Strict Control Dependence and Its Effect on Dynamic Information Flow Analyses

Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, Dongyan Xu
Department of Computer Science, Purdue University
{tbao,zheng16,zlin,xyzhang,dxu}@cs.purdue.edu

ABSTRACT

Program control dependence has substantial impact on applications such as dynamic information flow tracking and data lineage tracing (a technique tracking the set of inputs that affects individual outputs). Without considering control dependence, information can leak via implicit channels without being tracked; important inputs may be absent from output lineage. However, considering control dependence may lead to a large volume of false alarms in information flow tracking or undesirably large lineage sets. We identify a special type of control dependence called strict control dependence (SCD). The nature of SCDs highly resembles that of data dependences, reflecting strong correlations between statements and hence should be considered the same way as data dependences in various applications. We formally define the semantics. We also describe a cost-effective design that allows tracing only strict control dependence. Our empirical evaluation shows that the proposed technique has very low overhead and it greatly improves the effectiveness of lineage tracing and taint analysis.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Reliability, Security

Keywords

strict control dependence, control dependence, data dependence, dynamic information flow, taint analysis

1. INTRODUCTION

Program dependences are essential for a wide range of applications. Dynamic information flow tracking [20, 17, 11] prevents confidential information from being leaked to untrusted entities, e.g. it prevents user passwords from being sent out through a network connection, by monitoring propagation of information through program dependences. Taint analysis [2, 19] taints data received from outside. Taints are propagated through program dependences to mark the data that are affected by untrusted inputs. Once tainted data are used in critical places, such as pointer de-references or control flow transfers, warnings are produced to indicate that system integrity may be compromised. Slicing [21, 9] identifies the set of statements that influences the value of a given variable through program dependences, for the purposes of debugging, regression testing and so on. Lineage tracing [26, 3] computes the set of inputs that have contributed to the value of a given variable. It also relies on capturing program dependences.

Traditionally, program dependences are classified as data and control dependences [6]. Informally, a statement j is data dependent on i if and only if a variable is defined at i and then used at j. A statement j is control dependent on a predicate p if and only if the execution of j is directly decided by the branch outcome of p. Many of the aforementioned applications rely solely on data dependences. For instance, many taint analyses propagate taints through only data dependences. In other words, assume statements i and j are `i: x=...; ...; j: y=f(x);` and j is data dependent on i. Variable y gets tainted if x gets tainted. Similarly, most information flow tracking systems monitor only data dependences. For example, if x at i is confidential, y at j is confidential as well because of the data dependence. However, it is known that control dependence causes implicit information flow [11], which may lead to information leaks or integrity violations not being detected.

Consider the example in Fig. 1 (a). Variable `secret` contains confidential information. If the true branch is taken at line 2, there is information flow from `secret` to `output` because `output==1` dictates `secret==1000`. Hence, there is

```
1.  output=0;
2.  if (secret==1000) {
3.     ...
4.     output=1;
5.  }
6.  send(output);
```

(a) Strict Control Dep.

```
1.  output=0;
2.  if (secret>1000) {
3.     ...
4.     output=1;
5.  }
6.  send(output);
```

(b) Loose Control Dep.

Figure 1: Examples in information flow tracking.

information leak at 6. However, such leak can not be captured by considering only data dependences as the definition to `output` at 4 does not use any variables. In a recent study on the practicality of taint analysis [19], implicit information flow is noted as one of the dominant reasons for false negatives.

Simply considering control dependences is nonetheless problematic. The sample consequence in information flow and taint analyses is that substantial amount of data that is not closely relevant to the confidential (tainted) data is unnecessarily flagged as confidential (tainted). Consider the example in Fig. 1 (b). Assume the true branch of the predicate is taken. If the control dependence between statements 4 and 2 is considered, `output` is flagged as confidential and the packet send at 6 is considered as an information leak. However, there is hardly any information leak. In particular, from the observable packet sent at line 6, the attacker can only infer `secret>1000`. That is to say, the control dependence reveals very few information. We will show later in this paper: since such not-so-informative control dependences are so pervasive in programs, considering them leads to a large number of false positives. Similarly in lineage tracing, if such control dependences are considered, the lineage sets of many outputs contain almost the universal set of inputs even though most of the inputs may not be relevant to the outputs. Other researchers [2] also have similar observations.

Our observation is that while many control dependences should not be considered, otherwise false positives are introduced, there are some control dependences whose characteristics closely resemble those of data dependences and hence they should be considered the same way as data dependences. We call them the *strict control dependences* (SCD). The reason why most applications consider only data dependences is that data dependences often denote strong correlations. Consider the simple data dependence between i and j: `i: x=...; ...; j: y=x+1;`. We can observe that the value of y is closely related to the value of x. In particular, any change to x at i leads to change to y at j. From the value of y, we can precisely reverse engineer the value of x. We observe that such strong correlations are often manifested by some control dependences. Consider the variable `output` at line 4 and `secret` at line 2 in Fig. 1 (a) (assuming the true branch is taken). Although their correlation is introduced through control dependence, from the value of `output`, we can precisely decide the value of `secret`. Changing the value of `secret` to anything else leads to the other branch being taken and hence `output` containing a different value. Thus, it is a SCD. In comparison, the control dependence in Fig. 1 (b) is not a SCD as changing the value of `secret` does not necessarily change the branch outcome. We call them *loose* control dependences.

In this paper, we develop a technique to detect SCDs. In summary, we make the following contributions.

- We define the concept of strict control dependence and introduce its semantics. Based on the semantics, a static analysis is designed to identify predicate branches that give rise to SCDs. We find that SCD can be possibly induced by all comparative operators besides the "==" operator.
- We introduce instrumentation rules that detect dynamic SCDs during execution. We also demonstrate how these rules can be integrated into output lineage computation.
- We evaluate the effectiveness of SCDs by showing the sizes of the lineage sets computed. We compare the results with those considering only data dependences and those considering both data and classic control dependences. We develop a metric that can approximate the ideal lineage sets based on input fuzzing. We use the ideal lineage sets to show the false positives (FP) and false negatives (FN) of the different configurations. Our results show that considering only data dependences leads to high FN and considering both data and classic control dependences has high FP, whereas considering data and strict control dependences has low FN and low FP.
- We evaluate the cost of computing SCD, and compare it with other options. The results show that computing SCD slows down program execution by 76% while computing all control dependences slows down by 294%.

	[a]	[b]	[c]	[d]	[e]	[f]	[g]	[h]
[8]				rook	knight			king
[7]			pawn	bishop		rook	pawn	pawn
[6]								
···	···							

Figure 2: A chess board layout described by an input string "`3rn2k/2pb1rpp/`". In particular, letters `r`, `n`, `k`, `p`, and `b` denote `rook`, `knight`, `king`, `pawn`, and `bishop`, resp. Numbers indicate how many empty cells in between. Each '/'-delimited substring specifies the layout of a row.

