Relevant Inputs Analysis and its Applications

Yan Wang Rajiv Gupta Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside, CA, USA

***Abstract*—In this paper we develop a dynamic analysis, named *relevant input analysis*, that characterizes the *role* and *strength* of inputs in the computation of different values during a program execution. The *role* indicates whether a computed value is derived from an input value or its computation is simply influenced by an input value. The *strength* indicates if role (derived or influenced) relied upon the precise value of the input or it is among one of many values that can play a similar role. While it is clear that the results of our analysis can be very useful for the programmer in understanding relationships between inputs and program behavior, we also demonstrate the usefulness of the analysis by developing an efficient delta debugging algorithm. Other applications of relevant input analysis include—assisting in generating test inputs and detection of security holes.**

***Index Terms*—value dependence, address dependence, role of inputs, strength of inputs, delta debugging, testing**

I. Introduction

Understanding the behavior of a program during a specific execution is critical to many important tasks including: debugging, testing, and security analysis. A variety of dynamic analysis techniques have been developed to assist with these tasks (e.g., dynamic slicing [5], [8], delta debugging [10], [11], [13], generating test inputs [14], [15], [16], [17], information flow [18], [19]). In these tasks, understanding the role that input values play during execution can be very useful in understanding program behavior. During debugging, when a program crashes, knowing the part of the input that triggered the crash can help the programmer. During testing, understanding the role of inputs in one execution can help in generating new inputs that will likely exercise different program behaviors. Finally, understanding the role of inputs in exploiting security vulnerabilities helps identify security holes.

In this paper we develop a dynamic analysis, named *relevant input analysis*, that characterizes the *role* and *strength* of inputs in the computation of different values during a program execution. The *role* indicates whether a computed value is *derived* from an input value or its computation is simply *influenced* by an input value. The *strength* indicates if role relied upon the precise value of the input or it is among one of many values that can play a similar role.

In addition to understanding program behavior we demonstrate the benefits of relevant input analysis by using its results to enhance the *delta debugging* [10], [13] algorithm. The relevant input analysis is used to prune, as well as guide, and hence accelerate, the search for a minimal fault-inducing input. The results of our experiments show that our approach can significantly reduce the number of inputs on which the program is executed before the minimal input is located. We also briefly discuss how relevant input analysis can be used to assist in generating test inputs and detection of security holes.

The remainder of the paper is organized as follows. In Section II we develop our algorithm for relevant input analysis and evaluate its cost. In Section III we present our new delta debugging algorithm and its evaluation, as well as discuss other applications—test input generation and detection of vulnerabilities exploitable via malicious inputs. Section IV concludes the paper.

II. Relevant Input Analysis

A. Motivating example

Prior relevant input analyses such as lineage tracing [22], [23], [24] identify the subset of inputs that contribute to a specified output by considering data dependence only [22], both data dependence and control dependence [23], or both data dependence and strict control dependence [24]. However, they do not characterize the role and strength of inputs in the computation of different values during execution, as we do.

We now present an example to motivate the approach and illustrate the effectiveness of our relevant input analysis. The example is extracted from a real null pointer dereference bug in Tidy-34132, a program that parses, cleans and corrects ill-formed HTML documents. The relevant parts of the code are shown in Figure 1. In this simplified view, an HTML document contains a Header section (represented by 'H') and a frameset section (represented by 'S', lines 4–6). The frameset section holds one or more Frame elements (represented by 'F'), specifying the layout of views in the user agent window. In addition, the frameset section can contain a Noframes element (represented by 'N') to provide alternate content for browsers that do not support frames or have frames disabled (line 23). Noframes must contain a Body element (represented by 'B'). Framesets can be nested to any level. The body element can contain multiple Paragraphs (represented by 'P'). All the paragraphs should be included in the body element. When this property is violated, the program calls HandlePsOutsideBody (lines 51–56) to fix it. HandlePsOutsideBody simply discards those paragraphs when a body element has not been encountered. When the end of the body has been parsed, HandlePsOutsideBody moves such paragraphs into the body element by adding all paragraphs after the body as children of body (line 53–54). The function FindBody (line 71–84) retrieves the body node. There is a bug in FindBody: the wrong assumption that noframe is always included in the outermost frameset. So when the noframe and body are included in an

```
parser.c:
1  void ParseHtmlDoc() {
2    doc=malloc(sizeof(Doc));
3    doc−>seeEndBody=FALSE;
4    doc−>head=ParseHead();
5    doc−>fS=NULL;
6    ParseFrameSet(NULL); }
7  void ParseFrameSet(Node*p)
8  { Node *fS=NULL;
9    char c=GetChar(fin);
10   if(c=='S') {
11     fS=NewNode(fSTag);
12     if(p) AddChild(p,fS);
13     if(doc−>fS==NULL)
14       doc−>fS=fS;
15     c=PeekChar(fin);
16     while(c=='S'
17         ||c=='F') {
18       if(c=='S')
19         ParseFrameSet(fS);
20       else ParseFrame(fS);
21       c=PeekChar(fin);
22     }
23     ParseNoFrame(fS);
24     c=GetChar(fin);
25     if(c=='/') ... }
29 }
30 void ParseNoFrame(Node *fS)
31 { char c=GetChar(fin);
32   if(c=='N') {
33   Node *noF=NewNode(noFTag);
34   AddChild(fS,noF);
35   HandlePsOutsideBody();
36   ParseBody(noF);
37   HandlePsOutsideBody();
38   c=GetChar(fin);
39   if(c=='/') ... }
44 }
45 void ParseFrame(Node *fS)
46 { char c=GetChar(fin);
47   if(c=='F') {
48     Node *f=NewNode(fTag);
49     AddChild(fS,f); }
50 }
51 void HandlePsOutsideBody()
52 { if(doc−>seeEndBody==true)
53   { Node *body= FindBody();
54     ParseParagraphs(body); }
55   else ConsumeParagraphs();
56 }
57 void ParseBody(Node *noF)
58 { char c=GetChar(fin);
59   if(c=='B') {
60     Node *body=NewNode(bTag);
61     AddChild(noF,body);
62     ParseParagraphs(body);
63     c=GetChar(fin);
64     if(c=='/') {
65       c=GetChar(fin);
66       if(c=='B')
67     doc−>seeEndBody=true;
68       else Warn(...); }
69     else Warn(...); }
70   else Ungetc(c,fin); }
71 Node *FindBody()
72 { Node *node=doc−>fS;
73   if(node==NULL) return NULL;
74   node=node−>firstChild;
75   while(node &&
76         node−>type!=noFTag)
77     node=node−>sibling;
78   if(node) {
79     node=node−>firstChild;
80     while(node &&
81           node−>type!=bTag)
82       node=node−>sibling; }
83   return node;
84 }
85 void ParseParagraphs(Node *b)
86 { char c=GetChar(fin);
87   while(c=='P') { ...
90     ParseTextNode(p);
91     c=GetChar(fin);
92     if(c=='/'){
93       c=GetChar(fin);
94       if(c!='P') Warn(...); }
95     else Warn(...);
96     c=GetChar(fin); }
97   Ungetc(c,fin);
98 }
99  Node *NewNode(NodeType type)
100 { Node *node=malloc(...);
      ...
104   node−>sibling=NULL;
105   return node; }
106 void AddChild(Node*p,Node*c)
107 { if(p>lastChild!=NULL)
108     p−>lastChild−>sibling=c;
109   else p−>firstChild=c;
110   p−>lastChild=c;
111 }
112 void ParseTextNode(Node*p)
113 { char c=GetChar(fin);
114   if(c=='"') {
115   c=GetChar(fin);
      ...
118   c=GetChar(fin);
119   if(c!='") Warn(...); }
120   else Ungetc(c,fin); }
121 char GetChar(Stream *fp) {
122   if(fp−>r_ptr>=fp−>r_end)
123   return RefillBuf(fp);
124   return *(fp−>r_ptr++); }
```

Fig. 1. Buggy code for illustrating relevant input analysis.

inner frameset, FindBody will wrongly return a NULL pointer, which causes a program crash at line 107 (p is NULL here).

