack. Irregular interprocedural flow, which is also referred to as *arbitrary* interprocedural control flow, is usually caused by `exit`, `longjmp`, exception handling, etc. It is often manifested as a callee not returning to the call site.

Interprocedural control flow has different properties than intraprocedural control flow, which make it hard to deal with. The first key difference is that interprocedural flow makes a statement always execute in certain context. Executions of the same statement in different contexts should be treated as executions of different statements, in other words, they should be context-sensitive.

code	**execution (1)**	**execution (2)**
f() {	1_1	1_1
1: if (P)	2_1	4_1
2: g();	6_1	6_1
3: else	5_1	5_1
4: g();	6_2	6_2
5: g();		
}		
g() {		
6: s1;		
}		

Figure 3: Context-sensitivity

For example in Figure 3, 1_1 taking either the `TRUE` branch or the `FALSE` branch eventually results in 6_1 being executed. However, it is wrong if one conclude that 6_1 does not control depend on 1_1. Instance 6_1 in execution (1) should be considered different from that in execution (2) because they have different contexts. In contrast, 6_2s have identical contexts.

The second key difference lies in that interprocedural flow has different semantics than intraprocedural flow. For example, `call` and `return` are often corresponding to each other. In other words, certain interprocedural flow is mandated by previous occurred flow. Therefore, turning interprocedural flow into intraprocedural by doing simple function inlining might lose some important semantics and eventually the accuracy in computing dynamic control dependence.

Our definition of interprocedural dynamic control dependence needs to accommodate these differences. First of all, we define the calling context of a statement instance.

DEFINITION 5 (**Calling Context**). *Given a statement instance s_i, its calling context, denoted by $CC(s_i)$, is an ordered list of call sites with the form $< c^1, c^2, ..., c^{j-1}, c^j, ..., c^n >$, in which s_i is executed in the function called at c^n and any c^j is in the function called at its predecessor c^{j-1}.*

Here superscripts are used to create unique symbolic variables.

DEFINITION 6 (**Dynamic Post Dominance (REFINED)**). *$y_j \xrightarrow{dpd} x_i$, if and only if associating x with $CC(x_i)$, all the possible program paths, including both intraprocedural and interprocedural, between x and the end of program pass y with $CC(y_j)$ and there does not exist $x_i < y_k < y_j$ that satisfies the aforementioned condition.*

In the execution trace presented in Figure 1, $CC(1_3) =< 2_1, 2_2 >$ and $CC(3_1) =< 2_1 >$. If 1 is associated with the calling context of $< 2_1, 2_2 >$, all the paths from 1 to the end of the program have to pass 3 with the context of $< 2_1 >$. Therefore, $3_1 \xrightarrow{dpd} 1_3$. Similarly, $3_2 \xrightarrow{dpd} 3_1$.

Note that it is impossible for two instances of the same statement $s_i \xrightarrow{dpd} s_j$ if only intraprocedural execution is considered. In the execution (1) of Figure 3, $6_2 \xrightarrow{dpd} 6_1$, $6_2 \xrightarrow{dpd} 1_1$, 6_1 does not dynamically post-dominate 1_1 because of the existence of the program path as demonstrated in execution (2).

Let $\hat{s}_i$ be a BP instance and $IDPD(\hat{s}_i)$ be its immediate dynamic post-dominator based on Definition 6. Note that previously we used $IPD(\hat{s})_m$ to denote the *intraprocedural* immediate dynamic post-dominator.

THEOREM 4 (**Calling Context**).
If only regular interprocedural control flow is considered, $CC(\hat{s}_i) \equiv CC(IDPD(\hat{s}_i))$.

PROOF. Assume $CC(\hat{s}_i) \neq CC(IDPD(\hat{s}_i))$. Therefore there are only two cases:
(i). $(CC(\hat{s}_i) \cap CC(IDPD(\hat{s}_i)) < CC(\hat{s}_i)$;
(ii). $CC(\hat{s}_i) < CC(IDPD(\hat{s}_i))$.

Here $\cap$ and $<$ denote the common prefix and comparison operations on ordered lists.