2. MOTIVATING EXAMPLE

We use a real example in lineage tracing to demonstrate the effect of SCD. Lineage tracing identifies the set of inputs that contributes to individual output values. It has been used in data validation and debugging [26, 3]. One can consider it as a generalized form of taint analysis with the taint being a set of inputs. Consider `186.crafty` in SPECINT 2000. It is a high-performance artificial intelligence chess program. It takes chess board layout as the input and searches for best moves within bounded steps. A chess board layout is described by a string. For instance, a string such as "`3rn2k/2pb1rpp/`" describes a layout for the 8×8 chess board as shown in Fig. 2. Fig. 3 presents a fragment from the program that parses the input string and generates the chess board layout. In particular, the layout is stored in an array `tboard` of 64 elements. A dictionary `bdinfo` is used for interpreting each char in the input string. Array `firstsq` represents the starting index of each rank (row) in the `tboard` array. For example, as described by the values in line 8, the top row starts at index 56 in `tboard` and the second row starts at 48. Variables `whichsq` and `num` describe the rank position and column position of the current square, resp. The loop in lines 13-28 is responsible for parsing the input string. It traverses each char in the input string. For each char, in lines 14 and 15, it identifies the entry in the dictionary `bdinfo` that matches the input char, and stores it in variable `match`. If `match` equals to 24, meaning the input char is the end symbol of a rank, in lines 17-20, the column position is reset, the rank position is incremented, and the beginning index of the new rank is loaded. If `match` indicates

```
1   int tboard[64];                     //the board
2   char bdinfo[] =                     //the dictionary
3        { 'q', 'r', 'b', …,            //0-15, kinds of pieces
4          '1', '2', '3', …,            //16-23, the free squares
5          '/' };                       //24, end of a rank (row)
6
7    //the beginning index of each rank in 'tboard'
8   int firstsq[8]={56,48,…,0};
9   int whichsq=0;                      //rank(row) position
10  int num=0;                          //column position
11  square=firstsq[whichsq];            // current index in 'tboard'
12
13  for (pos=0; pos<(int) strlen(args[0]); pos++) {
14    for (match=0; ... args[0][pos]!=bdinfo[match];
15          match++);
16    if (match == 24)  {               //end of a rank, reset
17        num=0;
18        whichsq++;
19        if (whichsq > 7) break ;
20        square=firstsq[whichsq];
21    } else if (match >= 16) {         //numbers, adjusting indices
22        num+=match-15;
23        square+=match-15;
24    } else {                          //pieces, stored to tboard
25        … ++num;
26        tboard[square++]=match-7;
27    }
28  }
```

Figure 3: The `186.crafty` code snippet that parses the input string to chess board layout.

that the input char is a number, in lines 22 and 23, the current square is updated according to the specified number of free squares. If neither case is true, the input char must be a piece such that the corresponding array element is set at line 26.

Assume we want to compute the lineage of the elements in array `tboard` after the parsing phase, namely, the set of input values that affects the value of an array element. Consider the prior lineage computation [26] that relies on only data dependence. Note that all definitions to array `tboard` occur in line 26, using variables `square` and `match`. Variable `square` is defined at lines 11, 20, 23 and 26. The definition at line 11 relies on only the constant array `firstsq` and hence is not input relevant. The definition at line 20 is similar. The definition at line 23 uses variable `match`. Variable `match` is defined at line 15. However, the definition does not use any input relevant variables. The definition at 26 is similar. In other words, if only data dependence is considered, it has empty input lineage. Transitively, we can easily conclude that all the elements in `tboard` have empty lineage at the end of the parsing phase, which is clearly undesirable.

Next lets consider traditional control dependence in lineage computation. The execution of array definition at line 26 is control dependent on the predicate at 21, which in turn is control dependent on line 16 and the loop predicate at line 13. Note that the predicate at line 13 involves `strlen(args[0])` that uses the entire input string. Dictated by such a chain of control dependence, the lineage of each element in `tboard` contains the whole input string. However, such information is hardly useful.

We observe that some of the control dependences reflect *stronger* relevance compared to others. Consider the true branch of the predicate at line 16. Executing the statements inside the true branch, such as setting `num` to 0 at line 17, dictates `match` be 24, not any other value. And slightly changing the value of `match` must lead to the statement not being executed and `num` having a different value. The nature of such a strong relevance is very similar to a data dependence, in which change of the source variable leads to a different value of the destination variable. In comparison, if the false branch is taken, the correlation of the statements executed inside the branch and the predicate is weak because there are many values that satisfy `match!=24` and changing the value of `match` may not prevent their execution.

Leveraging such an observation, we propose to quantify the strength of control dependence and consider only strict control dependence. Next, we illustrate the idea by computing the lineage of `tboard[50]` for the previous input string, which corresponds to the `pawn` at square `c7` in Fig. 2 and the 8th char in the input string. From the code snippet in Fig. 3, we can tell that the value at `tboard[50]` is defined when the 8th input char is processed (during the 8th iteration of the main loop). Inside the iteration, the input char '`p`' is compared with the dictionary entries in lines 14 and 15 until the entry `bdinfo[6]`=='`p`' is found, reflected by the predicate `args[0][pos]!=bdinfo[match]` taking the false branch. It is a strict control dependence because making any changes to the variables involved (i.e. `pos` and `match`), the opposite branch is taken. Hence, the lineage of `match` when the inner loop exits contains the 8th char if we consider SCD. The lineage is later propagated to `tboard[50]` via the definition at 26.

Furthermore, the definition at 26 also uses variable `square` and hence the lineage of `tboard[50]` also involves those inputs propagated through `square`. The 7th input char '`2`' processed in the previous iteration lead to the increment of `square` in line 23. Hence, it is in the lineage set of `square` and then it flows to the lineage set of `tboard[50]`. Note that `square` is initialized for each new rank at line 20, which is strictly control dependent on line 16. Line 16 uses `match`, which is strictly control dependent on the last loop predicate instance at 14 that involves the 6th input char '`/`'. Hence, the lineage of `tboard[50]` contains the 6th, 7th and 8th chars ("`/2p`"). Indeed, the 6th and 7th chars decide the position of the square on the board and the 8th char decides the piece of the square. They are clearly strongly correlated to the piece at `c7` on the board. Changing any of these input chars will end up changing that piece. In contrast, changing other input chars does not change the piece. Compared to the empty lineage set generated by considering only data dependences, and the lineage set that is the universal set of inputs generated by also considering all control dependences, the lineage set generated by considering SCDs is more informative and more relevant.

3. EPSILON SEMANTICS

The goal of strict control dependence is to discover strong correlations between statements and the predicates that guard the statements. The formal definition of SCD is given as follows.

