User-Input Dependence Analysis via Graph Reachability

Bernhard Scholz* Chenyi Zhang Cristina Cifuentes
Sun Microsystems Laboratories
Brisbane, Australia
cristina.cifuentes@sun.com
*and The University of Sydney
scholz@it.usyd.edu.au

Abstract

Bug-checking tools have been used with some success in recent years to find bugs in software. For finding bugs that can cause security vulnerabilities, bug checking tools require a program analysis which determines whether a software bug can be controlled by user-input.

In this paper we introduce a static program analysis for computing user-input dependencies. This analysis can be used as a pre-processing filter to a static bug checking tool for identifying bugs that can potentially be exploited as security vulnerabilities. In order for the analysis to be applicable to large commercial software in the millions of lines of code, runtime speed and scalability of the user-input dependence analysis is of key importance.

Our user-input dependence analysis takes both data and control dependencies into account. We extend Static Single Assignment (SSA) form by augmenting phi-nodes with control dependencies. A formal definition of user-input dependence is expressed in a dataflow analysis framework as a Meet-Over-all-Paths (MOP) solution. We reduce the equation system to a sparse equation system exploiting the properties of SSA. The sparse equation system is solved as a reachability problem that results in a fast algorithm for computing user-input dependencies. We have implemented a call-insensitive and a call-sensitive analysis. The paper gives preliminary results on the comparison of their efficiency for various benchmarks.

1. Introduction

A security vulnerability is a software bug that can be exploited by malicious input to gain control over a system. Worms, including the Microsoft SQL server Slammer [18] and the Sun Telnet worm [28], exploit security vulnerabilities in software and can compromise hundreds of thousands of computers on the Internet within minutes, causing millions of dollars damage. Manual code inspection is current industry practice to find security vulnerabilities in code. An auditor analyzes the code for bugs that can be controlled by user-input. These inspections are time-consuming, repetitive and tedious. In recent years, bug checking tools that use static program analysis have successfully found bugs in software [3, 10, 1, 7, 9]. For classifying bugs as potential security vulnerabilities, a bug checking tool needs to test whether a detected bug is dependent on user-input.

The dynamic scripting language Perl [4] implements a user-input dependence test as a security feature called taint mode. Data from an untrusted source is tracked and marked as "tainted", dynamically, as the program is executed. A variable on the left-hand side of an assignment becomes tainted if there is a tainted value on the right-hand side, i.e., the variable on the left-hand side is data dependent on the variables on the right-hand side. At runtime Perl checks the arguments of a system call. If the arguments are tainted, a security error is raised. In Perl's taint mode data dependencies are considered but control dependencies are not taken into account. However, data dependencies are insufficient to track data from an untrusted source. For example, the Perl program `$a=<>;$b=$a;system("echo $b");` reads in a value, stores the value in `$a`, assigns the value of `$a` to `$b`, and outputs the content of variable `$b`. If this program is executed in taint mode, variable `$b` becomes tainted and the program terminates with an "insecure" error. Let's assume that variable `$a` can only read the values 0 and 1. Then, the statement `$b=$a;` can be rewritten to `if($a==1){$b=1;}else{$b=0;}` and Perl's taint mode cannot capture this implicit data dependency.

Static program analysis has been used to compute user-input dependencies for security vulnerabilities [13, 24]. The advantage of static program analysis is that it can take control dependencies into account and the analysis can consider all paths in the program, whereas dynamic program analysis exercises a single execution path. In this paper we propose