In case *(i)*, the executed path from $\hat{s}_i$ to $IDPD(\hat{s}_i)$ must encounter the context of $(CC(\hat{s}_i) \cap CC(IDPD(\hat{s}_i))$. In other words, the control flow has to return to the common calling ancestor of both $\hat{s}_i$ and $IDPD(\hat{s}_i)$. In other words, the return of the enclosing function of $\hat{s}_i$, denoted by $RET(\hat{s}_i)_j$, must satisfy $\hat{s}_i < RET(\hat{s}_i)_j < IDPD(\hat{s}_i)$. Moreover, since regular interprocedural control flow has only one return point for each function,
$RET(\hat{s}_i)_j \xrightarrow{dpd} \hat{s}_i$. It is contradictory to $IDPD(\hat{s}_i)$ being the immediate dynamic post-dominator —- (1).

In case *(ii)*, $CC(\hat{s}_i)$ is a prefix of $CC(IDPD(\hat{s}_i))$. Let
$CC(IDPD(\hat{s}_i)) \equiv CC(\hat{s}_i)| < c^t,\ c^{t+1},\ \ldots >$, , in which the executed instance c_j^t has the same context as $\hat{s}_i$ and $\hat{s}_i < c_j^t < IDPD(\hat{s}_i)$ Apparently, any program path, starting with $CC(\hat{s}_i)$, has to pass c^t with $CC(\hat{s}_i)$ in order to encounter $CC(IDPD(\hat{s}_i))$. Therefore, $c_j^t \xrightarrow{dpd} \hat{s}_i$. It is contradictory to $IDPD(\hat{s}_i)$ being the immediate dynamic post-dominator —- (2).

Combining (1) and (2), the theorem is proved. □

Based on this theorem, $IDPD(\hat{s}_i)$ is also an intraprocedural immediate dynamic post-dominator of $\hat{s}_i$ and therefore has the form of $IPD(\hat{s}_i)_m$.

Given the new definition of dynamic post-dominance. The definition of interprocedural dynamic control dependence remains identical to Definition 3. Since a region is still delimited by a BP and its immediate dynamic post-dominator, Theorem 3 and Theorem 2 hold. Proofs can be carried out in a similar way. Due to the space limit, they are omitted in this paper. The examples of interprocedural regions and dynamic control dependences can be found in Figure 1 and 3. In Figure 1, $R(\hat{1}_1) = (\hat{1}_1, 4_3)$. Therefore, $1_2 \xrightarrow{dcd} 1_1$ and $4_2 \xrightarrow{dcd} 1_1$. In the execution (1) in Figure 3, $6_1 \xrightarrow{dcd} 1_1$.

Based upon Theorem 4, the terminating point of a region has to be in the same calling context of the director of the region. The equivalence test of calling contexts can be efficiently performed by comparing the timestamped call stack frame base pointers.

Algorithm 3 Detecting Dependence With Regular Interprocedural Control Flow

```
 1: Branching ($\hat{s}_i$, $IPD(\hat{s})$, bp)
 2: {
 3:     if (CDS.top().second ≡ $IPD(\hat{s})^{bp}$ {
 4:         CDS.top().first=$\hat{s}_i$;
 5:     } else {
 6:         CDS.push(< $\hat{s}_i$, $IPD(\hat{s})^{bp}$ >);
 7:     }
 8: }
 9: Merging ($t_j$, bp)
10: {
11:     if (CDS.top().second ≡ $t^{bp}$)
12:         CDS.pop();
13: }
```

A refined algorithm which accommodates regular interprocedural control flow is presented by Algorithm 3. Variable *bp* represents the timestamped call stack pointer of $\hat{s}_i$. The algorithm labels the expected terminating IPD with *bp* and performs identity check before it pops an entry from the CDS. The last column of Table 2 presents an example of computing interprocedural dependence for Figure 1.

Effect On Intraprocedural Dependence. Interprocedural execution can affect computation of intraprocedural dynamic control dependence especially when functions are recursively called. If not handled correctly, recursive execution may result in wrong dependence.

Let us revisit the example in the introduction, Table 2 presents the computation table for the example trace. Assume we want to compute the controlling predicate of 3_2^{bp1}, in which the superscript *bp*1 is the calling context label. According to the previous Algorithm 1, it is retrieved from $S(1)$, the shadow variable of 3's static depended predicate. The result is 1_3, which is wrong. The correct answer would be 1_1. A similar effect is also observed in Algorithm 2.