DEFINITION 1. *A statement s is strictly control dependent on a predicate $p : e_1 \diamond e_2$ with $\diamond$ a comparative operator, de-*

noted as $s \xrightarrow{scd} p$*, if and only if:*
(1) p *directly or transitively guards the execution of* s*;*
(2) the execution of s *infers that* $e_1 \equiv e_2 + c$ *with* c *a compile time constant;*
(3) p *is the closest predicate to* s *that satisfies the above two conditions along control flow.*

The branch of p *leading to* s*'s execution is called the SCD branch.*

Condition (1) dictates that s be directly or transitively control dependent on p. Condition (2) specifies that the branch outcome leading to s's execution determines that the left-hand side and the right-hand side expressions of the predicate have only a compile time constant difference. Such a condition reflects the strong correlations between the variable defined by s (let it be x) and e_1 and e_2. Intuitively, if an arbitrarily small ϵ[1] is applied to e_1, the expression equivalence must not hold, s must not get executed according to the condition, and hence x must have a different value. The same correlation is present between e_2 and x. Such strong correlations are very similar to those enabled by data dependences, e.g. in $x = e_1 + e_2$, because ϵ applied to either e_1 or e_2 will be reflected on x. The SCD defined above is also called the ϵ-SCD. We will show later that in many cases the constant c in the expression equation is 0, but SCD may occur even when c is some non-zero compile time constant.

Consider the example in Fig. 3. According to the definition, we have $17 \xrightarrow{scd} 16$. It is trivial to see condition (1) is satisfied. From the fact that 17 is executed, we can infer that `match` $\equiv 24$ and hence condition (2) is satisfied. Condition (3) is also true and hence there is a SCD. Observe that applying an ϵ to `match` will lead to 17 not being executed. If we change the predicate at line 16 to `if (match>24)`, the true branch is no longer a SCD branch and there is no SCD between 17 and 16 because condition (2) is not satisifed. From the fact that 17 is executed, we can only infer `match>24`, there are many valuations of `match` that satisfy the constraint. The compile time constant that ensures equivalence does not exist.

An important source of SCD is equivalence comparison such as line 16 in Fig. 3. More particularly, the true branch of an equivalence predicate (==) or the false branch of a non-equivalence predicate (!=) are SCD branches. Such predicates can be easily identified at compile time through static analysis. Later, we will show SCDs can originate from other comparative predicates.

Rule	Event	Instrumentation
(1)	Pred. $p : e_1 == e_2$ takes branch T	stack.push(p^T)
(2)	Pred. $p : e_1 != e_2$ takes branch F	stack.push(p^F)
(3)	Statement s	while (s is the immediate post-dominator of stack.top()) stack.pop()

Figure 4: Computation Rules for SCD Caused by Equivalence Testing.

[1]Expressions e_1 and e_2 could be any types allowing comparison; ϵ must always be 1 if they are integers.

4. RUNTIME DETECTION OF SCD

In this paper, we consider applications such as lineage tracing and dynamic information flow tracking. Hence, we are interested in detecting dynamic occurrences of SCDs through instrumentation. Particularly, *if* s *is SCD on* p*, an execution instance of* s *causes a dynamic SCD between the* s *instance and the* p *instance that guards the execution of the* s *instance.* In the remainder of the paper, the term strict control dependence also means dynamic strict control dependence, depending on the context.

Because a SCD branch may be nested in another SCD branch, we use a stack at runtime to handle such nesting as presented in [23, 11]. If a branch is decided to be a SCD branch at compile time, at runtime, the instrumentation pushes the predicate onto the stack (Rules (1) and (2) in Fig. 4). The stack entry is only popped when the immediate post-dominator of the entry is encountered (Rule (3)), because a branch is delimited by the predicate and its immediate post-dominator. In the instrumentation of Rule (3), a loop is used to pop multiple consecutive predicates on the stack that have the same immediate post-dominator. That is, the same immediate post-dominator serves as the end of multiple branches. This could occur in the presence of nested conditional statements or loops. Any statement execution is strictly control dependent on the top entry on the stack. Nesting is captured by the inner branch being placed on top of the outer branch in the stack. Note that other non-SCD predicate executions, such as the false branch of an equivalence predicate, do not lead to any stack operations.

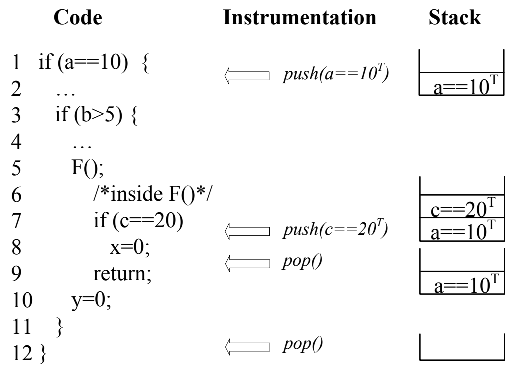
```
 1  if (a==10)  {
 2     …
 3     if (b>5) {
 4        …
 5        F();
 6           /*inside F()*/
 7           if (c==20)
 8              x=0;
 9           return;
10        y=0;
11     }
12  }
```

Figure 5: Example for Rules in Fig. 4.

Consider an example in Fig. 5. The code snippet is presented on the left. During execution, the instrumentation according to the rules in Fig. 4 is presented in the middle. The stack status is shown on the right. Observe that predicates 1 and 7 are pushed onto stack according to Rule (1). They are popped before the executions of 12 and 9, resp. No instrumentation is added for the predicate at 3 as the true branch is not a SCD branch. The execution of line 8 is strictly dependent on predicate at 7, which is in turn strictly dependent on the predicate at 1. Line 10 is strictly dependent on predicate at 1 even though it is syntactically enclosed by the true branch of 3.

Fig. 6 presents the rule for lineage computation in the presence of SCD. In particular, a statement execution is canonicalized to the form shown in Rule (L1), which is an assignment to a righthand side variable d after computation

Rule	Event	Instrumentation
(L1)	d=$f(s_1, s_2, ..., s_n)$	LN(d)=LN(stack.top()) ∪ LN(s_1) ∪ ... ∪ LN(s_n)

Figure 6: Rules for Lineage Tracing. LN(x) represents the lineage of variable x.

on a set of lefthand side variables s_1, ..., s_n. Note that a predicate can be considered as an assignment to a phantom boolean variable if needed. The meaning of the rule is that all the inputs involved in the computation of any lefthand side variable become part of d's lineage, together with the lineage of the closest SCD branch that guards the assignment. In Fig. 5, assume a, b, and c are inputs. Upon executing line 7, LN(c==20^T)=LN(a==10^T) ∪ {c}={a, c}. The rule for dynamic information flow tracking can be similarly defined and is omitted for brevity.

5. IMPLICIT SCD

It is known that information can be propagated through channels caused by execution omission [28]. Such dependences are neither data dependences nor control dependences. They are implicit, and hence called implicit dependences. In particular, such dependences arise if the definition of a variable is present in one branch (let it be the true branch without losing generality) but absent from the other (let it be the false branch). If at runtime, the false branch is taken, there is a correlation between the variable and the predicate even though the definition of the variable is omitted in the false branch. It is the omission that causes the correlation.