Given a failure-inducing input, shown at the top of Figure 2), the program crashes at the eighth execution of line 107, denoted as 107_8. Consider the computation of the relevant input for variable p at failure point 107_8. The results of relevant input analyses, both as computed by prior work [22], [23], [24], as well as our algorithm, are given in Figure 2. As the program crashes when parsing the third P in the input, all the unprocessed inputs ("b"/P/N/S/S) are successfully excluded by all approaches. Lineage computation [22] only considers data dependence, and it gives an empty lineage because no input propagates into p at 107_8 via data dependence edges only. Penumbra [23] can be configured to consider either data dependences only, or both data and control dependences. Thus, it can generate two relevant input sets: one is empty just as lineage computation [22] or a set that includes almost

Original Input:

$S\ S\ F\ F\ N\ B\ P\ "\ a\ "\ /\ P\ /\ B\ P\ "\ b\ "\ /\ P\ /\ N\ /\ S\ /\ S$

Inputs Labeled with Occurrence Frequency:
$S^1\ S^2\ F^1\ F^2\ N^1\ B^1\ P^1\ "^1\ a^1\ "^2\ /^1\ P^2\ /^2\ B^2\ P^3\ "^3\ b^1\ "^4\ /^3\ P^4\ /^4\ N^2\ /^5\ S^3\ /^6\ S^4$

Compute relevant input/lineage for failure Point–107_8(p is NULL):
Result of lineage [22]: {}

Result of Penumbra [23]:
$\{\} \mid \{S^1, S^2, F^1, F^2, N^1, B^1, P^1, "^1, /^1, /^2, B^2, P^3\}$

Result of lineage with strict control dependence[24]:
$\{S^1, S^2, F^1, F^2, N^1, B^1, P^1, "^1, /^1, /^2, B^2, P^3\}$

Result of our approach:
$\{S^{2=}\rightarrow$NULL(node $\rightarrow$sibling@104)$\} \wedge \{S^{1=}, S^{2=}, F^1, F^2, N^{1=}, B^{1=}, P^1, "^1, /^1, /^{2=}, B^{2=}\rightarrow$true(doc$\rightarrow$seeEndBody@67), $P^{3=}\}$
$\wedge \{S^1, S^2, F^1, F^2, N^1, B^1, P^1, "^1, /^1, /^2, B^2, P^3\}$

Fig. 2. Comparing prior work results with our relevant input analysis.

all the parsed inputs. Subsequent improvements of lineage computation [24] consider both data dependences and strict control dependences, and produce the same relevant input set as Penumbra when configured considering both data and control dependences. By examining the program execution, we discover that the reason why lineage computation with strict control dependence and Penumbra include nearly all the inputs is because of the data and control dependences involving the index of buffer ($fp \rightarrow r_ptr$) at line 124. It is a common programming practice to maintain a buffer to store the input data and then process the data in the buffer. The program in Figure 1 maintains such a buffer and parses inputs based on the buffer (GetChar, Ungetc, PeekChar operates on this buffer). Hence whenever an input is read (e.g., line 86 reads the third P used in the predicate at line 87 just before the crash point), it is data dependent on the last modification of the index of the input buffer ($fp \rightarrow r_ptr$ at line 124). Because this buffer index is increased after an input is read at line 124, and line 124 is (strict) control dependent on the predicates which guard the execution of GetChar, i.e., GetChar at line 118 is (strict) control dependent on line 114, and GetChar at line 91 is (strict) control dependent on line 87, such data and control dependence chains explain why nearly all the processed inputs are included in the relevant input. Naturally, such broad and imprecise information will not be very useful in practice.

Our relevant input analysis is based upon two observations. First, we observe that data dependences incurred by operand (later defined as value dependence) should be treated differently from data dependence incurred by index or pointer (later defined as address dependence) which is used to select the operand (i.e., different dependences/inputs have different roles). Second, we observe that each dependence/input has different strength regarding the concerned output value. Our relevant input analysis characterizes the *role* and *strength* that dependence/inputs play in the computation of different values during a program execution. Going back to the example in Figure 1, the result of our relevant input analysis is shown at the bottom of Figure 2. As we can see, instead of one,

we present three sets: the first set includes only inputs which the concerned value p is derived from (inputs contribute to p only through value dependence); the second set includes inputs which influence p through control dependence and value dependence; and the third set includes inputs which influence p through address, control, and value dependence. Inputs labeled with $=$ in the three sets have a strong impact on the value of p. Specifically, in order to trigger or understand this bug, two conditions must be satisfied: (1) the NULL value must be generated somewhere; (2) the program execution must reach a point where this NULL value gets dereferenced. The $S^{2=}$→NULL(node→sibling@104) in our first set exactly shows that the NULL value of p is propagated from $(node \rightarrow sibling)$ at line 104, and this NULL value again is generated because of the second frameset ($S^{2=}$). The $S^{1=}$, $S^{2=}$, $N^{1=}$, $B^{1=}$, $/^{2=}$, $B^{2=}$→true(doc→seeEndBody@67), $P^{3=}$ in the second set shows that in order to cause the execution to reach this failure point, we must exactly have such inputs: $SSNB/BP$ (which turns out to be the minimal input to trigger the same bug). As we can see, our relevant input analysis provides valuable information for aiding program comprehension, debugging, test case generation, etc.

B. Definitions

Our relevant input analysis tracks dynamic dependences, originating from points where the program reads the inputs, and categorizes them to distinguish the ways in which they impact the computation of values. Given the ith execution of statement s (denoted as s_i), we use VAL(sto_i) to denote the value computed at s_i, and during this computation, m variables are used (denoted as sfr_1, sfr_2, sfr_k, ..., sfr_m). The predicate on which s_i is control-dependent, is denoted as $pred_j$. Statement s itself can also be a predicate, in which case, VAL(sto_i) denotes the evaluated result of this predicate (TRUE/FALSE). We now describe the three categories.

- *Value Dependence* – VAL(sto_i)$\overset{v}{\leftarrow}$VAL(sfr_k): VAL(sto_i) is *value dependent* upon VAL(sfr_k) if the latter is used as an operand for computing the former;
- *Address Dependence* – VAL(sto_i)$\overset{a}{\leftarrow}$VAL(sfr_k): VAL(sto_i) is *address dependent* upon VAL(sfr_k) if the latter is used to select the address whose contents are used as an operand for computing the former. These dependences arise due to the presence of pointers and arrays; and
- *Control Dependence* – VAL(sto_i)$\overset{c}{\leftarrow}$VAL($pred_j$): sto_i is *dynamically control dependent* [3], [7] upon $pred_j$, i.e., VAL($pred_j$) causes the execution of sto_i.

C. Role of Relevant Inputs

We treat all the external inputs to a program (e.g., file, stdin, network) as concerned inputs. For simplicity, we model the input as a string, and the newly-arriving inputs are simply appended to this string. The relevant inputs for a value VAL computed in a program execution that reads a set of inputs INPUTS are represented as follows:

$$\text{VAL} \leftarrow \text{DERIVED} \wedge \text{CINFLUENCED} \wedge \text{AINFLUENCED}$$

- Value VAL is *derived from* inputs belonging to DERIVED $\subseteq$ INPUTS if there is a chain of value dependences from each input in DERIVED to VAL:
$$\{r \mid r \in \text{INPUTS} \wedge \exists\ \text{VAL} \overset{v}{\leftarrow} \ldots \overset{v}{\leftarrow} \text{READ(r)}\}$$
- Value VAL is *control influenced by* inputs belonging to CINFLUENCED $\subseteq$ INPUTS if there is a chain of value and/or control dependences from each input in CINFLUENCED to VAL such that at least one control dependence is present in the chain:
$$\{r \mid r \in \text{INPUTS} \wedge \exists\ \text{VAL} \overset{v/c}{\leftarrow} \ldots \overset{v/c}{\leftarrow} \text{READ(r)}\}$$
- Value VAL is *address influenced by* inputs belonging to AINFLUENCED $\subseteq$ INPUTS if there is a chain of value, and/or control, and/or address dependences from each input in AINFLUENCED to VAL such that at least one address dependence is present in the chain:
$$\{r \mid r \in \text{INPUTS} \wedge \exists\ \text{VAL} \overset{v/c/a}{\leftarrow} \ldots \overset{v/c/a}{\leftarrow} \text{READ(r)}\}$$