The key to this problem is that the calling context should be taken into account when computing intraprocedural dynamic control dependence in case of interprocedural execution. In our previous work [20], we coupled the shadow variables in Algorithm 1 with the call stack by allocating shadow variables on the application's call stack, which can be achieved by manipulating stack pointers. In other words, the same predicate may have multiple shadow variables depending on if there exist multiple active calling contexts that contain the predicate. For example, as shown in the third column of Table 2, at the execution point of 1_3^{bp3}, predicate 1 has three active shadow variables, allocated in three active stack frames pointed by *bp*1, *bp*2, and *bp*3. At the moment of 3_2^{bp1} being executed, the shadow variable in the current calling context, which is referred to as $S^{bp1}(1)$, contains the depended predicate instance 1_1.

However, our experience shows that in practice it is very hard to manipulate the stack pointers for certain functions, especially when those functions are from libraries that may be compiled using various compilers or even written by hands. The reason is that allocating shadow variables on a stack frame entails identifying the particular instruction that allocates stack space for local variables

Table 2: Computation of The Execution In Figure 1.

Trace	Algo 1	Refined Algo 1	CDS in the Refined Algo 2
$**1_1^{bp1}$	$S(1) =< 1, 1_1 >$	$S^{bp1}(1) =< 1, 1_1 >$	$[< 1_1, 4^{bp1} >]$
2_1^{bp1}	-	-	same as above
1_2^{bp2}	$S(1) =< 3, 1_2 >$	$S^{bp2}(1) =< 3, 1_2 >$	$[< 1_1, 4^{bp1} > — < 1_2, 4^{bp2} >]$
2_1^{bp2}	-	-	same as above
1_3^{bp3}	$S(1) =< 5, 1_3 >$	$S^{bp3}(1) =< 5, 1_3 >$	$[< 1_1, 4^{bp1} > — < 1_2, 4^{bp2} > — < 1_3, 4^{bp3} >]$
4_1^{bp3}	-	-	$[< 1_1, 4^{bp1} > — < 1_2, 4^{bp2} >]$
3_1^{bp2}	-	-	same as above
4_2^{bp2}	-	-	$[< 1_1, 4^{bp1} >]$
3_2^{bp1}	-	-	same as above
4_3^{bp1}	-	-	[]

** the superscript describes the current stack frame pointer .

and/or parameters, usually by pattern matching. While code generated by commonly used compilers such as `gcc` has unique patterns, library functions, especially a lot of `libc` functions, do not follow these patterns.

The problem can be easily overcome by Algorithm 3. As shown by the last column in Table 2, the expected terminating IPDs of regions are labeled with stack pointers. For instance, the correct depended predicate of 3_2^{bp1} can be retrieved from the top entry of the CDS, which is 1_1.

Handling Function Pointers. Function pointers are used quite often in some C and C++ programs. They do not impose any new challenges to our technique. We treat the call sites of function pointers as branching points whose immediate post-dominators are the return sites. In other words, a call site that has more than one outgoing call edges and its return site delimit a region at runtime, all the executed statement instances within the region are directly/indirectly control dependent on the call site instance.

4.2 Detecting Dependence For Irregular Interprocedural Control Flow

The most intriguing cases happen when irregular interprocedural control flow occurs because of `longjmp`, `exit`, and so on. As observed in earlier work on static interprocedural control dependence [15, 3, 14], it is no longer true that a function has only one return point. A function may not return or have multiple return sites.

Irregular Flow Caused by `longjmp`. A typical type of irregular interprocedural control flow is caused by `setjmp` and `longjmp`, which is commonly used to implement exception handling and error recovery. Figure 4 presents a sample program with multiple functions and an irregular flow caused by `setjmp` and `longjmp`. Let us look at the following execution trace.

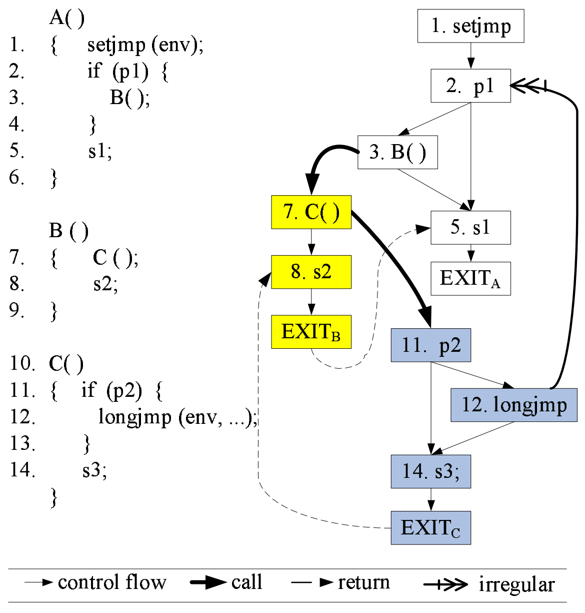
$$1_1\ 2_1\ 3_1\ 7_1\ 11_1\ 12_1\ 2_2\ 5_1;$$