Consider the example in Fig. 7 (a). Variables x and y are initialized to 0. If the predicate at 2 takes the true branch, x is incremented. Otherwise, y is incremented. Assume the false branch is taken, we observe that there is dependence between the value of x and the predicate outcome. In other words, from $x = 0$, we can infer the false branch must have been taken. Observe that such dependence is not caused by data dependence or control dependence, but rather the omission of the definition to x in the false branch.

Handling implicit dependences in general is hard as predicates may guard large program regions. For instance, the two branches of a predicate may contain different complex control flows and call different sequences of functions. The set of variables that have their definitions omitted from one of the branches is hard to identify. Or even if they can be identified, there may be so many of them, giving rise to a large volume of implicit dependences. In [28], an offline algorithm was proposed to reveal the implicit dependence in a demand-driven fashion for the purpose of debugging. To test the presence of an implicit dependence between a variable and a predicate, the technique switches the branch outcome of the predicate during execution and observes if the variable is defined in the other branch. If so, the implicit dependence is true. One re-execution is needed to decide an implicit dependence. While such a solution is good for debugging, in which a large number of re-executions can be tolerated, it is not suitable for applications such as input lineage tracing or dynamic information flow that requires on-the-fly computation.

Properly handling implicit dependences is highly desirable for SCD computation, which is particularly true for loop related SCDs. Consider the example in Fig. 3. As mentioned

```
1  x=y=0;
2  if (a!=10) {
3    x++;
4  } else {
5    y++;
6  }
7  z=f(x);
```
(a) Original

```
1  x=y=0;
2  if (a!=10) {
3    x++;
4  } else {
5    y++;
     x=x;
6  }
7  z=f(x);
```
(b) Transformed

Figure 7: Example for Implicit SCD. Assume a is input.

earlier, ideally, we want to capture the strict dependence at line 14 between variable `match` and the input variable `args[0][pos]` when the loop exits. However, this cannot be achieved without handling implicit dependences. Without losing generality, we assume `args[0][pos]`='b'. Hence, the loop iterates two times and exits upon the third instance of the loop predicate, because `bdinfo[2]=='b'`. The execution is equivalent to executing the following code snippet.

```
    if (args[0][pos]!=bdinfo[match]) { // 'b'!='q'
       match++;
       if (args[0][pos]!=bdinfo[match]) { // 'b'!='r'
          match++;
L:        if (args[0][pos]!=bdinfo[match]) { // 'b'=='b'
             match++;
          }
       }
    }
```

The predicate instance at L takes the false branch and gives rise to SCD. However, the SCD branch has an empty body. In other words, the fact that `match` does not get updated induces the strict dependence between `match` and the predicate at L: from `match` ≡ 2, we can infer `args`[0][`pos`] ≡ `'b'`.

Fortunately, implicit dependences become easier to be handled in the context of SCD. The reason is that most predicate branches are not SCD branches and implicit dependences caused by those branches do not reveal strong correlations and hence don't need to be captured. For example, if the true branch of the predicate at line 2 in Fig. 7 (a) is taken, there is an implicit dependence between the value of y and the predicate. However, since it is not a SCD branch, the correlation between y and the predicate is not strong. In particular, applying an ϵ to a (the left-hand side of the predicate) or to 10 (the right-hand side) may not change the branch outcome and hence the value of y remains the same.

We transform the program such that implicit dependences inside SCD branches can be turned into regular explicit SCDs. The idea is similar to those presented in [2, 11]: for a variable that is defined in the branch opposite to the SCD branch but not in the SCD branch, a dummy identity assignment to the variable is inserted to the SCD branch. Since the identity assignment does not change the value of the variable, it allows the SCD being explicitly captured without changing program semantics. The transformation rules are presented in Fig. 8. Statement $\mathtt{S}_1$ represents the statement(s) in the true branch and $\mathtt{S}_2$ the statement(s) in the false branch. The highlighted statements are inserted by the transformation. Function `mustD(S)` gives the set of variables that must be defined in the (composite) statement `S`. If the SCD branch is the true branch (Rule T1), for each variable v_x that must be defined in the false branch but not

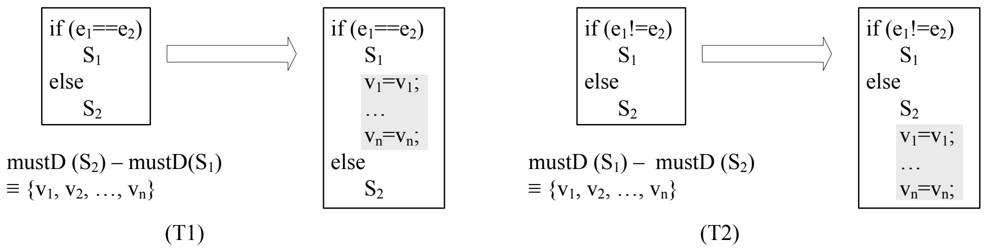

Figure 8: Handling Implicit SCD.

in the true branch, an identity assignment is inserted at the end of the true branch. In applications such as taint analysis or lineage tracing, the explicit SCDs between the dummy assignments and the predicate allow the taint bit (lineage set) of the predicate being propagated to variables v_1, v_2, ..., and v_n, and then the following program statements that make use of these variables. The missing links are hence remedied. The rule for the false branch (T2) is similarly defined.

In Fig. 7 (a), `mustD(3)-mustD(5)`={`x`}-{`y`}={`x`}. The identity assignment "`x=x`" is inserted after 5 in Fig. 7 (b). Assume we are computing lineage, according to Rule (L1) in Fig. 6, LN(`x`) = LN(`a!=10`^{F}) ∪ LN(`x`) = {`a`} upon executing the inserted statement. Later, when `x` is used at line 7, LN(`z`)=LN(`x`)={`a`}. The effect of both SCD and implicit dependence is faithfully captured.

For the loop at line 14 in Fig. 3, according to the transformation rule, an identity assignment to `match` is added to the false branch of the loop predicate, which is essentially the loop exit.

It is worth noting that the function `mustD` is computed through static analysis. It is defined in a way similar to busy expression analysis. In particular, a variable is present in `mustD(S)` if and only if it is defined along all paths in the sub control flow graph of `S`. Details are elided. Furthermore, observe that the inserted dummy assignments are not contributing to the state of the original program but rather to the analysis piggyback on the execution. They can be eliminated after the phase of application specific instrumentation, e.g. instrumentation for lineage tracing.

6. SCD BEYOND EQUIVALENCE TESTING

So far we have been discussing SCD caused by equivalence testing (including '==' and '!=' operators). We observe other comparison operators such as '<', '>', '<=', and '>=' can nonetheless induce SCD. According to Definition 1, one of the conditions is that the equivalence of the left-hand side and the right-hand side expressions of the predicate can be inferred. While this cannot be directly induced when only these operators are assumed, it can be induced when considered jointly with other predicates along the path to the predicate. In other words, the equivalence can be inferred from the path conditions. Consider the example in Fig. 9 (a). When line 4 is executed, the path 1, 2 and 3 must have been taken. The conditions are `a0<10`, `a1=a0-1` and `a1>7`, in which `a0` and `a1` represent the values of variable `a` at lines 1 and 3, resp. From these conditions, `a1`≡ `7+1`. Hence, line 4 is strictly control dependent on line 3 according to Definition 1.