We illustrate the aforementioned relevant input notions with an example in Figure 3 (note that we do not consider the strength of inputs for now). The code fragment on the left contains two loops. The first loop reads a sequence of numbers into the `data[]` array. The input consists of a sequence of positive and negative integers which is terminated by the value 0. The second loop scans the array and computes the sum of positive numbers (`posSum`) and sum of negative numbers (`negSum`). Finally the values of `posSum` and `negSum` are printed out. In the right column the execution trace and relevant inputs of computed values is presented for the input sequence {3, -15, 0}. The results of our analysis show that the DERIVED sets of `posSum` and `negSum` are found to be {3} and {-15} due to chains of value dependence. The CINFLUENCED set for `posSum` is {3} due to control/value dependence chain $22_1 \overset{c}{\leftarrow} 19_1 \overset{v}{\leftarrow} 18_1 \overset{v}{\leftarrow} 12_1 \overset{v}{\leftarrow} 11_1(READ(3))$. AINFLUENCED set for `posSum` is empty because no relevant inputs are propagated along address/control/value dependence chain. The CINFLUENCED set for `negSum` is {3,-15} due to chains of control/value dependences along which the values 3 and -15 are tested by predicates eventually causing the execution of statement 24_1. Note that AINFLUENCED set for `negSum` is {3} because of such address/control/value chain:

$$24_1 \overset{v}{\leftarrow} 18_2 \overset{a}{\leftarrow} 25_1 \overset{c}{\leftarrow} 19_1 \overset{v}{\leftarrow} 18_1 \overset{v}{\leftarrow} 12_1 \overset{v}{\leftarrow} 11_1(READ(3))$$

D. Strength of Relevant Inputs

Next we show that we can further qualify the inputs by determining their strength in computing other values. In particular, we determine if the computed values rely upon the *precise value* of an input, or the input value is among *one of many* values that can cause similar behavior. For this purpose a specific input value r will appear in the DERIVED, CINFLUENCED, or AINFLUENCED sets as $r^{=}$ (to indicate that computed value depends upon the precise value of r) or simply r (to indicate that potentially other values will lead to similar behavior as r). We now present the situations in which

Code	Execution Trace	DERIVED ∧ CINFLUENCED ∧ AINFLUENCED
	2_1 posSum=0;	VAL(posSum) ← {} ∧ {} ∧ {}
	3_1 negSum=0;	VAL(negSum) ← {} ∧ {} ∧ {}
	6_1 num=0;	VAL(num) ← {} ∧ {} ∧ {}
	7_1 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
1 int data[100];	9_1 if(num>=100)	VAL(num>=100) ← {} ∧ {} ∧ {}
2 int posSum=0;	11_1 fscanf(fin,"%d",&dt); // 3	VAL(dt) ← {3} ∧ {} ∧ {}
3 int negSum=0;	12_1 data[num]=dt;	VAL(data[num]) ← {3} ∧ {} ∧ {}
4 int dt;	13_1 num++;	VAL(num) ← {} ∧ {} ∧ {}
5 int i;	7_2 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
6 int num=0;	9_2 if(num>=100)	VAL(num>=100) ← {} ∧ {} ∧ {}
7 while(!feof(fin))	11_2 fscanf(fin,"%d",&dt); // -15	VAL(dt) ← {-15} ∧ {} ∧ {}
8 {	12_2 data[num]=dt;	VAL(data[num]) ← {-15} ∧ {} ∧ {}
9 if(num>=100)	13_2 num++;	VAL(num) ← {} ∧ {} ∧ {}
10 break;	7_3 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
11 fscanf(fin, "%d", &dt);	9_3 if(num>=100)	VAL(num>=100) ← {} ∧ {} ∧ {}
12 data[num]=dt;	11_3 fscanf(fin,"%d",&dt);// 0	VAL(dt) ← {0} ∧ {} ∧ {}
13 num++;	12_3 data[num]=dt;	VAL(data[num]) ← {0} ∧ {} ∧ {}
	13_3 num++;	VAL(num) ← {} ∧ {} ∧ {}
14 }	7_4 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
15 i=0;	15_1 i=0;	VAL(i) ← {} ∧ {} ∧ {}
16 while(i<num)	16_1 while(i<num)	VAL(i<num) ← {} ∧ {} ∧ {}
17 {	18_1 dt=data[i]; // 3	VAL(dt) ← {3} ∧ {} ∧ {}
18 dt=data[i];	19_1 if(dt==0) // end marker	VAL(dt==0) ← {3} ∧ {} ∧ {}
19 if(dt==0) *//end marker*	21_1 if(dt>0)	VAL(dt>0) ← {3} ∧ {3} ∧ {}
20 break;	22_1 posSum+=dt;	VAL(posSum) ← {3} ∧ {3} ∧ {}
21 if(dt>0)	25_1 i++;	VAL(i) ← {} ∧ {3} ∧ {}
22 posSum+=dt;	16_2 while(i<num)	VAL(i<num) ← {} ∧ {3} ∧ {}
23 else	18_2 dt=data[i]; // -15	VAL(dt) ← {-15} ∧ {3} ∧ {3}
24 negSum+=dt;	19_2 if(dt==0) //end marker	VAL(dt==0) ← {-15} ∧ {3} ∧ {3}
25 i++;	21_2 if(dt>0)	VAL(dt>0) ← {-15} ∧ {3,-15} ∧ {3}
26 }	24_1 negSum+=dt;	VAL(negSum) ← {-15} ∧ {3,-15} ∧ {3}
27 printf ("%d", posSum);	25_2 i++;	VAL(i) ← {} ∧ {3,-15} ∧ {3}
	16_3 while(i<num)	VAL(i<num) ← {} ∧ {3,-15} ∧ {3}
28 printf ("%d", negSum);	18_3 dt=data[i]; // 0	VAL(dt) ← {0} ∧ {3,-15} ∧ {3,-15}
	19_3 if(dt==0) //end marker	VAL(dt==0) ← {0} ∧ {3,-15} ∧ {3,-15}
	20_1 break;	
	27_1 printf("%d", posSum);	VAL(posSum) ← {3} ∧ {3} ∧ {}
	28_1 printf("%d", negSum);	VAL(negSum) ← {-15} ∧ {3,-15} ∧ {3}

Fig. 3. Example illustrating the role of input values.

the above attributes can be associated when dynamic *value dependences*, *control dependences*, and *address dependences* are encountered.

Value Dependence: When the DERIVED set of a computed value VAL contains an input value $r^{=}$ it means that to keep VAL unchanged, we need the exact value of r (VAL is highly likely to be changed if the input value r is changed); otherwise DERIVED simply contains r (VAL may change if we change the input value r). The example below illustrates the propagation of value 10 input by the read statement. When the value 10 is first read into x and later copied to another variable y (strong value dependence), the corresponding DERIVED sets contain $10^{=}$ (strong value dependence maintains the strength of inputs). However, when the value of z is computed from the value of x at line 3 (weak value dependence), z's DERIVED set contains 10 (weak value dependence weakens the strength of inputs). Besides, because 10 has already been weakened at line 3, when the value of z is later copied to w, w contains 10 instead of $10^{=}$ (strong value dependence only maintains the strength of inputs). Similarly, when x is used in the predicate at line 5 (or 6, respectively) and it tests whether x is equal (not equal, respectively) to a precise input value 10 and when the predicate outcome is true (false, respectively), the DERIVED set will contain $10^{=}$ (strong value dependence maintains the strength of inputs); otherwise, DERIVED will simply contain 10 (line 7).