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.22

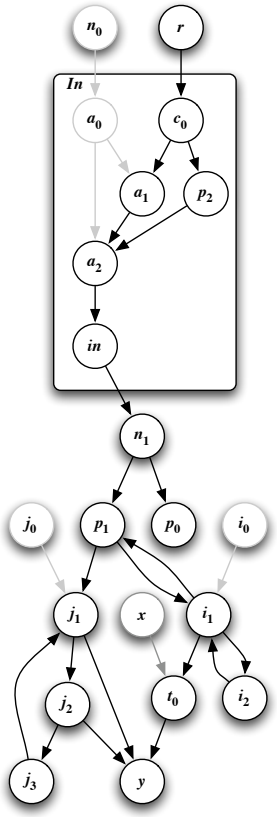
```
 1 void copy_to_utf()
 2 {
 3  int n = 0,
 4      i,
 5      j;
 6  char x[BUFSIZ],
 7       y[BUFSIZ];
 8  ...
 9  n = in(n);
10  if (n > 0) {
11   j = 0;
12   for (i=0;i<n;i++){
13    y[j++] = x[i];
14    y[j++] = 0;
15   }
16  }
17  ...
18 }
19 int in(int a)
20 {
21  int c = getchar();
22  if (isdigit(c)) {
23   a = a + c - '0';
24  }
25  return a;
26 }
```

(a) Input Program

```
     ...
     n1:=in(n0);
     p0:=(n1>0);
     if(¬p0) goto ex;
     j0:=0;
     i0:=0;
for: i1:=φ'(i0,i2;p1);
     j1:=φ'(j0,j3;p1);
     p1:=(i1<n1);
     if(¬p1) goto ex;
     t0:=load x(i1);
     store y(j1),t0;
     j2:=j1+1;
     store y(j2),0;
     j3:=j2+1;
     i2:=i1+1;
     goto for;
ex:  ...

int in(int a0){
     c0:=getchar();
     p2:=isdigit(c0);
     if(¬p2) goto br;
     a1:=a0+c0-10;
br:  a2:=φ'(a0,a1;p2);
     return a2;
}
```

(b) aSSA



(c) Reachability Graph

Figure 1. Motivating example

a new static program analysis technique for locating user-input dependencies in programs based on SSA form [6]. This analysis can be used as a pre-processing pass to a static bug checking tool for finding the relevant statements in a program that are prone to vulnerabilities. Runtime speed and scalability of this filtering phase is important for use in large commercial software in the order of millions of lines of source code. It is beyond the scope of this paper to discuss and detect security vulnerabilities in programs. In this work we present a fast and scalable analysis for determining user-input dependencies of statements and variables statically.

The contributions of this work are as follows: (1) the solution of user-input dependence as a Meet-Over-all-Paths problem, (2) the introduction of Augmented Static Single Assignment (aSSA) form, that makes control dependencies upon the values in phi-nodes explicit, (3) a fast algorithm for computing user-input dependencies that reduces the data flow equation system to a sparse equation system that is solved via a graph reachability problem in a rooted directed graph, and (4) an inter-procedural call-sensitive and call-insensitive extension of the analysis.

The rest of this paper is organized as follows. Section 2 demonstrates our approach based on a motivating example, Section 3 presents the user-input dependence analysis, and Section 4 describes the implementation and the preliminary results for this work. In Section 5 we survey related work. We conclude with future work and conclusions.

2. Motivating Example

We illustrate an example in Figure 1(a) that demonstrates our user input-dependence analysis. The example comprises a C-code fragment for copying a character string in ANSI code to a Unicode array. The function declares two fixed-size character arrays `x` and `y` of the same length (`BUFSIZ`). The variable `n` has value zero before entering the code fragment and is passed on to function `in`. Function `in` reads a single character from standard input. If the character is a digit, then the value of the character will be added to argument `a` and returned by function `in`. The result of `in` is assigned to variable `n` and checked to be greater than 0. Inside the then-branch, the for-loop controls variable `i` that ranges from 0 to `n-1`. Inside the loop body, the content of array `x` is copied to array `y`, with a zero-byte padding.

In the example, a **buffer overflow** may occur in line 13

and line 14 if the length of array y is too small to hold twice the number of characters of array x, i.e., when index j is greater than or equal to BUFSIZ—the size of array y. Index j is control dependent on variable n, which in turn is dependent on the result of function in. In function in, the return value of the library function getchar is user-input dependent and used for the computation of the return value of function in. Hence, the result of function in is dependent on user-input. The buffer overflow poses a potential security vulnerability because it can be exploited via user-input. In the example, an **out of bound array access** may occur in line 13 if index i is greater than or equal to BUFSIZ—the size of array x. Since i is dependent on user-input, it may imply a potential security vulnerability as well.