A more common pattern of such SCDs occurs in loops. In Fig. 9 (b), we can observe that upon the exit of the loop, there is a strong connection between the value of `sum` and

```
1  if (a<10)  {
2    a--;
3    if (a>7) {
4       b=0;
5    }
```
(a)

```
11  for (i=0; i<end; i++) {
12      sum=sum+A[i];
13  }
```
(b)

Figure 9: SCD Not Caused by Equivalence Testing.

the value of `end`, although the predicate does not have an equivalence testing. Note that the execution corresponding to the last two instances of the loop predicate is equivalent to the following.

```
     /* loop starts*/
     ......
11a  if (i<end) {
12      sum=sum+A[i];
        i++;
11b     if (i<end)
        else { /* i>=end */
           sum=sum;
           i=i;
           goto L;
        }
     }
L:   /* loop exit*/
```

Identity assignments are inserted into the else branch of 11b according to the previously described transformation. Let the values of `i` in lines 11a and 11b be `i0` and `i1`, resp. From the path conditions, we have the conjunction `i0<end` ∧ `i1=i0+1` ∧ `i1>=end`. We can easily infer that `i1==end` when the loop exits. According to the definition, there are SCDs from the identity assignments to the predicate at 11b. Hence, the relation between `sum` and `end` is faithfully captured. Note the loop pattern in Fig. 9 (b) is very common.

Since the operators are no longer equivalence comparison, it is not trivial to statically identify the SCD branches such that proper instrumentation can be added. We propose a static analysis to identify such SCD branches. In particular, given a branch outcome (true or false) of a predicate $e_1 \diamond e_2$, we identify the predicate branches that it is directly or transitively control dependent on. Note that it means that such branches have to be taken in order to reach the given predicate. The conjunction of the path conditions denote the *must* condition of executing the given predicate branch. If variables involved in the path conditions are updated, such as "`a--`" in Fig. 9 (a) or "`i++`" in (b), the corresponding constraints also need to be conjoined with the path conditions. If the $e_1 \equiv e_2 + c$ can be inferred from the conjunction, the given branch is a SCD branch.

For a loop predicate p, the algorithm conducts similar reasoning by unrolling the last n instances of the loop predicate with the last instance taking the false branch and others the true branch. The algorithm gradually strengthens the conjunction by increasing n from 2. If equivalence can be inferred, the last instance denotes a SCD branch. It is worth mentioning that if the equivalence is inferred when n equals to a constant t, at runtime, the loop predicate has to execute t times to ensure the last instance is a SCD branch. For example, we prove the equivalence for the case in Fig. 9 (b) with $n = 2$. Hence, the loop has to iterate once (and hence the predicate has to execute twice) for the SCD to be true. If the first instance of loop predicate takes the false branch and the loop is not entered, it implies `end<=0`, the strong connection between `end` and `sum` is not present because `sum` has the same value for many possible values of `end` satisfying `end<=0`. Theoretically, the instrumentation needs to test the loop count to make sure it is larger than t. In practice, our instrumentation only needs to test if the difference between the two expressions in the loop predicate is equivalent to the derived constant c. For the case in Fig. 9 (b), we add a guard "`i==end`" in the false branch of the loop predicate to ensure it is a SCD branch. Note that in many cases equivalence cannot be inferred. For example, assume the loop "`for (i=a;i<100;i+=3)`". Upon loop exit, it may be true that `i==100`, `i==100+1` or `i==100+2`. Hence, there is not a compile time constant `c` such that `i==100+c` holds. We find that most loops fall into a small number of patterns. We leverage these patterns to speed up the analysis. For instance, if the loop induction variable is updated continuously, e.g. `i++`, the false branch of the loop predicate, regardless the type of the comparison, can always give rise to SCD.

7. EVALUATION

Our technique is implemented on *GCC-4.4.0*. The implementation consists of the SCD analysis and a lineage tracing system that is used to evaluate the effectiveness of SCD. The static analysis and transformations are done on the GIMPLE IR. Compared to other gcc IRs, GIMPLE is relatively simple and is language independent and machine independent. Our analysis is implemented as a compiler pass that is right after the generation of the GIMPLE IR and before any optimization passes. The reason is that we can leverage the later optimizations to optimize the instrumentation together with the original code. We use SPECINT 2000 as our benchmark for both effectiveness and efficiency. All experiments are run on an Intel Dual Core 2.5GHz machine with 2GB memory. The OS is Linux-2.6.30.

Table 1: Static program characteristics.

	LOC	# SCD	# branches	%
164.gzip	8616	263	1780	14.78%
175.vpr	17729	630	3848	16.37%
176.gcc	222182	10942	54318	20.14%
181.mcf	2423	31	338	9.17%
186.crafty	21150	814	5528	14.73%
197.parser	11391	455	3296	13.80%
253.perlbmk	85076	2665	22338	11.93%
254.gap	71430	1782	20220	8.81%
255.vortex	67220	2355	18680	12.61%
256.bzip2	4649	123	1134	10.85%
300.twolf	19748	1040	7462	13.94%

Table 1 presents the results of static analysis, including the number of branches in the program (column `# branches`) and the number of SCD branches (`# SCD`). We can see 8.8-20.1% of the branches are SCD branches. `176.gcc` has 20.1% SCD branches. It seems to be caused by the large number of `switch-case` statements.

7.1 Effectiveness

The next set of experiments is designed to evaluate the impact of SCD on runtime applications, including lineage tracing and taint analysis.

7.1.1 Lineage Tracing

We compare the lineage sets of individual output values in three settings: considering only data dependences (`DD` columns in Table 2); considering both data dependences and SCDs (`DD+SCD` columns); considering both data dependences and all control dependences (`DD+FCD` columns). We present the average lineage set sizes (`Avg` columns) and the maximum sizes (`Max` columns). Three benchmarks `175.vpr`, `255 vortex`, and `300.twolf` run out of memory and hence omitted from the result table. We can observe that DD lineage sets are usually very small and FCD lineage sets are large and the SCD lineage sets are in the middle.