1: read x;	VAL(x) ← $\{10^{=}\}$ ∧ {} ∧ {}
2: y = x;	VAL(y) ← $\{10^{=}\}$ ∧ {} ∧ {}
3: z = f(x);	VAL(z) ← {10} ∧ {} ∧ {}
4: w = z;	VAL(w) ← {10} ∧ {} ∧ {}
5: if(x==10) true	VAL(x==10) ← $\{10^{=}\}$ ∧ {} ∧ {}
6: if(x!=10) false	VAL(x!=10) ← $\{10^{=}\}$ ∧ {} ∧ {}
7: if(x > 0)	VAL(x > 0) ← {10} ∧ {} ∧ {}

Control Dependence: If a predicate tests whether the value of a variable is equal (not equal, respectively) to a precise input value r, then the CINFLUENCED set of a statement that is control dependent upon the true (false, respectively) outcome of the predicate will contain $r^{=}$; otherwise CINFLUENCED will simply contain r. The example below illustrates the propagation of value $0^{=}$ input by the read statement and thus contained in DERIVED set of x. The value $0^{=}$ is propagated to the CINFLUENCED sets of values of w and y via control dependences.

1: read x;	VAL(x) ← $\{0^{=}\}$ ∧ {} ∧ {}
2: z = x;	VAL(z) ← $\{0^{=}\}$ ∧ {} ∧ {}
3: if (x==0)	VAL(x==0) ← $\{0^{=}\}$ ∧ {} ∧ {}
4: w = z;	VAL(w) ← $\{0^{=}\}$ ∧ $\{0^{=}\}$ ∧ {}
5: y = 1;	VAL(y) ← $\{0^{=} \rightarrow 1(y@5)\}$ ∧ $\{0^{=}\}$∧{}
6: if(y < 100)	VAL(y < 100) ← {0} ∧ {0}∧{}

Consider a predicate that tests if an input value is precisely equal to constant c_1, and if the predicate evaluates to true, it sets another variable to a constant value c_2. Such a computation essentially maps the value of c_1 to the value c_2, i.e., c_2 is derived from c_1. Therefore in this situation we also propagate input value c_1 from the DERIVED set of a predicate to the DERIVED set of a control dependent statement that assigns c_2. The propagation also captures the mapping by including $c_1 \rightarrow c_2$ in the DERIVED set. In the above example, $0^=$ is propagated from DERIVED set of predicate (x==0) to the DERIVED set of y's value by inclusion of $0^= \rightarrow$1(y@5). Note that in such chains all values are exact values. Note that if y is later used in line 6 (weak value dependence), DERIVED and CINFLUENCED sets include 0 instead of $0^=$ (weak value dependence weakens the strength of inputs).

Address Dependence: If the value of a variable v used to select the address whose content (e.g., $*v$) is used as operand exactly relies on some input r, then the AINFLUENCED set of computed value VAL contain $r^=$ (changing the value of r will highly likely change the value of v and then $*v$); otherwise AINFLUENCED will simply contain r (changing value of r may change the value of v and then $*v$). The example below illustrates the propagation of value 10 input by the read statement. When the value 10 is first read into x and later used to select the address, the computed value z's AINFLUENCED set contains $10^=$. On the other hand, when a value of y is computed from the value of x and then used to select address, the computed value w's AINFLUENCED set contains 10. When z is tested in predicate $if(z > 0)$, the AINFLUENCED set for this predicate contains 10, rather than $10^=$.

1: read x;	VAL(x) $\leftarrow \{10^=\} \wedge \{\} \wedge \{\}$
2: z = buf[x];	VAL(z) $\leftarrow \{50^=\} \wedge \{\} \wedge \{10^=\}$
3: y = f(x);	VAL(y) $\leftarrow \{10\} \wedge \{\} \wedge \{\}$
4: w = buf[y];	VAL(w) $\leftarrow \{40^=\} \wedge \{\} \wedge \{10\}$
5: if(z > 0)	VAL(z > 0) $\leftarrow \{50\} \wedge \{\} \wedge \{10\}$

E. Computation of Relevant Inputs

The dynamic value analysis is performed by instrumenting the program such that for each instruction that is executed, the relevant input sets of the computed value are found according to the dynamic dependences of the executed instruction. Figure 4 shows how the DERIVED (DER), CINFLUENCED (CINF) and AINFLUENCED (AINF) sets are computed via propagation of relevant input information along all dynamic dependences (Value, Address, and Control). The $\uplus$ operation used in the figure is a slight modification of traditional union. When two values derived from same input are encountered, the stronger condition is retained:

$$\{c^=\} \uplus \{c\} = \{c^=\}$$

Similarly, when two chains are encountered such that one is a prefix of another, then the longer chain is retained as it represents a stronger condition.

$$\{c^= \rightarrow d(var@s)\} \uplus \{c^=\} = \{c^= \rightarrow d(var@s)\}$$

The $S[c^= \ldots /c]$ operation used in Figure 4 is used to drop the $=$ label (i.e., weaken the strength of inputs), and it is defined as follows:

$$S[c^= \ldots /c] = \{c \mid c \in S \vee c^= \ldots \in S\}$$

For example,

$$\{c_1^=, c_2^= \rightarrow d(var@s), c_3\}[c^= \ldots /c] = \{c_1, c_2, c_3\}$$

```
Initialize: DER(sto_i) ← CINF(sto_i) ← AINF(sto_i) ← φ;
Compute DER(sto_i) ∧ CINF(sto_i) as follows:
for each prior statement execution on which
 VAL(sto_i) is directly dependent do
  – Value Dependence
  case VAL(sto_i) ←v VAL(sfr_k):
    case sto_i : ... = sfr_k:
    case sto_i : if (sfr_k == c_1) TRUE:
    case sto_i : if (sfr_k != c_1) FALSE:
    DER(sto_i) ← DER(sto_i) ⊎ DER(sfr_k)
    CINF(sto_i) ← CINF(sto_i) ⊎ CINF(sfr_k)
    AINF(sto_i) ← AINF(sto_i) ⊎ AINF(sfr_k)
    otherwise:
    DER(sto_i) ← DER(sto_i) ⊎ DER(sfr_k)[c= .../c]
    CINF(sto_i) ← CINF(sto_i) ⊎ CINF(sfr_k)[c= .../c]
    AINF(sto_i) ← AINF(sto_i) ⊎ AINF(sfr_k)[c= .../c]
  – Address Dependence
  case VAL(sto_i) ←a VAL(sfr_k):
    case sto_i : ... = *sfr_k:
    case sto_i : *sfr_k = ...:
    case sto_i : if (*sfr_k == c_1) TRUE:
    case sto_i : if (*sfr_k != c_1) FALSE:
    AINF(sto_i) ← AINF(sto_i) ⊎ DER(sfr_k)
      ⊎ CINF(sfr_j) ⊎ AINF(sfr_k)
    otherwise:
    AINF(sto_i) ← AINF(sto_i) ⊎ DER(sfr_k)[c= .../c]
      ⊎ CINF(sfr_k)[c= .../c] ⊎ AINF(sfr_k)[c= .../c]
  – Control Dependence
  case VAL(sto_i) ←c VAL(pred_j):
    case sto_i : ... = sfr_k:
    case sto_i : if (...):
    CINF(sto_i) ← CINF(sto_i) ⊎ DER(pred_j) ⊎ CINF(pred_j)
    otherwise:
    CINF(sto_i) ← CINF(sto_i) ⊎ DER(pred_j)[c= .../c]
      ⊎ CINF(pred_j)[c= .../c]
    DER(sto_i) ← DER(sto_i) ⊎ CHAIN, such that
    case sto_i : sto_i = c_2 is TRUE dependent
      on pred_j : if(var == c_1):
    case sto_i : sto_i = c_2 is FALSE dependent
      on pred_j : if(var! = c_1):
      CHAIN={c_1= ... → c_2(sto_i@s)|c_1= ... ∈ DER(pred_j)}
    otherwise: CHAIN = φ
endfor
```

Fig. 4. Dynamically Computing Relevant Inputs of VAL(sto_i).

For example, In Figure 5 we present the results of the above analysis when it is applied to a code segment that parses a string and if the string is "body," then seeBody is set to true. The input in this case is contained in name[] and we assume that it is indeed the string "body" terminated by "\0". The first loop in the code fragment compares the input string with "body" which is stored in str. If there is an exact match, we exit the loop after setting cmp to 0. A chain of mappings $\backslash 0^= \rightarrow$0(cmp@12)$\rightarrow$BODY(tag@27)$\rightarrow$true(seeBody@29) finally leads us to statement return seeBody (line 30).