The aSSA form of our motivating example is given in Figure 1(b). All variables have a single assignment, higher-level control-flow constructs are reduced to if-gotos, and at confluence points we introduce augmented phi-nodes which incorporate both control and data dependencies (see Section 3). Note that the example makes use of load and store instructions to denote read and write accesses to memory, including address computation.

Augmented phi-nodes i_1 and j_1 are extended phi-nodes of the SSA form [6]. They control whether the variable values of i_1 and j_1, respectively, are taken from inside or outside the loop depending on predicate p_1. Note that predicate p_0 of the outer if-statement is not involved in the selection, though both statements are only executable if p_0 holds. Similarly, the augmented phi-node of a_2 selects between the value a_0 and a_1 depending on predicate p_2.

We map the user-input dependence test to a graph reachability problem in a rooted directed graph. Graph reachability analysis checks whether there exists a path from the root node to a node in the graph. A simple graph traversal can compute this problem in $\mathcal{O}(n+m)$ where n is the number of nodes and m is the number of edges in the directed graph. Nodes in the reachability graph represent results of instructions (i.e., local variables in SSA form), functions, function arguments, and global variables. The root node is special and represents input that is controlled by the user. Edges in the rooted directed graph represent either data or control dependencies between the nodes. If a node is reachable from the root node, the user may control the value of the node and the value becomes user-input dependent. In Figure 1(c) the reachability graph of our example is depicted. The unreachable nodes are depicted in gray whereas the reachable nodes are depicted in black. All aSSA variables and the two arrays x and y are each represented by a node.

Consider the value $i_2 := i_1 + 1$ that has a single data dependency on its right-hand side, i.e., i_1. There is an edge from i_1 to i_2 modeling the data dependency on i_1. For an augmented phi-node we have two kinds of incoming edges: (1) edges representing data dependencies and (2) edges representing control dependencies. For instance, the value $i_1 := \phi'(i_0, i_2; p_1)$ depends on i_0 and i_2 by data dependencies but also on p_1 by control dependency. Therefore, node i_1 has incoming edges from i_0, i_2 and p_1. Function calls are mapped to the reachability graph as follows: the nodes of the actual arguments are connected to their associated nodes representing the formal arguments of the function. The function node itself is connected to the left-hand side of the assignment for the return value, and a return expression inside a function is linked to its function node. For example, the actual argument n_0 is connected to the formal argument a_0 and the return value a_2 is connected to the function node in. Variable n_1 that is assigned the result of the function in has the in-coming edge from in. Furthermore, we have two library calls in our example. The call to getchar returns a value controlled by the user. Therefore, we connect c_0 with the root node r. The library call to isdigit checks whether the argument is a digit, and, therefore, connects the actual argument with its result.

As shown in Figure 1(c) the array indices i_1 and j_1 are dependent on user-input. Hence, any bugs in the C-code dependent on these values can be potentially exploited as a security vulnerability, in particular, lines 13 and 14 of the C-code in Figure 1(a).

3. User-Input Dependence Analysis

SSA form provides an efficient representation of the def-use relation on data dependencies without introducing any false dependencies [6]. However, in this work control dependencies [30, 11] are required as well, which are implicit in SSA form. To represent control dependencies, we augment SSA phi-nodes as follows,

$$x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l) \tag{1}$$

where we write Y_x (the selection set) for the set $\{y_1, \ldots, y_k\}$ and P_x (the control set) for $\{p_1, \ldots, p_l\}$. Informally, P_x are the set of nodes which contribute to the selection of a value from the set Y_x, but P_x does not explicitly state how to make the choice. Therefore an augmented phi-node is an abstracted gating function [27, 19] [1]. As explained in Section 5, the complexity of aSSA is smaller than that of GSA, which is important to our needs of performance and scalability in large code bases. In the rest of this section we formalize a Meet-Over-all-Paths solution for user-input dependence and show that it can be solved as a graph reachability problem. Proofs for the paper are available in its extended version [23].

[1] In Gated Single Assignment (GSA) form, a gating function explicitly decides a unique $y \in Y_x$ from the value of members in P_x.