A `longjmp` is performed in the execution. Following our definitions of dynamic post-dominance in Definition 6 and dynamic control dependence in Definition 3, instance $5_1 \xrightarrow{dpd} \hat{11}_1$ because all the possible paths from 11 to 5, if constrained by the calling contexts, have to take either the executed path or the path $11 \rightarrow 14 \rightarrow 8 \rightarrow 5$. Moreover, $5_1 \equiv IDPD(\hat{11}_1)$ because neither 12_1 nor 2_2 dynamically post-dominates 11_1. Therefore,

$$R(\hat{11}_1) \equiv (\hat{11}_1, 5_1) \equiv \{12_1, 2_2\}$$

, and thus $2_2 \xrightarrow{dcd} \hat{11}_1$.

The observation is that even though our definitions still accommodate irregular interprocedural flow. Theorem 4 no longer holds. In other words, a region may be bounded by instances with different calling contexts as demonstrated by $R(\hat{11}_1)$.



```
     A( )
1.   {    setjmp (env);
2.        if (p1) {
3.           B( );
4.        }
5.        s1;
6.   }

     B ( )
7.   {     C ( );
8.         s2;
9.   }

10.  C( )
11.  {   if (p2) {
12.          longjmp (env, ...);
13.      }
14.      s3;
     }
```

Figure 4: An Example For `setjmp` and `longjmp`

Our solution is demonstrated by Figure 5.

For the function that contains `setjmp`, a dummy edge is added between the call site of the function that may incur `longjmp` and the control flow successor of `setjmp`. For example, a dummy edge is added from 3 to 2 in the CFG of $A()$. This edge turns the call site into a *pseudo-predicate*, which is taken into consideration when computing post-dominance. However, it is not instrumented as a BP at runtime.

For functions that do not contain `setjmp` but may lead to a `longjmp`, a $DUMMY$ node is added and dummy edges are introduced between the `longjmp` or the call site leading to `longjmp` and the $DUMMY$ node, and between the $EXIT$ and the $DUMMY$ nodes. The modified CFGs for $B()$ and $C()$ are presented in Figure 5. The dummy nodes and edges affect computation of post-dominance but they do not correspond to any runtime instrumentation.

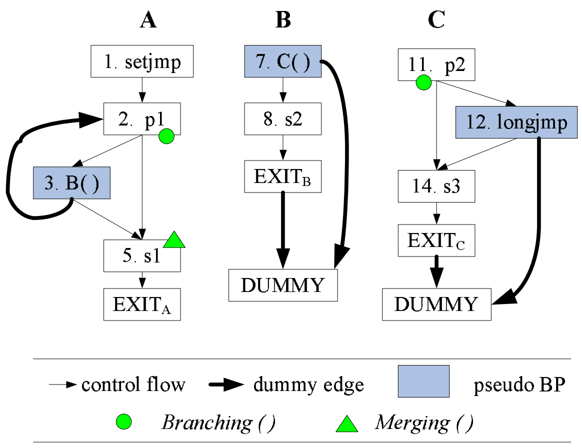


Figure 5: Handling `longjmp` flow

The fact that a BP is statically post-dominated by the $DUMMY$ node can be interpreted at runtime as the immediate dynamic post-dominator of the BP being the post-dominator of the pseudo predicate. For example, 11 has $DUMMY$ as its IPD. During the aforementioned sample execution, $5_1 \xrightarrow{dpd} \hat{1}1_1$.

Next we prove the correctness of this statement.

Let s be the pseudo-predicate in the procedure P with `setjmp`. Given an executed instance s_i, let $IPD(\hat{s})_m$ be the first instance of s's intraprocedural immediate post-dominator executed since.

THEOREM 5 **(Longjmp Flow).** *For any $\hat{s}_i < \hat{t}_j < IPD(\hat{s})_m$ such that t is a real BP in a procedure $G \neq P$ and $IPD(\hat{t}) \equiv DUMMY$, $IPD(\hat{s})_m \equiv IDPD(\hat{t}_j)$.*

The theorem says that $IPD(\hat{s})_m$ is the immediate dynamic post-dominators for all the executed predicates in between $\hat{s}_i$ and $IPD(\hat{s})_m$, whose IPDs are $DUMMY$ nodes. Therefore, $IPD(\hat{s})_m$ serves as the upper bound for not only $R(\hat{s}_i)$ but also all $R(\hat{t}_j)$s.