Both DERIVED and CINFLUENCED sets of seeBody at statement 30_1 capture very useful information. The chain $\backslash 0^= \rightarrow$0(cmp@12)$\rightarrow$BODY(tag@27)$\rightarrow$true(seeBody@29) in DERIVED set indicates how \0 is mapped to 0 for cmp first, and then eventually to true for seeBody. The CINFLUENCED indicates that the exact characters in "body" must be encountered as the set contains $b^=$, $o^=$, $d^=$, $y^=$, and $\backslash 0^=$. As

Code	Execution Trace	DERIVED ∧ CINFLUENCED ∧ AINFLUENCED
	1_1 Parse(node)	// node→name="body\0"
	3_1 seeBody=false;	VAL(3_1)←{} ∧ {} ∧ {}
	4_1 str="body";	VAL(4_1)←{} ∧ {} ∧ {}
1 Parse(**char**∗name)	6_1 i=0;	VAL(6_1)←{} ∧ {} ∧ {}
2{	7_1 c=name[i];//'b'	VAL(7_1)←{$b^{=}$} ∧ {} ∧ {}
3 seeBody=false;	8_1 **while**(c==str[i])	VAL(8_1)←{$b^{=}$} ∧ {} ∧ {}
4 str="body";	10_1 **if** (c == '\0')	VAL(10_1)←{b} ∧ {$b^{=}$} ∧ {}
5 **int** cmp;	15_1 i++;	VAL(15_1)←{} ∧ {b} ∧ {}
6 **int** i=0;	16_1 c=name[i];//'o'	VAL(16_1)←{$o^{=}$} ∧ {$b^{=}$} ∧ {b}
7 c=name[i];	8_2 **while**(c==str[i])	VAL(8_2)←{$o^{=}$} ∧ {$b^{=}$} ∧ {b}
8 **while**(c==str[i])	10_2 **if** (c == '\0')	VAL(10_2)←{o} ∧{$b^{=}$,$o^{=}$} ∧ {b}
9 {	15_2 i++;	VAL(15_2)←{} ∧ {b,o} ∧ {b}
10 **if** (c == '\0')	16_2 c=name[i];//'d'	VAL(16_2)←{$d^{=}$} ∧ {$b^{=}$,$o^{=}$} ∧ {b,o}
11 {	8_3 **while**(c==str[i])	VAL(8_3)←{$d^{=}$} ∧ {$b^{=}$,$o^{=}$} ∧ {b,o}
12 cmp = 0;	10_3 **if** (c == '\0')	VAL(10_3)← {d} ∧ {$b^{=}$,$o^{=}$,$d^{=}$} ∧ {b,o}
13 **break**;	15_3 i++;	VAL(15_3)←{} ∧ {b,o,d} ∧ {b,o}
14 }	16_3 c=name[i];//'y'	VAL(16_3)←{$y^{=}$} ∧ {$b^{=}$,$o^{=}$,$d^{=}$} ∧ {b,o,d}
15 i++;	8_4 **while**(c==str[i])	VAL(8_4)←{$y^{=}$} ∧ {$b^{=}$,$o^{=}$,$d^{=}$} ∧ {b,o,d}
16 c=name[i];	10_4 **if** (c == '\0')	VAL(10_4) ← {y} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$} ∧ {b,o,d}
17 }	15_4 i++;	VAL(15_4)←{} ∧ {b,o,d,y} ∧ {b,o,d}
18 **if** (c!=str[i])	16_4 c=name[i];//\0	VAL(16_4)←{$\backslash 0^{=}$} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$} ∧ {b,o,d,y}
19 {	8_5 **while**(c==str[i])	VAL(8_5)←{$\backslash 0^{=}$} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$} ∧ {b,o,d,y}
20 **if** (c>str[i])	10_5 **if** (c == '\0')	VAL(10_5)←{$\backslash 0^{=}$} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y }
21 cmp= 1;	12_1 cmp= 0;	VAL(12_1)←{$\backslash 0^{=}$→0(cmp@12)} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y}
22 **else**	18_1 **if** (c!= str[i])	VAL(18_1)←{$\backslash 0^{=}$} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$} ∧ {b,o,d,y}
23 cmp=−1;	25_1 tag=OTHER;	VAL(25_1)←{} ∧ {} ∧ {}
24 }	26_1 **if** (cmp==0)	VAL(26_1)←{$\backslash 0^{=}$→0(cmp@12)} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y}
25 TAG tag=OTHER;	27_1 tag=BODY;	VAL(27_1)←{$\backslash 0^{=}$→0(cmp@12)→BODY(tag@27)} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y}
26 **if** (cmp==0)	28_1 **if** (tag==BODY)	VAL(28_1)←{$\backslash 0^{=}$→0(cmp@12)→BODY(tag@27)} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y}
27 tag=BODY;	29_1 seeBody=true;	VAL(29_1)←{$\backslash 0^{=}$→0(cmp@12)→BODY(tag@27)→true(seeBody@29)} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y}
28 **if** (tag==BODY)	30_1 ret seeBody;	VAL(30_1)←{$\backslash 0^{=}$→0(cmp@12)→BODY(tag@27)→true(seeBody@29)} ∧ {$b^{=}$,$o^{=}$,$d^{=}$,$y^{=}$,$\backslash 0^{=}$} ∧ {b,o,d,y}
29 seeBody=true;		
30 ret seeBody;		
31}		

Fig. 5. Body parse example.

we can see, such information will be very useful for program comprehension and fault localization.

F. Implementation

We have implemented the relevant input analysis using the Pin [21] dynamic instrumentation framework. As shown in Figure 4, we need to update DERIVED, CINFLUENCED and AINFLUENCED sets for each written value based on the relevant input sets of used values and the control-dependent predicate. To get more accurate control dependence, we adopted the online dynamic control dependence detection algorithm in [7]. To speed up the look-up of relevant input sets for each dependent value, we bound each computed value with its relevant input sets by shadow memory. To save space and allow efficient set operations, we stored all distinct computed relevant input sets in a balanced binary tree, and then only stored the pointer to each set in shadow memory. The CHAIN was implemented similarly to save time and space.

G. Performance Evaluation

Next we use several real programs (described in Section III-A) to investigate whether the time overhead of our technique is acceptable. The experiments were conducted on a machine with a 3.0GHz Intel Xeon processor and 3GB RAM, running Linux, kernel version 2.6.18. We use the "Null Pin" (the program running under Pin without our debugger) time overhead as the baseline, which is shown in the second column, and the time overhead with relevant input analysis on is given in the third column. From Table II, we can see that the time overhead incurred by our technique ranges from 31.7x to 39.2x compared to the baseline, which is reasonable compared to related work [22], [7].

TABLE I
OVERVIEW OF BENCHMARKS.

Program	LOC	Bug Source	Program description
tidy-34132	35.9K	BugNet [20]	HTML checking & cleanup
bc-1.06	10.7K	BugNet [20]	Arbitrary-precision Calculator
expat-1.95.3	11.9K	sourceforge.net/p/expat/bugs	XML parser

TABLE II
PROGRAM EXECUTION TIMES (FROM START TO FAILURE POINT), WITH RELEVANT INPUT ANALYSIS.

Program name	Null Pin seconds	Relevant Input Analysis seconds (factor)
tidy-34132	1.08	37.4(34.6x)
bc-1.06	0.73	28.6(39.2x)
expat-1.95.3	0.48	15.2(31.7x)

H. Related Work

While many different forms of value flow analysis have been developed before, their purpose is quite different from ours. Static analysis techniques to reason about values with the goal of optimizing code have been developed [1], [2]. While static analysis techniques are aimed at identifying properties that hold during all executions of a program, our work is aimed at analyzing the flow of input values during a particular execution. Dynamic analysis is much more suitable for applications such as debugging [11], test input generation [14], [17], and security [4]. As mentioned earlier, the most closely-related effort to our relevant input analysis is lineage tracing [22], [23], [24]. However, none of previous approaches differentiate the

role and *strength* inputs play in computing a specified value. Our relevant input analysis characterizes the *role* and *strength* of inputs play in the computation of different values during a program execution. Dynamic information flow analysis (e.g., taint analysis) has been used in security applications [18], [19], [4]. The information we collect is much richer and hence enables a wider range of applications (e.g., delta debugging) as illustrated in this paper. While dynamic slicing [5], [8] has been widely studied to assist in debugging, the relevant input analysis presented in this paper differs from dynamic slicing in two important ways: dynamic slicing treats both value and address dependences as data dependences while we distinguish between them; and our analysis yields much richer information that characterizes role and strength.