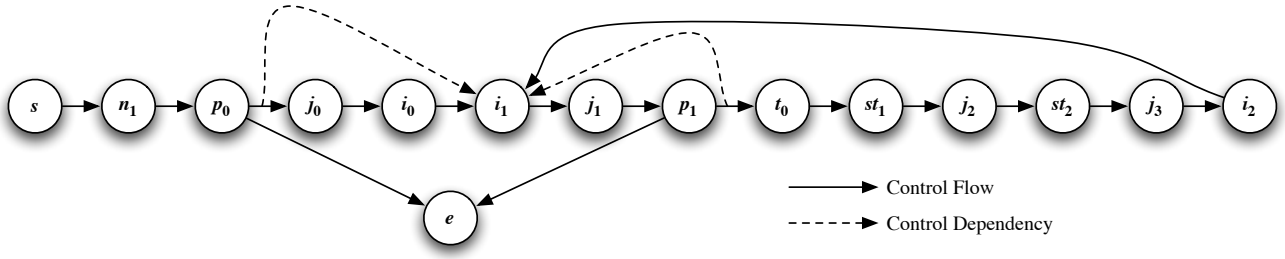


Figure 2. Control dependencies of node $\texttt{i}_1$ in flowgraph of Figure 1

3.1. Definition of Tainted Values

The user-input dependence analysis obtains the information whether a variable in a program is potentially tainted (i.e., dependent on user-input) or strictly untainted (i.e., not dependent on user-input). For a single variable this information can be represented in a semi-lattice $(\mathcal{L}, \sqcap)$ that consists of element $\blacktriangle$ representing the tainted value and $\triangle$ representing the untainted value. The meet operation is defined by:

$$a \sqcap b = \begin{cases} \triangle, & \text{if } a = \triangle \text{ and } b = \triangle, \\ \blacktriangle, & \text{otherwise.} \end{cases} \quad (2)$$

Semi-lattice $(\mathcal{L}, \sqcap)$ is isomorphic to the boolean semi-lattice $(\mathbb{B}, \vee)$ assuming element $\blacktriangle$ is 1 and element $\triangle$ is 0. Hence, the meet operation has the properties of commutativity, associativity, and idempotence. We extend the meet operation to $\bigsqcap_{i \in I} a_i$ for any countable set I. If the index set I is empty, then the result of the meet operation is $\triangle$ by convention. The semi-lattice imposes a partial order $\sqsubseteq$, such that $a \sqsubseteq b \Leftrightarrow a \sqcap b = a$. In this partial order set $\mathcal{L}$ element $\triangle$ is the top and $\blacktriangle$ is the bottom element.

We define an information lattice $(\mathcal{L}^n, \sqcap, \blacktriangle^n, \triangle^n)$ where n is the number of variables. An element $\vec{c} \in \mathcal{L}^n$ is called *configuration* and represents the taint information of variables that we may assume at a certain point in the flowgraph. We write *Var* for the set of variables in the program. We associate a unique index i_x ($i_x = 1 \ldots n$) to a variable $x \in$ *Var* that denotes the position of x in a vector of size n. The result of meet operation $\vec{a} \sqcap \vec{b}$ is vector $\vec{c}$ with element $c_i = a_i \sqcap b_i$ for all $i = 1 \ldots n$. The top and bottom elements are $\blacktriangle^n = \langle \blacktriangle, \ldots, \blacktriangle \rangle$ and $\triangle^n = \langle \triangle, \ldots, \triangle \rangle$, respectively. For the sake of readability, we use the notation $\vec{c}(x)$ for element c_{i_x}, and $\vec{c}_{[x \leftarrow a]}$ is a configuration identical to $\vec{c}$ except for $\vec{c}_{[x \leftarrow a]}(x) = a$.