PROOF. According to Definition 6, we need to prove:
(i) $IPD(\hat{s})_m \xrightarrow{dpd} \hat{t}_j$;
(ii) There does not exist z_k in between $\hat{t}_j$ and $IPD(\hat{s})_m$ such that $z_k \xrightarrow{dpd} \hat{t}_j$

It is easy to prove that given any $\hat{s}_i < z_k < IPD(\hat{s})_m$, $IPD(\hat{s})_m \xrightarrow{dpd} z_k$ because all program paths from $\hat{s}_i$ to the end of execution have to reach $IPD(\hat{s})_m$ with $CC(\hat{s})$.

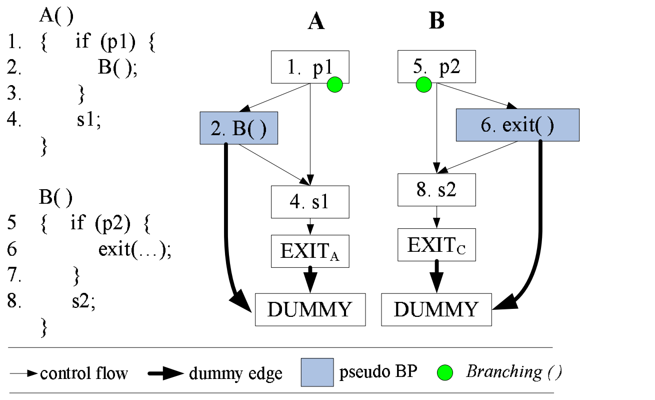
In order to prove *(ii)*, assume there is z_k such that $\hat{t}_j < z_k < IPD(\hat{s})_m$ and $z_k \xrightarrow{dpd} \hat{t}_j$. Let us further assume there is an edge between a call site c and $DUMMY$ in G. In other words, the function called at c may not return. Therefore, there exist a program path from $\hat{t}_j$ to $IPD(\hat{s})_m$, the calling contexts of the path follows the below pattern:
$CC(\hat{t}_j) \rightarrow CC(\hat{t}_j)| < c_l > \rightarrow CC(\hat{t}_j)| < c_l, ... > \rightarrow ... \rightarrow CC(\hat{t}_j)| < c_l, ... > \rightarrow CC(IPD(\hat{s})_m) \equiv CC(\hat{s}_i)$.

Since $IPD(\hat{t}) \equiv DUMMY$, there is a program path that does not include c. As a result, the calling context of $CC(\hat{t}_j)| < c_l, ... >$ never happens along that path.

According to the definition of dynamic post-dominance in Definition 6, *(a).* $CC(z_k) \equiv CC(t_j)$ or *(b).* $CC(z_k) \equiv CC(\hat{s}_i)$. Case (a) is impossible because $IPD(\hat{t}) \equiv DUMMY$. Neither is case (b) possible because $IPD(\hat{s})_m$ is the intraprocedural immediate post-dominator.

Therefore, z_k does not exist, which completes the proof. $\square$

This theorem explains why we do not instrument statement 14 or either of the $DUMMY$ nodes. Because we only need to check the termination of regions even those directed by 11_x at statement 5. The instrumentation remains almost identical as Algorithm 3. The only change required is to pass $IPD(\hat{s})$ as the expected termination point and the *bp* of $CC(\hat{s}_i)$ for t_js. Note that t_j and $\hat{s}_i$ belong to different functions.



```
   A( )
1. {   if (p1) {
2.         B( );
3.     }
4.     s1;
   }

   B( )
5  {   if (p2) {
6          exit(...);
7.     }
8.     s2;
   }
```

Figure 6: An Example For `exit`.

Irregular Flow Caused by `exit`. Another type of irregular interprocedural flow is caused by `exit` statements, which terminate program execution.