III. Applications of Relevant Input Analysis

A. Delta Debugging

From the results of the preceding section it is clear that relevant input analysis can help in understanding program behavior. Therefore, in this section we show that the results of analysis can be used to develop an enhanced *delta debugging* [10], [11] algorithm. Given a program input on which the execution of a program fails, delta debugging automatically simplifies the input such that the resulting simplified input causes the same failure. In particular, it finds a 1-minimal input, i.e., an input from which removal of any entity causes the failure to disappear. This is achieved by carrying out a search in which: new simpler inputs are generated; the program is executed to determine if same failure is caused by the simpler input; and the above steps are repeatedly applied until the input cannot be simplified any further.

We now present a new delta debugging algorithm, called IDTHDD (Input Decomposition Tree-based Hierarchical Delta Debugging), that uses relevant input analysis to accelerate the search for the 1-minimal input. This is achieved as follows:

- **Step 1: Removal of Irrelevant Inputs**. The input is simplified by removing entities that do not appear in the relevant input set of the wrong value identifying the failure (e.g., wrong output or reference causing a crash).
- **Step 2: Construct Input Decomposition Tree**. From the dynamic dependence chop, that includes all dependence chains from input entities to faulty value, we derive a tree that represents a hierarchical decomposition of the entire input into subsets of input entities.
- **Step 3: Search for 1-Minimal Input**. The decomposition tree enables a pruned search (relative to the default delta debugging) for finding a 1-minimal input.

Next we will present the three steps in detail and illustrate our algorithm on the program in Figure 1. To better illustrate our algorithm, we use a longer failing input which has 59 entities; after removal of irrelevant inputs failing input size is 10, and finally the 1-minimal input size is 7.

Step 1: Remove Irrelevant Inputs: On a failing run, the failure is revealed either because the program crashes or it generates a wrong output. In either case, at some point in execution, a wrong value is produced and detected. Since the goal of input simplification is to reproduce the same failure, we can reduce the original input by removing all irrelevant input entities, i.e., those entities that do not appear in the relevant input set of the wrong value. We further try to reduce the input size by generating multiple inputs of different sizes from the relevant input set of the wrong value. In particular, we generate the following inputs and select the first input that reproduces the same failure. The $DER^{=}$, $CINF^{=}$ and $AINF^{=}$ sets include only those subsets of input entities from DER, $CINF$, and $AINF$ that are attributed with $^{=}$.

First Input:	$DER^{=}$
Second Input:	$DER^{=} \cup CINF^{=}$
Third Input:	$DER^{=} \cup CINF^{=} \cup AINF^{=}$
Fourth Input:	$DER \cup CINF^{=} \cup AINF^{=}$
Fifth Input:	$DER \cup CINF \cup AINF^{=}$
Sixth Input:	$DER \cup CINF \cup AINF$

In Figure 6 we show the impact of removing irrelevant inputs for our running example. The original, 59-entities input is reduced to a simple 10-entities failing input. Note that it is possible that none of the subsets can reproduce the original fault because the removal of irrelevant parts may result in a malformed input. In this case, our algorithm simply defaults to the standard delta debugging algorithm.

Step 2: Construct Input Decomposition Tree: Next, for the input obtained in the previous step, an *input decomposition tree* is constructed that hierarchically decomposes the input as follows. The root of the tree represents the wrong value computed in the failing run and its children represent a subset of other values computed during execution upon which the root value is dependent. Moreover, while the root is labeled with the entire input on which it is dependent, its children are labeled with *disjoint subsets* of inputs labeling the root node. The inputs labeling each child node of the root node are similarly further decomposed among their children and so on. Finally, each leaf node represents a read of an input value.

Thus, each level in the tree represents a decomposition of the input into disjoint subsets such that at the root node all inputs are in a single partition while at each subsequent level the inputs are decomposed into increasing number of disjoint subsets. During delta debugging, from the input decomposition at a given level in the tree, simpler inputs will be constructed by excluding or including each subset as a unit. This reduces the search space explored by delta debugging and thus accelerates the search for a 1-minimal input.

Figure 7 shows the input decomposition tree for our running example. As we can see, while the input associated with the root node is the entire input found in Step 1 (SFFFSNB/BP), each of the leaf nodes has a single input entity attached to it, which is the specific entity that was read by the leaf node. Internal nodes correspond to larger subsets of the input set. Note that although the leaf nodes at levels other than the last level are shown once, these nodes must be viewed as being repeated at later levels so that each level represents a decomposition of the entire input.

The input decomposition tree is derived from the dynamic dependence subgraph consisting of dynamic depen-

Original Input:

$H\ "\ t\ "\ /\ H\ S\ F\ F\ F\ S\ F\ F\ N\ P\ "\ a\ "\ /\ P\ P\ "\ b\ "\ /\ P\ B\ P\ "\ c\ "\ /\ P\ P\ "\ d\ "\ /\ P\ /\ B\ P\ "\ e\ "\ /\ P\ P\ "\ f\ "\ /\ P\ /\ N\ /\ S\ /\ S$

Inputs Labeled with Occurrence Frequency:

$H^1\ "^1\ t^1\ "^2\ /^1\ H^2\ S^1\ F^1\ F^2\ F^3\ S^2\ F^4\ F^5\ N^1\ P^1\ "^3 a^1 "^4 /^2 P^2\ P^3\ "^5\ b^1\ "^6\ /^3\ P^4\ B^1\ P^5\ "^7\ c^1\ "^8\ /^4\ P^6\ P^7\ "^9\ d^1\ "^{10}$
$/^5\ P^8\ /^6\ B^2\ P^9\ "^{11}\ e^1\ "^{12}\ /^7\ P^{10}\ P^{11}\ "^{13}\ f^1\ "^{14}\ /^8\ P^{12}\ /^9\ N^2\ /^{10}\ S^3\ /^{11}\ S^4$

Relevant Inputs for 107_{14} (Failure Point, p is NULL) :

VAL(107_{14}) $\leftarrow \{S^{2=} \rightarrow$NULL(node$\rightarrow$sibling@104)$\} \wedge \{H^1, "^1, /^1, S^{1=}, F^{1=}, F^{2=}, F^{3=}, S^{2=}, F^4, F^5,$
$N^{1=}, P^1, "^3, /^2, P^3, "^5, /^3, B^{1=}, P^5, "^7, /^4, P^7, "^9, /^5, /^{6=}, B^{2=} \rightarrow$true(doc$\rightarrow$seeEndBody@67), $P^{9=}\}$
$\wedge \{ H^1, "^1, /^1, S^1, F^1, F^2, F^3, S^2, F^4, F^5, N^1, P^1, "^3, /^2, P^3, "^5, /^3, B^1, P^5, "^7, /^4, P^7, "^9, /^5, /^6, B^2, P^9 \}$

Construct and Try Simpler Inputs:

First Input Constructed from: $DER^{=}$=$\{S^2\}$

$\longrightarrow S \longrightarrow$ original failure cannot be reproduced.

Second Input Constructed from: $DER^{=} \cup CINF^{=}$= $\{S^1, F^1, F^2, F^3, S^2, N^1, B^1, /^6, B^2, P^9\}$

$\longrightarrow S\ F\ F\ F\ S\ N\ B\ /\ B\ P \longrightarrow$ **original failure is reproduced !!**

Resulting Simpler Input following Step 1: $\longrightarrow$ $S\ F\ F\ F\ S\ N\ B\ /\ B\ P$

Fig. 6. Step 1: Removing irrelevant inputs.