We employ the notions of a distributive data flow analysis framework [16] to describe taint information. We define the MOP solutions for a node $u \in N$ by

$$\mathit{mop}(u) = \bigsqcap_{\pi \in \mathit{Path}(s,u)} M(\pi)(\triangle^n). \quad (3)$$

Function M describes the transfer function of node u and is extended to paths, i.e., $M(\pi)$ is the function composition $M(u_k) \circ \cdots \circ M(u_1)$ of path $\pi = (u_1, \ldots, u_k)$. If π is the empty path, then function $M(\pi)$ is the identity function. Note that we do not differentiate between a statement of a node (as either an assignment or predicate of a branch) and the node itself. The transfer functions $M[\![.]\!] : N \rightarrow (\mathcal{L}^n \rightarrow \mathcal{L}^n)$ are defined in the following.

- **Nop Statement:** does not change the configuration, i.e., $M[\![\texttt{nop}]\!](\vec{c}) = \vec{c}$.
- **Read Operation:** taints variable x, i.e., $M[\![x := \texttt{read}]\!](\vec{c}) = \vec{c}_{[x \leftarrow \blacktriangle]}$.
- **Assignment:** If the right-hand side of an assignment contains a tainted value, then the variable on the left-hand side becomes tainted, i.e., $M[\![x := \mathit{op}(y_1, \ldots, y_k)]\!](\vec{c}) = \vec{c}_{[x \leftarrow \bigsqcap_{1 \le i \le k} \vec{c}(y_i)]}$; if there are no variables on the right-hand side, then $M[\![x := \mathit{op}()]\!](\vec{c}) = \vec{c}_{[x \leftarrow \triangle]}$.
- **Augmented Phi-Node:** If one of the arguments or one of the predicates is tainted, then the result will be tainted, i.e., $M[\![x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l)]\!](\vec{c}) = \vec{c}_{[x \leftarrow \bigsqcap_{1 \le i \le k} \vec{c}(y_i) \sqcap \bigsqcap_{1 \le j \le l} \vec{c}(p_j)]}$, where predicate p_i ($i = 1 \ldots l$) refers to a controlling if-statement of x. Note that if-statements are modeled as assignments that have two successor nodes.

In aSSA form variables have a single assignment in the flowgraph. Therefore, a variable can only become tainted at the node that contains its assignment.

Definition 1 *Variable $v \in Var$ in aSSA form is untainted, if $\mathit{mop}(v)(v)$ is $\triangle$.*

The information lattice $(\mathcal{L}^n, \sqcap, \blacktriangle^n, \triangle^n)$ with transfer function $M[\![.]\!]$ is an instance of a monotone and distributive dataflow analysis framework, on which we define a simultaneous equation system:

$$\vec{z}_u = M(u) \left(\bigsqcap_{v \in \mathit{preds}(u)} \vec{z}_v \right) \quad \text{for all } u \in N. \quad (4)$$

Note that variable $\vec{z}_u$ is a vector in $\mathcal{L}^n$ that has n elements in $\mathcal{L}$, and there are $|N|$ equations. Hence, the equation system has $\mathcal{O}(n^2)$ variables in $\mathcal{L}$.

Let $Z \in \mathcal{L}^{|N|\times n}$ denote the vector of variables $\{z_u\}_{u\in N}$ in the simultaneous equation system, then a concise notation of this equation system is $Z = F(Z)$, where F is the right-hand side of the equations. It can be shown that function F is monotone and distributive, therefore, there exists a maximum fixed point *mfp*(F). We write *mfp*(u) for the maximum fixed-point of F on node u.

Theorem 1 *mop*$(u)(x) =$ *mfp*$(u)(x)$ *for all* $u \in N$ *and* $x \in$ *Var.*

Since lattice $\mathcal{L}^n$ has finite height, the maximum fixed point is computed by a finite number of applications of F on the top element, i.e., $F \circ \cdots \circ F(\triangle^{|N|\times n})$.

3.2. Compression of the Simultaneous Equation System

SSA form has specific properties that allow the compression of the simultaneous equation system to $\mathcal{O}(n)$ variables in $\mathcal{L}$. We make the following observations about the structure of statements in aSSA form:

1. For each variable there exists a single assignment in the program, i.e., for all $x \in$ *Var*, there is a unique $u \in N$ such that x is defined at u. We use $u \in N$ as a synonym for $x \in$ *Var* if u defines x, and vice versa.