Figure 6 presents a sample program with multiple functions and an irregular flow caused by `exit`. Let us look at the following execution trace.

$$1_1 \ 2_1 \ 5_1 \ 8_1 \ 4_1$$

According to the definition of dynamic post-dominance, none of the execution instances after $\hat{5}_1$ dynamically post-dominates $\hat{5}_1$ because of the path $5 \rightarrow 6$. The same holds for $\hat{1}_1$. Therefore, all the execution instances after $\hat{5}_1$ directly or indirectly dynamically control depends on it. In other words, $8_1 \xrightarrow{dcd} \hat{5}_1$ and $4_1 \xrightarrow{dcd} \hat{5}_1$. Intuitively, the executions of 8_1 and 4_1 are controlled by the branch outcome of $\hat{5}_1$.

The observation is that *the CDS never needs to pop $\hat{5}_1$ or beyond*. Figure 6 shows how we deal with this case. In the figure, dummy edges are introduced to turn the IPD of $\hat{1}$ from 4 to $DUMMY$ and that of $\hat{5}$ from 8 to $DUMMY$. Here, the IPD of a BP being the $DUMMY$ node means that the corresponding $Branching()$ instrumentation flushes the entire CDS and pushes the current BP instance, which will never be popped by any $Merging()$. The previous Algorithm 3 can be easily extended to accommodate this case. The detail is omitted here for brevity.

5. IMPLEMENTATION AND EVALUATION

In this section, we discuss the implementation strategy and evaluate the efficiency of our algorithm for detecting dynamic control dependence. We also compare the overhead of the algorithm presented here with the previous algorithm that is based on static control dependence.

Our dynamic detection algorithm is integrated into user program through instrumentation. To build the instrumentation tool, we choose the Diablo/Fit [1] toolkit. Diablo is a link time binary

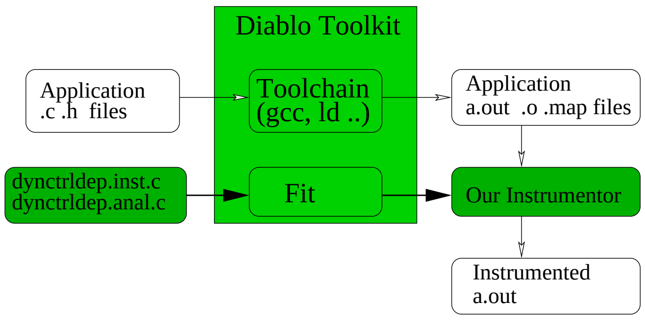


Figure 7: Tool Implementation and evaluation setup.

Benchmark	Base(s)	DCD(s)	Overhead
008.espresso	1.35	5.03	3.73x
124.m88ksim	0.18	0.64	3.55x
129.compress	115	255	2.22x
132.ijpeg	40	73	1.83x
164.gzip	3.7	12.6	3.41x
175.vpr	24	81	3.37x
181.mcf	90	127	1.41x
197.parser	23	52	2.26x
256.bzip2	36	71	1.97x
300.twolf	39	79	2.02x
Avg.	-	-	2.57x

Table 3: Overhead of our detection algorithm.

rewriting tool and Fit is an instrumentation tool generator based on Diablo. With Fit, we can code our detection algorithm in two separate `.c` files. We then feed these two files to Fit to generate our customized instrumentation tool. Our tool accepts application binaries and produces instrumented versions (In order to use Diablo, the binaries have to be produced by the compiler from the toolchain shipped with Diablo). One can then run the instrumented application to collect dynamic control dependence trace. This whole process is shown in Figure 7.

Implementing our algorithm is mostly straightforward. However, there are some difficulties that we have to overcome when recording the function calling contexts as discussed in Algorithm 3. Conceptually, since we are working on X86 architecture, the value of the base frame pointer, i.e. the *%EBP* register, could be used to denote a calling context. However, we found that in the assembly code, this register is sometimes manipulated explicitly, for example, being used to store a temporary value. This makes it unusable for our purpose. We handle this by using a global calling context ID that is increased (decreased) at function entry (exit).

For benchmarks, we use those from the Trimaran tool kit. It includes subsets of SPEC2000 and SPEC95. The infrastructure failed for the benchmarks `gcc` and `vortex`. All data are collected on an Intel Pentium III 1GHz machine with 500MB memory, running Gentoo Linux (kernel 2.6.14).

Efficiency. In order to show the efficiency of our algorithm, we compare the execution time of applications with and without dynamic control dependence detection code. The execution times are presented in Table 3. The *Base* column shows the execution time of code without instrumentation (in seconds), the *DCD* column with dynamic control dependence detection code. On average, it is about 2.57 times slower when running with the detection code. We think that this is generally an affordable price to pay. Furthermore, we are exploring ways inside Diablo that will allow us to selectively inline instrumentation functions. Since the instrumented code size is relatively small, especially for the `Merging` function, inlining may provide further room for reducing the runtime overhead of our algorithm. There are also optimization opportunities regarding loop entry basic blocks.