Measuring False Positives (FP) and False Negatives (FN). Lineage size does not sufficiently reflect the quality of the resulting sets. Hence, we design a metric based on input fuzzing. In particular, we decide the correlation between each pair of input i and output o as follows. We identify the value range of i based on its type and input description. We mutate i to each possible different value while retaining the values of all the other inputs. If for each mutation, o either changes or is absent from the output, we say i is in the *ideal lineage set* of o, denoted as $i \in \mathrm{LI}(o)$, because it indicates that the correlation between i and o is strong. In other words, if multiple values of i lead to the same o value, i is not in $\mathrm{LI}(o)$. Note that o may be absent if the mutation alters the control flow of the execution. In order to determine the absence of a particular output and carry out proper output value comparison, we have to align the outputs of the mutated execution and the original execution. Output values are dumped to a text file, together with the program points that emit them. The longest common substring (LCS) algorithm is used to align two text files. Program locations are used to provide better identification because it may occur that even though some outputs are absent due to the mutation, similar outputs are emitted by a different program location. A purely value-based LCS algorithm may be confused and draw a wrong conclusion that the outputs are still present.

Let $\mathrm{LN}(o)$ be the computed lineage. FP and FN are computed as follows.

$$\mathrm{FP} = \frac{\mathrm{LN} - \mathrm{LI}}{\mathrm{LN}} \qquad \mathrm{FN} = \frac{\mathrm{LI} - \mathrm{LN}}{\mathrm{LI}}$$

The FP and FN results for different configurations are presented in Table 2. Overall, we observe that DD lineage has very high FNs, meaning it misses many important correlations. But it almost always has 0 FPs. In contrast, FCD lineage has very high FPs, but almost 0 FNs, meaning it captures all important correlations but also many weak correlations. SCD lineage provides more accurate results in general: it has both low FPs (maximum 19% and average 7.81%) and low FNs (maximum 25% and average 7.38%). The following explains some of the prominent data entries.

Table 2: Output Lineage Size and FP/FN Ratio.

	DD				DD + SCD				DD + FCD			
	Max	Avg	FP	FN	Max	Avg	FP	FN	Max	Avg	FP	FN
164.gzip	496	438.14	0.00%	10.98%	496	491.20	0.00%	0.23%	497	493.11	0.19%	0.00%
176.gcc	14	0.43	0.00%	99.36%	92	57.32	8.17%	13.26%	108	67.74	23.91%	0.03%
181.mcf	3	2.48	9.26%	7.41%	3	2.67	14.81%	0.00%	43	39.70	87.41%	0.00%
186.crafty	3	0.07	0.00%	95.99%	32	24.72	10.99%	25.29%	60	56.38	39.87%	0.00%
197.parser	0	0	0.00%	100.00%	5	3.06	0.00%	0.00%	138	123.19	97.71%	0.00%
253.perlbmk	2	0.8	0.00%	96.45%	57	33.80	19.14%	7.46%	95	66.80	66.29%	0.00%
254.gap	1	0.59	0.00%	100.00%	54	18.41	9.32%	12.82%	90	36.55	42.66%	0.00%
256.bzip2	211	10.62	0.00%	90.58%	211	197.79	0.00%	0.00%	211	197.79	0.00%	0.00%
Aveage	91.25	56.64	1.16%	75.10%	118.75	103.62	7.81%	7.38%	155.25	135.16	44.75%	0.00%

- The lineage sizes of different methods for `164.gzip` and `256.bzip2` are not much different and they are all large. The reason is that the two compression programs rely on an accumulative process of finding patterns and hence are very sensitive to any individual input values. In fact, making any change to any single input value changes almost all individual outputs.
- It is counter-intuitive that some FPs are observed for the DD lineage in `181.mcf`. Further inspection shows that they are mainly caused by the following line of code in `readmin.c`.

```
59  net->arcs = (arc_t *)calloc( net->max_m, sizeof(arc_t) );
```

  There is data dependence from `net` $\rightarrow$ `max_m` to `net` $\rightarrow$ `arcs` through which the lineage information is propagated. However, the link is weak as although mutating inputs may change the size of allocation, the allocated memory with the new size does not affect the output as long as the new size is sufficiently large.
- FPs are observed in SCD lineage. We inspect the case for `186.crafty`. We find the FPs are mainly caused by the following code snippet.

```
In lookup.c:

 63  if (!Xor(And(htable->word2,mask_80)
         temp_hash_key))
 65      transposition_hits++;
           .....
100  if (!Xor(And(htable->word2,mask_80),
         temp_hash_key))
102      transposition_hits++;

In iterate.c:

436  printf("hashing->trans/ref:%d%%...\n",
437    transposition_hits/(transposition_probes+1));
```

  Variable `transposition_hits` is not transitively data dependent on any input. The true branches of the predicates at lines 63 and 100 are SCD branches. Lineage is hence correctly propagated to the variable at lines 65 and 102. Line 437 emits execution statistics. However, input changes can be masked by the division at line 437 such that the corresponding inputs are not strongly correlated with the output in lines 436-437. Such FPs are not observed in DD lineage because the propagation of lineage is broken at lines 65 and 102.
- FNs are observed in both SCD and FCD lineage. We inspect the case for `176.gcc` and find that they are mainly caused by early program termination. Consider the following code snippet. Variable `finput` at line 2005 has `name` in its lineage set. If the true branch is taken, the program immediately terminates. Hence, all the following execution is determined by the predicate. In particular, any mutation on `name` leads to the failure to open the file and then early termination. Input `name` shall be in the ideal lineage of all outputs. However, `GCC` is not able to perform analysis across source files. Hence, the control dependences between 2005 and all execution beyond it are not correctly captured. We expect a whole program analysis would mitigate the problem.

```
In toplev.c:

2004    finput = fopen (name, "r");
2005    if (finput == 0)
2006      pfatal_with_name (name);
```

We want to mention that the results in Table 2 are sensitive to the definition of ideal lineage. Another possible definition is that given a pair of input i and ouput o, if there exists a mutation of i that ever changes the value of o, i should be in LI(o). This definition will admit inputs that are very loosely correlated to the given output. We will leave it to our future work to study other ways of measuring FPs/FNs.

Case Study. We have presented a case study in Section 2. Here we use another case study to further demonstrate the effectiveness of SCD. In this study, we want to show that after a compiler optimization, the output lineage can precisely include the statements in the source file that are relevant to the optimized code. We use the common sub-expression elimination (CSE) pass in `176.gcc`, in which the compiler goes through the code and identifies sub-expressions that must have the same value (i.e., none of the variables get killed in between two occurrences of the same sub-expression) and then replaces the sub-expression with a variable. We use the following input file.

```
     void foo () {
1      int a, b, c, d;
2      int p, q, r;

3      r = p ^ q;
4      c = a * b;
5      d = b * a;
     }
```