2. Every node $x := op(y_1, \ldots, y_k)$ is dominated by y_i for all $i = 1 \ldots k$, hence all definitions y_i reach node x.

3. Every augmented phi-node $x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l)$ is not necessarily dominated by elements in Y_x, however, we still have statement x reachable from y for all $y \in Y_x$.

4. Every augmented phi-node $x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l)$ is reachable by its predicates p_i for all $p_i \in P_x$.

We observe for each assignment $x := op(y_1, \ldots, y_k)$, each y_i is equal to the value *mop*$(x)(y_i)$ at y_i's definition node since node x is reachable from y_i for all $i = 1 \ldots k$. For an augmented phi-node x it holds as well for all variables on its right-hand side $y \in Y_x$ and all predicates $p \in P_x$.

This observation allows us to construct a new simultaneous equation system with variables $\widehat{z}_x \in \mathcal{L}$ for all $x \in$ *Var*.

$$\widehat{z}_x = \begin{cases} \blacktriangle, & \text{if } x := \texttt{read}, \\ \triangle, & \text{if } x := op(), \\ \bigsqcap_{1\le i\le k} \widehat{z}_{y_i}, & \text{if } x := op(y_1, \ldots, y_k), \\ \bigsqcap_{1\le i\le k} \widehat{z}_{y_i} \sqcap \bigsqcap_{1\le j\le l} \widehat{z}_{p_j}, & \text{if } x := \phi'(y_1, \ldots, y_k; \\ & \qquad p_1, \ldots, p_l). \end{cases} \tag{5}$$

Let $\widehat{\vec{z}} \in \mathcal{L}^n$ denote the vector of variables and $\widehat{F}$ the right-hand side of the equation system. Since $\widehat{F}$ is monotone, there is a maximal fixed point *mfp*$(\widehat{F})$. We write $\widehat{mfp}(x)$ for the maximal solution of $\widehat{F}$ on variable x.

Theorem 2 *For all* $x \in Var$, *mop*$(x)(x) = \widehat{mfp}(x)$.

3.3. Linear Boolean Equation Systems and Reachability

The compressed equation system is solved by a reachability graph. To show that the reachability solves the maximum fixed point of the compressed equation system, we establish a relationship between a *linear boolean equation system* and the reachability graph. Later we show that any instance of the compressed equation system is solvable by a linear boolean equation system. Note that the linear boolean equation system is a theoretical vehicle. In the actual implementation the reachability graph is constructed directly from the flowgraph.

For the boolean lattice $(\mathbb{B}, \vee, \wedge, 1, 0)$ we establish a partial order $a \le b$ if $a \vee b = a$. In this partial order 0 is the top element and 1 is the bottom element. The partial order $\le$ is further extended to vectors, i.e., $\vec{a}, \vec{b} \in \mathbb{B}^n$, $\vec{a} \le \vec{b}$, if $a_i \le b_i$, for all $i = 1, \ldots, n$.

Definition 2 *Given* $A \in \mathbb{B}^{n\times n}$ *and* $\vec{b} \in \mathbb{B}^n$ *in the boolean lattice* $(\mathbb{B}, \vee, \wedge, 1, 0)$, *define* $\vec{x} \in \mathbb{B}^n$ *the maximal solution of the boolean equation system*

$$x_i = (\bigvee_{j=1}^{n} a_{ij} \wedge x_j) \vee b_i, \quad \textit{for } i = 1, \ldots, n. \tag{6}$$

where a_{ij} *is the element of matrix* A *in row* i *and column* j, *and* x_i *and* b_i *are the* i*th elements of vector* $\vec{x}$ *and* $\vec{b}$, *respectively.*

We associate to Equations (6) a reachability graph that is a rooted directed graph $G = (V, Arc, r)$ where $V = \{r, v_1, \ldots, v_n\}$, r is the distinguished root node, and $Arc = \{(v_j, v_i) \in V \times V | a_{ij} = 1\} \cup \{(r, v_i) \in V \times V | b_i = 1\}$. We associate node v_i in G with variable x_i in the linear boolean equation system. A node $v_i \in V$ is in the set of *reachable nodes* $R \subseteq V$, if there exists a path from r to v_i. We also say that a node u is *reachable* if $u \in R$.