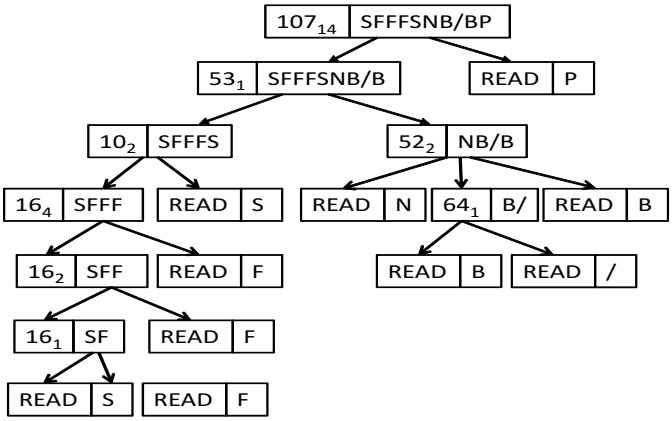

Fig. 7. Step 2: Generating the Input Decomposition Tree.

Level	Step	Test Case		Result	
1	1	∇_{2_1}	P	√	
	2	∇_{2_2}	S F F F S N B / B	√	Go to next level
2	3	∇_{2_1}	N B / B P	√	
	4	∇_{2_2}	S F F F S P	√	Go to next level
3	5	∇_{2_1}	N B / B P	√	
	6	∇_{2_2}	S F F F S P	√	Increase granularity
	7	∇_{5_1}	S N B / B P	√	
	8	∇_{5_2}	S F F F N B / B P	√	
	9	∇_{5_3}	S F F F S B / B P	√	
	10	∇_{5_4}	S F F F S N B P	√	
	11	∇_{5_5}	S F F F S N B / P	√	Go to next level
4	12	∇_{2_1}	S N B / B P	√	
	13	∇_{2_2}	S F F F S N B P	√	Increase granularity
	14	∇_{4_1}	F S N B / B P	√	
	15	∇_{4_2}	S F F S N B / B P	×	Reduce to complement
	16	∇_{3_1}	S N B / B P	√	
	17	∇_{3_2}	S F F S N / B P	√	
	18	∇_{3_3}	S F F S N B B P	√	Go to next level
5	19	∇_{2_1}	F S N B / B P	√	
	20	∇_{2_2}	S F S N B / B P	×	Reduce to complement
6	21	∇_{2_1}	F S N B / B P	√	
	22	∇_{2_2}	**S S N B / B P**	×	**Done - 1-minimal Input found !!**

Fig. 8. Step 3: Searching for 1-minimal Input – found $S\ S\ N\ B\ /\ B\ P$.

dence chains originating from the input entities and terminating at the faulty value, produced as follows:

- Construct a breadth first *spanning tree* starting from the faulty value as the root and continuing until all inputs the faulty value is dependent on (i.e., inputs identified in Step 1) have been included in the tree. Collapse chains such that no node in the tree has only a single child.
- Label each node with its *input subset* which is simply the set of inputs that are reachable from the node via the edges in the spanning tree. Note that for any given level in the spanning tree, each node at that level will be labeled by a disjoint input subset since the input sets are computed using the paths that exist in the spanning tree, i.e., no input is reachable from multiple nodes at the same level in the spanning tree.

Step 3: Search for 1-Minimal Input: Let us briefly discuss how the input decomposition tree is used to search for a 1-minimal input. We apply hierarchical delta debugging according to levels in the spanning tree. At each level when delta debugging is applied, each distinct input subset at that level is viewed as a single entity, i.e., it is either entirely included in or entirely excluded from a generated input. This is similar to the hierarchical delta debugging [13]; although the source of hierarchy is altogether different. When applying delta debugging to each level of the input decomposition tree, we only try each complementary set instead of first trying delta sets and then complementary sets. This is based on the observation that step 1 already successfully pruned large failure irrelevant chunks from the input.

Taking the spanning tree given in Figure 7 as input, the input simplification using delta debugging is illustrated in Figure 8. We consider the root as being level 0. Therefore the figure shows inputs derived from level 1 onward. At levels 1, 2, and 3 delta debugging generates 2, 2, and 7 simpler inputs; but none of them cause a failure. Thus, we go to level 4 where the fourth simpler input reproduces the failure. This input is further simplified by applying delta debugging at level 5

which yields a simpler input that is further simplified at level 6, yielding the 1-minimal input (SSNB/BP).

As we can see, when we apply delta debugging to the input decomposition tree, for leaf nodes we have two choices: always include the leaf node in the generated input (called IDTHDD), or reconsider this leaf node again when we go to next level (called as IDTHDD*). Figure 8 adopts the first choice. That is, assuming we are applying delta debugging to level l in the input decomposition tree, each simpler input we try consists of two parts: the input generated by delta debugging at level l, and inputs from leaf nodes in upper levels. For example, all inputs tested at level 2 include the leaf node in level 1 ("P") in Figure 8. Because all nodes which read input from outside will be the leaf node in the input decomposition tree, IDTHDD* guarantees that we can get 1-minimal input. Intuitively, IDTHDD* can generate smaller (or equal) inputs with more test runs, compared with IDTHDD. However, as our experiments (discussed soon) show, IDTHDD is already good enough to get similar minimized input with much fewer test runs, compared to IDTHDD*.

Comparison with standard delta debugging: As we have shown, for our running example, the original input of size 59 is converted to a simpler 1-minimal input of size 7 and during this process the program is executed on 17 different inputs (on 2 inputs in step 1, and on 15 inputs in step 3). We now compare these results with those obtained by standard delta debugging. We found that to identify a 1-minimal input, the standard delta debugging required executing the program on 222 different inputs. Moreover, it yielded a 1-minimal input whose size is 24 (H""/HSSNPPBP""/PP""/P/BP) in contrast to the 1-minimal input of size 7 (SSNB/BP) generated by our algorithm. Thus, pruning the search space using relevant input analysis is very effective in both reducing the size of the input and the number of executions required.

We observe that removal of irrelevant inputs by Step 1 is very useful in finding smaller 1-minimal inputs. This can be explained as follows. In general, for a given original input, there may be many 1-minimal inputs that can be derived from it. The larger the original input, the more likely it is that the sizes of these 1-minimal inputs vary significantly. Since the search for 1-minimal input terminates as soon as the first such input is found, we may end up with one of the larger 1-minimal inputs when standard delta debugging is used. On the other hand, our algorithm engages in the search for a 1-minimal input only after it has eliminated the irrelevant inputs. Starting from an already simpler (i.e., smaller) input is likely to yield a smaller 1-minimal input. This is indeed what happened in the above example. After the irrelevant inputs have been removed, the input's size is 10 which is much smaller than 24, size of the 1-minimal input found by standard delta debugging.

Finally, note that finding the 1-minimal input of size 7 from the input of size 10 produced after step 1 (i.e., SFFF-SNB/BP) required our algorithm to perform 15 executions of the program. On the other hand, if standard delta debugging is applied to this size 10 input (i.e., SFFFSNB/BP), it finds the same 1-minimal input as our algorithm after 37 executions of the program. Thus, guiding the search using the input decomposition tree also improves the efficiency of the search significantly (i.e., 15 vs. 37 executions).

Experimental evaluation: A summary of benchmarks used in our evaluation is shown in Table I; each benchmark contains a real reported bug in a widely-used program, with the details in columns 2-4. Tidy-34132 contains a NULL pointer dereference bug. It has a similar bug trigger condition as the example in Figure 1: a noframe tag is included in an inner frameset and some paragraphs are wrongly placed outside body. Bc-1.06 fails with a memory corruption error due to heap buffer overflow (variable v_count is misused due to a code clone error). Expat-1.95.3 fails when XML_DTD is not defined and an empty function pointer is dereferenced to allocate memory for an entity name.

Comparison with standard delta debugging: The comparison of our approach with standard delta debugging is summarized in Table III. The size of original failure-inducing input is given in the second column. The number of test runs and size of minimized input for standard delta debugging is given in third and fourth column respectively. The fifth (seventh, respectively) and sixth (eighth, respectively) columns show the number of test runs and size of minimized input for IDTHDD (IDTHDD*).