Comparison with the previous approach. In our previous work on dynamic control dependence detection, we devised an algorithm that is based on static control dependence data. As briefly discussed in Section 2, each branch point (BP) is assigned an ID and a timestamp. As the branch points are encountered during execution, their timestamps are updated. Each basic block's dynamic control dependence are determined by finding the largest timestamp among its static control dependent basic blocks. Basic blocks with zero or only one static control dependence do not need search for the largest timestamp.

We reimplemented this algorithm for this experiment. We use a separate stack to store the timestamps of branch points, as opposed to allocating space in the current function frame on the call stack, as suggested in the previous approach. We thus avoid the issue associated with library code. Branch points in each function are statically assigned an 10-bit ID (functions with more than 1024 branch points are the rare case), starting from 0. Upon method entry, one word is allocated for each branch point in that function on top of the stack. When dynamic dependence is determined for a block, its static dependent BPs' IDs are packed into words and passed to the `GetControlDep` function to find the BP with the largest timestamp. With 2 words, a maximum of 6 static dependences can be checked. Again, basic blocks with more that 6 static dependences are the rare case.

In order to do a fair comparison, interprocedural control dependence also needs to be detected for this algorithm. We achieve this by pushing an additional entry to the stack at a function call site. The entry will be the PC of the basic block that is the dynamic control dependence of the basic block invoking the call. Within the called function, basic blocks with 0 static control dependence will have this entry as their dynamic dependence.

As seen from the above discussion, our reimplementation strategy is fairly optimized. We implemented this algorithm in the same Diablo/Fit framework. The performance data are shown in Table 4. The *DCD* column corresponds to the algorithm in this paper, and the *Old* column corresponds to the previous algorithm. As seen from the table, our new algorithm are, on average, 2.54 times faster than the old algorithm. Both algorithms have to update some data structure when a branch point is executed, but for checking the dynamic control dependence, the new algorithm is much faster. In the new algorithm, the dynamic control dependence is always readily available on top of the control dependence stack, while in the *Old* approach, a number of comparisons are required, linear to the number of static control dependences of the current basic block.

6. RELATED WORK

In this section, we discuss some previous work on program control dependence, both static and dynamic.

Static Approaches. Among the static approaches, Ferrante et al. [6] studied the using of dependence graphs in compiler optimizations. They proposed the concept of *Program Dependence Graph* that combines both data and control dependence relations in one graph. They then demonstrated how certain compiler optimizations can be done more efficiently on this kind of graph. Our efforts toward clear definitions for dynamic post-dominance and dynamic control dependence are generalized from the definitions they adopted for

Benchmark	DCD(s)	Old(s)	Improvement
008.espresso	5.03	14	2.78x
124.m88ksim	0.64	1.98	3.09x
129.compress	255	657	2.58x
132.ijpeg	73	160	2.19x
164.gzip	12.6	37	2.94x
181.mcf	127	196	1.54x
197.parser	52	175	3.37x
256.bzip2	71	128	1.80x
Avg.	-	-	2.54x

Table 4: Comparison with our old detection algorithm.

the static counterparts.

Horwitz et al. [7] worked on dependence graphs in the context of precise static program slicing. They introduced what is called *System Dependence Graph* that combines both data and control dependence as well as interprocedural data dependence that is captured by the concept of *transitive flow dependence edges*, which in turn are computed by a technique borrowed from Attribute Grammar. Based on this graph, they developed an efficient two-phase slicing algorithm that curbs the loss of precision due to merged call sites in previous methods.

Sinha et al. [15, 16] extended the work by Horwitz et al. to specifically handle what they call *potentially non-returning call sites (PNRCs)*. They are caused by statements such as `exit`, `setjmp/longjmp`, `try/catch` in popular languages like C/C++ and Java. The way they handled these cases inspires part of our solution. We adopt a similar view when we define interprocedural dynamic control dependence in light of arbitrary interprocedural control flow in this paper.