Function `cse_insn(insn)` is responsible for analyzing the expressions appearing in instruction `insn`. Part of the lineage before `cse_insn()` starts to process line 5 is the following. Symbol `'b'@5` means the char `'b'` at line 5.

```
6095 void cse_insn (rtx insn) {
6290   rtx dest = SET_DEST (...);
6291   rtx src = SET_SRC (...);

       /* Locate all possible equivalent forms for SRC. */
6427   hash = HASH (src, ...);
6448   elt = lookup (src, hash, ...);

       /* Replace the expression with the cheaper equivalent. */
6838   validate_change (insn, &SET_SRC (...),
                        elt->first_same_value, ...)
     }

1174 struct table_elt * lookup (rtx x, unsigned hash, ...) {
1181   for (p = table[hash]; p; p = p->next_same_hash)
1182     if (...  || exp_equiv_p (x, p->exp, ...) == 1)
1183       return p;

       return 0; /* not found */
     }

2051 static int exp_equiv_p (rtx x, rtx y, ...) {
2062   if (x == y && ...)
         return 1;
2067   code = GET_CODE (x);
2095   switch (code) {
        ...
       /* For commutative operations, check both orders. */
2136   case PLUS:  case MULT:  case AND:
2143     return ((exp_equiv_p (XEXP (x, 0), XEXP (y, 0), ...)
                 && exp_equiv_p (XEXP (x, 1), XEXP (y, 1), ...))
               || (exp_equiv_p (XEXP (x, 0), XEXP (y, 1), ...)
                 && exp_equiv_p (XEXP (x, 1), XEXP (y, 0), ...));
       }
     }
```

Figure 10: Code snippet for CSE in 176.gcc.

```
LN('b'@5) = {'int'@1, 'b'@1, 'b'@5, '*'@5}
LN('a'@5) = {'int'@1, 'a'@1, 'a'@5, '*'@5}
LN('*'@5) = {'int'@1, 'a'@1, 'a'@5,
             'b'@1, 'b'@5, '*'@5}
```

The highly simplified code snippet relevant to CSE is shown in Fig. 10. When function `cse_insn(insn)` starts to process line 5, variables `src` at line 6291 holds the expression "`b * a`", which is hashed into variable `hash` at line 6427. The hash value is used in calling `lookup()` at line 6448 to look for the equivalent expressions. In function `lookup()`, the hash table is traversed to find the candidate expression by calling function `exp_equiv_p()` at line 1182. In function `exp_equiv_p()`, the incoming expressions `x` and `y` are checked for their equivalence recursively. In this case, `x` and `y` are "`b * a`" and "`a * b`" respectively. These two expressions are further broken down to check the equivalence for each of their operands.

Since the return value of `exp_equiv_p()` is an integer, guarded by the result of comparisons at lines 2062 and 2095, whose true branches are SCD branches. When SCDs are captured, the lineage set of the resulting integer is the union of those of `x` and `y`, reflecting their relevance. In other words, the lineage sets from the original expression and the one in the hash table are unioned. The unioned lineage set is further propagated to variable `elt` at line 6448 through the SCD at line 1182. Hence, after the CSE optimization, the compiled file and its corresponding lineage set look like the following.

```
4     c = a * b;
5'    d = c;
```

```
LN('c'@5') = {'int'@1, 'a'@1, 'a'@4, 'a'@5,
              'b'@1, 'b'@4, 'b'@5, '*'@4, '*'@5}
```

Observe that the lineage of line 5' includes symbols of '`a`' and '`b`' in both lines 4 and 5 in the original code, indicating the optimization is indeed the result of finding the common sub-expression in lines 4 and 5. If SCDs are not considered, the lineage contains only line 5 but not line 4. Moreover, if all control dependences are considered, the resulting lineage of `LN('c'@5')` contains all expressions, including the one at line 3. The reason is that the lineage of an expression in the hash table contains all the expressions that are encountered before it due to the (loose) control dependences occurred in hash table operations during parsing. The relevant code snippet is omitted due to space limitation.

7.1.2 Taint analysis

We use taint analysis as another application to demonstrate the effectiveness of SCD. Note that taint analysis can be easily achieved with minor tweaking of the lineage system. In Fig. 11, four programs from SPECINT2000 are randomly selected. The x axis represents the amount of inputs that we taint. The y axis represents the corresponding amount of outputs that get tainted at the end of program execution. We present four sets of results: considering only data dependences (DD curve in Fig. 11); considering both data dependences and SCDs (DD+SCD curve); considering both data dependences and all control dependences (DD+FCD curve). We also plot an ideal curve. The curve is acquired through fuzzing. Given a tainted input, if any mutation on the input must cause mutation on an output or make the output disappear, the output should be tainted ideally. Observe that the DD+FCD curve ascends much faster than the DD curve, meaning considering all control dependences often leads to over-tainting. The ideal and DD+SCD curves are in the middle and the DD+SCD curve closely follows the ideal curve except the middle part of `186.crafty`, representing SCD can result in high quality tainting.

7.2 Efficiency

We evaluate the impact of tracking control dependences. The overheads of tracking strict control dependences (SCD) and all control dependences (FCD) are collected. For FCD detection, we implement the algorithm in [11, 23]. We normalize the results according to the uninstrumented execution time. As shown in Fig. 12, the average overheads of SCD and FCD are 76% and 294% respectively. It causes more than two times slowdown to record FCD than recording SCD. These results are before any aggressive algorithmic optimizations. The compiler option was -O3.

8. RELATED WORK

Control dependence computation. Ferrante et al. [6] studied the use of dependence graphs in compiler optimizations. They proposed the concept of *Program Dependence Graph* that combines both data and control dependences in one graph. They also demonstrated how certain compiler optimizations can be done more efficiently on the graph. Horwitz et al. [8] presented 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*

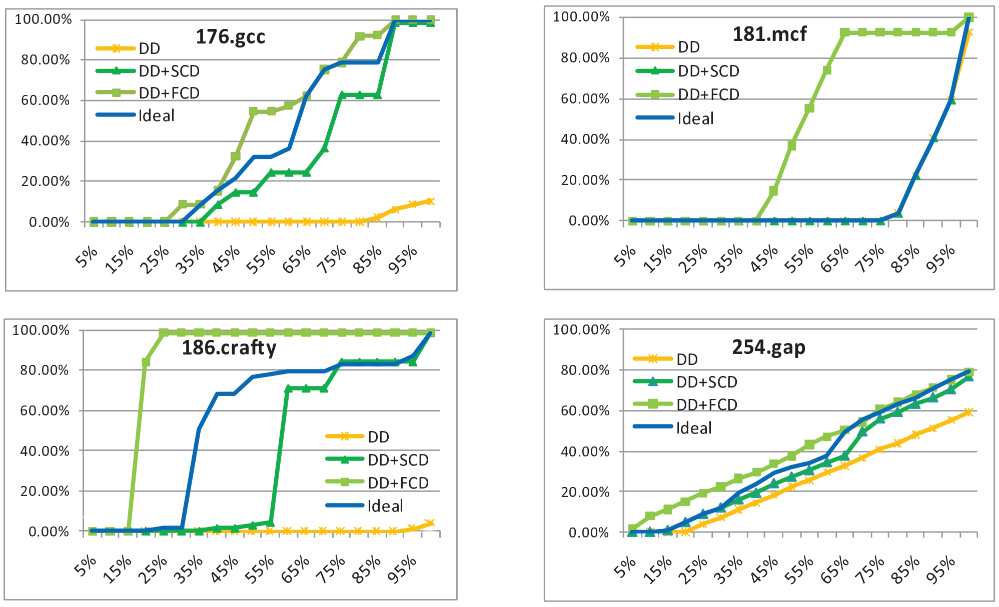

Figure 11: Taint Analysis Result.