As we can see, for tidy-34132, IDTHDD only requires 176 test runs and produces a smaller input with size 44, while standard delta debugging needs to run 852 different inputs to produce the a minimized input with size 50. For the bug in bc-1.06, IDTHDD greatly outperforms standard delta debugging with *9 times* fewer test runs than standard delta debugging, while generating a slightly smaller input. For expat-1.95.3 IDTHDD generates a smaller input (52 vs. 63 characters) than standard delta debugging with much fewer test runs (8 times fewer) than standard delta debugging. For each of the three bugs, IDTHDD* generates a smaller input than IDTHDD, but requires more test runs.

To further evaluate how our relevant input analysis helps with delta debugging, the detailed comparison of our approach with standard delta debugging is presented in Table IV. The third and fourth columns show the number of test runs and size of simplified input for step 1 of our algorithm. The seventh (eighth, respectively) column shows the number of test runs for step 3 of IDTHDD (IDTHDD*, respectively). To show the effectiveness of input decomposition tree-based delta debugging, we also show the number of test runs and size of minimized input by applying standard delta debugging to the simplified input after step 1. As we can see, step 1 alone reduces the input size from 2018 to 124 for tidy-34132 (16x smaller) with 3 test runs, and from 1138 to 125 for expat-1.95.3 (9x smaller) with 2 test runs.

By comparing step 3 of IDTHDD with standard delta debugging, we can see that IDTHDD generates smaller (or equal) inputs with fewer test runs for the three bugs (e.g., 1192 test runs vs. 8372 runs for bc-1.06, and 214 test runs vs. 896 runs for expat-1.95.3).

TABLE III
SUMMARY OF COMPARISON WITH STANDARD DELTA DEBUGGING.

Program	Test Case (input size)	DDMIN		IDTHDD		IDTHDD*	
		# (test runs)	# (minimized input)	# (test runs)	# (minimized input)	# (test runs)	# (minimized input)
tidy-34132	test1.html(2018)	852	50	176	44	405	39
bc-1.06	test1.b(1310)	10800	191	1194	190	4185	190
expat-1.95.3	test1.xml(1138)	1785	63	216	52	393	49

TABLE IV
COMPARISON WITH STANDARD DELTA DEBUGGING AFTER STEP 1.

Program	Test Case (input size)	Step 1		DDMIN on Simplified Input		IDTHDD-Step3	IDTHDD*-Step3
		# (test runs)	# (simplified input)	# (test runs)	# (minimized input)	# (test runs)	# (test runs)
tidy-34132	test1.html(2018)	3	124	378	44	173	402
bc-1.06	test1.b(1310)	2	399	8372	190	1192	4183
expat-1.95.3	test1.xml(1138)	2	125	896	56	214	391

Comparison with hierarchical delta debugging: Our approach has several advantages compared to hierarchical delta debugging (HDD) [13]. First, HDD requires that the initial failure-inducing input be well-formed; otherwise, the parser which HDD is based on will fail. Note that HDD only generates syntactically valid input. However, it is common that programs often fail because of ill-formed input. For example, the original failure-inducing inputs for tidy-34132, expat-1.95.3 and the program in Figure 1 are ill-formed, so HDD would fail for such kind of bugs. Second, HDD users must provide infrastructure for input parsing, unparsing a configuration, and pruning nodes from the input tree for different languages, which turns out to be non-trivial [13].

B. Other Applications

Test Input Generation: Dynamic techniques require multiple program executions for generating desired test inputs [14]. The cost of test input generation can be reduced by using symbolic execution [15] for Java programs. For pointer based languages such as C, execution based analysis is more effective. A test input generation algorithm can make use of DERIVED and CINFLUENCED sets from a single execution to effectively derive test inputs at a moderate cost.

Security: Buffer overflow bugs can be exploited to launch attacks via which an attacker can transfer program control to malicious code. To detect such bugs, taint analysis can be used to track values that are data dependent on the inputs; if tainted values are used in computing branch target addresses, the vulnerability has been successfully detected [18]. However, data dependences may be obfuscated as control dependences to avoid detection. Our formation of chains in the DERIVED and AINFLUENCED sets help capture obfuscated vulnerabilities.

IV. CONCLUSIONS

We introduced a novel relevant input analysis that, for a particular execution, determines the role inputs play in deriving values, controlling branch predicate outcomes, and selecting referenced addresses. This information is useful for delta debugging, test input generation, and detecting potential security vulnerabilities. Experiments show that relevant input analysis significantly narrows down the scope of inputs that are relevant for computing a value during execution. The benefits of narrowing the scope were demonstrated by developing an effective and efficient delta debugging algorithm.

ACKNOWLEDGMENT

This research is supported by the NSF grants CCF-0963996 and CCF-1149632 to the University of California, Riverside.

REFERENCES

[1] Bodk, R. and Anik, S. Path-Sensitive Value-Flow Analysis. *POPL* 1998.
[2] Bodk, R., Gupta, R., and Soffa, M.L. Interprocedural Conditional Branch Elimination. *PLDI*, pages 146-158, 1997.
[3] Ferrante, J., Ottenstein, K., and Warren, J.D., The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319-349, 1987.
[4] Johnson, N.M., Caballero, J., Chen, K.Z., McCamant, S., Poosankam, P., Reynaud, D., and Song, D. Differential Slicing: Identifying Causal Execution Differences for Security Applications. *IEEE Symposium on Security and Privacy*, pages 347-362, 2011.
[5] Korel, B. and Laski, J.W. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187-195, 1990.
[6] Sridharan, M., Fink, S.J., and Bodk, R. Thin slicing. *PLDI*, pages 112-122, 2007.
[7] Xin, B. and Zhang, X. Efficient online detection of dynamic control dependence. *ISSTA*, pages 185–195, 2007.
[8] Zhang, X., Gupta, R., and Zhang, Y. Precise dynamic slicing algorithms. *ICSE* 2003.
[9] Lin, Z. and Zhang, X. Deriving input syntactic structure from execution. *FSE*, pages 83-93, 2008.
[10] Zeller, A. and Hildebrandt, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, pages 183–200, 2002.
[11] Cleve, H. and Zeller, A. Locating causes of program failures. *ICSE*, pages 342–351, 2005.
[12] Leitner, A., Oriol, M., Zeller, A., Ciupa, I., and Meyer, B. Efficient unit test case minimization. *ASE*, pages 417–420, 2007.
[13] Misherghi, G. and Su, Z. HDD: hierarchical Delta Debugging. *ICSE*, pages 142-151, 2006.
[14] Gupta, N, Mathur, A.P., and Soffa, M.L. Automated Test Data Generation Using an Iterative Relaxation Method. *FSE*, pages 231-244, 1998.
[15] Visser, W., Pasareanu, C.S., and Khurshid, S. Test input generation with java PathFinder. *ISSTA*, pages 97-107, 2004.
[16] Fraser, G. and Zeller, A. Generating Parameterized Unit Tests. *ISSTA*, pages 364–374, 2011.
[17] Rler, J., Fraser, G, Zeller, A., and Orso, A. Isolating Failure Causes through Test Case Generation. *ISSTA*, pages 309–319, 2012.
[18] Nagarajan, V., Kim, H-S., Wu, Y., and Gupta, R. Dynamic Information Flow Tracking on Multicores. *Workshop on Interaction between Compilers and Computer Architectures* (INTERACT), 10 pages, 2008.
[19] McCamant, S. and Ernst, Michael D. Quantitative information flow as network flow capacity. *PLDI*, pages 193–205, 2008.
[20] Narayanasamy, S., Pokam, G., and Calder, B. BugNet: Continuously recording program execution for deterministic replay debugging. *ISCA'05*.
[21] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI* 2005.
[22] Zhang, M., Zhang, X., Zhang, X., and Prabhakar, S. Tracing lineage beyond relational operators. *VLDB*, pages 1116–1127, 2007.
[23] Clause, J. and Orso, A. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. *ISSTA*, pages 249–260, 2009.
[24] Bao, T., Zheng, Y., Lin, Z., Zhang, X., and Xu, D. Strict control dependence and its effect on dynamic information flow analyses. *ISSTA*, pages 13–24, 2010.