Kumar et al. [9] worked on how to efficiently compute static source-level executable program slice in light of irregular control flow like `switch` and `goto` in C. They discussed a series of definitions, with each one refined over the previous one, for what constitutes as a *correct slice*, starting from a definition Weiser had in [18]. In the process, they discussed how these irregular flow can be handled. This process is very similar to our work presented in this paper except that we are working in the dynamic context and we are working on the binary level. Also, since we are not concerned about reproducing program fragments at source level, we can treat the irregular flow like `switch` and `goto` the same as other branching or fall-through basic blocks. These differences also apply when comparing our work with the all other static approaches.

Dynamic Approaches. Research on dynamic program control/data dependence are drawing increasing attentions. Wang et al. [17] addressed the space cost issue that is associated with collecting dynamic traces in dynamic slicing. They devised an algorithm to compress Java bytecode traces and demonstrated how to perform dynamic slicing by directly backward traversing the compressed traces. During the backward traversing, static control dependence information is consulted for finding dynamic control dependence for a bytecode. They did not discuss how interprocedural control dependence are discovered.

Vachharajani et al. [12] worked on architecture support for information flow analysis that is based on dynamic data and control dependence. In their proposed framework, security labels are assigned to data/control memory locations and these labels are propagated as instructions execute. Finally when data are written to certain channels (files, sockets etc.), their security labels are checked against a user policy. Besides data flow dependence, dynamic control dependence information is crucial for their work. They did not discuss how dynamic control dependence can be detected in their paper, so that our work here is a good compliment.

Zhang et al. [20] worked on using dynamic slicing in the context of fault location. In their work, efficient dynamic control dependence detection algorithms are required. The comparison was made clear in the main body of the paper.

In summary, we consider that our work presented in this paper a good compliment for some of the previous work done in this field. Furthermore, to our best knowledge, our formulation on dynamic control dependent and dynamic post-dominator relations, taking into consideration the interprocedural case, is the first effort toward such a goal.

7. CONCLUSIONS

In this paper, we introduce a novel definition of dynamic control dependence that accommodates both intraprocedural and interprocedural cases even in presence of irregular control flow. Based upon this definition, an efficient online detection technique is proposed. The experimental results show that our algorithm incurs only 2.57 times slowdown to program execution. Compared to the existing algorithm, it improves the performance by a factor of 2.54. Given the critical role of dynamic control dependence in many dynamic program analyses, our technique has the potential to foster more efficient new designs and implementations for those analyses.

8. REFERENCES

[1] http://www.elis.ugent.be/diablo/.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990.

[3] Matthew Allen and Susan Horwitz. Slicing java programs that throw and catch exceptions. In *Proceedings of the ACM SIGPLAN 2003 Workshop on Partial Evaluation and Semantics Based Program Manipulation*, 2003.

[4] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging*, pages 206–222, 1993.

[5] Arpad Beszedes, Tamas Gergely, and Tibor Gyimothy. Graph-less dynamic dependence-based dynamic slicing algorithms. *Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, 0:21–30, 2006.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, Atlanta, Georgia, United States, 1988.

[8] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[9] Sumit Kumar and Susan Horwitz. Better slicing of programs with jumps and switches. In *Fundamental Approaches to Software Engineering*, volume 2306, page 96, April 2002.

[10] Andrew Myers. Flow: Practical mostly-static information flow control. In *POPL '99: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[11] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, 2004. Technical Report CMU-CS-04-140, Carnegie Mellon University.

[12] N.Vachharajani, M.J.Bridges, J.Chang, R.Rangan, G.Ottoni, J.A.Blome, G.A.Reis, M.Vachharajani, and D.I.August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 31st annual ACM/IEEE International Symposium on Microarchitecture*, 2004.

[13] A. Podgurski and L.A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[14] V. Ranganath, T. Amtoft, A. Banerjee, M. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *ESOP 2005: The European Symposium on Programming*.

[15] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 432–441, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[16] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, 2001.

[17] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE'04:Proceedings of the International Conference on Software Engineering*, pages 512–521, Edinburgh, United Kingdom, 2004.

[18] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the International Conference on Software Engineering*, pages 439–449, San Diego, California, United States, 1981.

[19] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. Tracing lineage beyond relational operators, 2007. Technical Report, Purdue University.

[20] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, 2005.

[21] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI '07: Proceedings of the ACM SIGPLAN 2007 conference on Programming Language design and Implementation*, San Diego, CA, 2007.

[22] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 197–206, Lisbon, Portugal, 2005.

[23] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the International Conference on Software Engineering*, pages 319–329, Portland, Oregon, 2003.