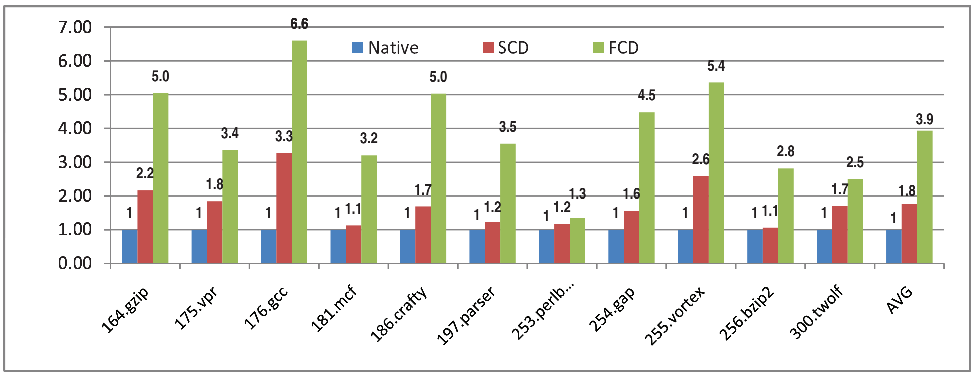

Figure 12: Overhead of Tracking Control Dependences.

dependence edges, which in turn are computed by a technique borrowed from Attribute Grammar. Later, a more thorough algorithm to compute interprocedural control dependence was proposed in [18]. Podgurski and Clarke have proposed the term *weak control dependence* in [16] to describe dependences between loop termination and the statements following the loop. To distinguish with the classic control dependence, they called the classic control dependence *strong control dependence*. In [1], a general framework was proposed by Bilardi and Pingali to perform faster computation of both the classic and weak control dependences.

Compared to these static techniques, our definition of *strict control dependences* (SCD) is different, and it represents a subset of control dependences that manifests the characteristics of data dependence and hence we argue it should be equally considered in applications that consider data dependences.

SCD is relevant to confidence analysis in dynamic slicing [27]. Confidence analysis is an offline analysis that allows pruning unimportant dependence edges to produce smaller slices. It requires constructing dependence graph and then traverses the graph in a selective backward fashion with confidence information. SCD and confidence analysis share similar spirit in deciding the importance of dependences. The difference lies in that SCD branches are statically decided without profiling. SCDs are computed online and in a forward fashion, and hence they are desirable in applications such as taint analysis and lineage tracing. Furthermore, we handle implicit dependences caused by code omission through code transformation.

Taint analysis. Taint analysis[24, 14, 2, 10] assigns a taint to inputs (usually those received from untrusted sources) and then monitors the propagation of the taints during program execution. Lineage tracing [26] can be considered as a generalized form of taint analysis that captures the set of relevant inputs contributing to an output. These techniques and most other tainting systems mainly focus on data dependence and do not consider control dependence. As shown by our results, our technique would be a good complement. Clause and Orso [2, 3] provide a more generalized infrastructure supporting dependences along both control-flow and data-flow. But they do not consider the strength of control dependence. Most recently, Slowinska and Bos [19] evaluate the practicality of taint analysis. They conclude that taint analysis might induce both high false positive(FP) and high false negative(FN), and even taint explosion depending on the architecture, operation system and the nature of the subject program. According to their study, not considering control dependence is one important source of FNs. Our experiments also largely support their observation.

Information flow tracking. Information flow tracking systems [5, 4, 7, 12, 13, 15, 17, 25] prevent confidential information from leaking. Most of these systems rely on program dependence analysis. Podgurski and Clarke[16] showed that if one statement is being data or control dependent on by other statements, it could affect their execution. Many systems are not accurate enough to catch implicit information flow due to the limited support for control dependence. To achieve better result, in [11], Masri et al. considered control dependence in dynamic information flow analysis. SCD would help addressing the induced FPs and FNs.

9. CONCLUSION

We propose a new concept called strict control dependence (SCD). SCD belongs to traditional control dependence but its characteristics are more like data dependences, which indicate strong correlation between statements. We formally define the semantics of SCD. We find that any predicates that involve comparative operators may give rise to SCDs. We devise a static analysis to locate these predicates. We also introduce the instrumentation rules detecting runtime instances of SCD. Implicit dependences are handled by program transformation. We evaluate the effectiveness of SCD on lineage tracing and taint analysis. We find that SCD substantially improves the quality of computed lineage sets and provides better tainting. We also find that online detection of SCDs incurs 76% overhead.

10. ACKNOWLEDGEMENT

We would like to thank the reviewers for their substantial efforts. This research is supported, in part, by the National Science Foundation (NSF) under grants 0916873 and 0847900. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

11. REFERENCES

[1] G. Bilardi and K. Pingali. A framework for generalized control dependence. In *PLDI'96*.

[2] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA'07*.

[3] J. Clause and A. Orso. Penumbra: Automatically identiying failure-relevant inputs using dynamic tainting. In *ISSTA'09*.

[4] D. Denning and P. Denning. Certification of programs for secure information flow. In *Communications of ACM*, 1977.

[5] D. E. Denning. A lattice model of secure information flow. In *Communications of the ACM*, 1976.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *TOPLAS*, 1987.

[7] N. Heintze and J. G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL'98*.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI'88*.

[9] B. Korel and J. Laski. Dynamic program slicing. In *Information Processing Letters*, 1988.

[10] L.C. Lam and T.Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC'06*.

[11] W. Masri, A. Podgurski, and Leon D. Detecting and debugging insecure information flows. In *ISSRE'04*.

[12] A. C. Myers. Jflow: Practical mostly-static information flow control. In *POPL'99*.

[13] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. In *TOSEM*, 2000.

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS'05*.

[15] 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'04*.

[16] A. Podgurski and L. A Clarke. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. In *TSE*, 1990.

[17] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO'06*.

[18] S. Sinha, M.J. Harrold, and G. Rothermel. Interprocedural control dependence. In *TOSEM'01*.

[19] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *EuroSys'09*.

[20] G.E. Suh, J.W. Lee, D. Zhang, and S.Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS'04*.

[21] M. Weiser. *Program Slicing: Formal , Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor , Michigan, 1979.

[22] Mark Weiser. Program slicing. In *ICSE'81*.

[23] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA'07*.

[24] W. Xu, S. Bhatkar, and R.Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, 2006.

[25] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Capturing system-wide information flow for malware detection and analysis. In *CCS'07*.

[26] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *VLDB'07*.

[27] X. Zhang, R. Gupta. Pruning dynamic slices with confidence. In *PLDI'06*.

[28] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI'07*.