Theorem 3 *In the maximal solution $\vec{x}$, an element x_i has value* 1 *iff* $v_i \in R$.

The next theorem connects the solution $\vec{x}$ of the linear boolean equation system with the MOP solution, where we assume there is a one-to-one mapping from each variable in the compressed equation system of Equations (5) to a unique variable in the linear boolean equation system. Note that semi-lattice $(\mathbb{B}, \vee)$ is isomorphic to $(\mathcal{L}, \sqcap)$.

Theorem 4 *Given the same index $I = \{1, \ldots, n\}$, in the maximal solution $\vec{x}$ of Equations* (6)*, we have for all $i = 1, \ldots, n$,*

$$\vec{x}_i = 1 \text{ iff } \widehat{mfp}(u_i) = \blacktriangle$$

3.4. Inter-procedural Analysis, Arrays, and Pointers

We have two approaches for the inter-procedural user-input dependence analysis: a call-insensitive and a call-sensitive analysis. The insensitive analysis is less precise but fast, whereas the sensitive analysis has more precision at the expense of longer runtimes (cf. Section 4). The **call-insensitive** analysis maps the whole program to a single reachability graph using the mapping as sketched in the motivating example in Figure 1. The following outlines the idea: for a function f we add a new variable f to *Var*, that represents the return value of f. Value $mop(u)(f)$ reflects whether the return value of f is tainted at node u. The transfer function of a call-site is $M[\![x := \texttt{call}\, f\,(y_1, \ldots, y_k)]\!](\vec{c}) = \vec{c}_{[x \leftarrow \vec{c}(f), a_1 \leftarrow \vec{c}(y_1), \ldots, a_k \leftarrow \vec{c}(y_k)]}$ where $y_1, \ldots, y_k$ are actual arguments and $a_1, \ldots, a_k$ are formal arguments of function f. The result x becomes tainted if f is tainted. A formal argument a_i becomes tainted if the actual argument y_i is tainted.

The **call-sensitive** approach is performed in two phases. In the first phase the call-graph of the input program is split into a set of topologically ordered strongly connected components (SCC) containing functions that potentially invoke recursively each other. Each SCC is analyzed in reverse topological order by constructing a separate reachability graph for the SCC. For each function in the SCC a summary function that expresses user input dependencies between global variables, arguments and result values of functions is constructed. A call-site in the SCC invokes either a function that is in the SCC, or a function for which there exists already a summary function, or an external function (libraries, system calls, etc.). In the former case, we use the connection scheme as used for the insensitive-analysis. In the latter case, we wire the dependencies as given in the summary function. For external functions, we rely on specifications as shown in Figure 3 and explained in Section 4. After constructing the reachability graph we probe

```
/* user-input dependent variables,
   arguments, results */
_input stdin;
_input int scanf(const char *, _input ...);
_input int getchar(void);
int main(_input int argc, _input char *argv);

/* summary functions */
int isdigit(int n){ return n; }
char *strcpy(char *str1, char *str2 {
             str1 = str2; return str2; }
void *malloc(size_t size){ return size; }
```

Figure 3. Excerpt of configuration file

which arguments, globals, and results of function are user-input dependent, and mark them user-input dependent in the summary functions of the SCC. Dependencies to arguments and globals are computed by resetting the root node of the reachability graph to either a global or an argument and probing the connectivity for arguments, globals and result of functions again. The second phase proceeds in the topological order, propagating tainted information from callers to callees. The worst-case complexity of the sensitive approach is $\mathcal{O}(\sum_{s \in SCC}[(n_g(s) + n_a(s)) * (n(s) + m(s)) + n_g(s)^2 + n_a(s) * n_g(s) + n_a(s)^2]$ where $n_g(s)$ and $n_a(s)$ are the number of global variables and the number of arguments in strongly connected component s, and $n(s)$ and $m(s)$ are the number of nodes and edges in the reachability graph of a strongly connected component s. Note for both the insensitive and the sensitive analysis we use a simple may-alias analysis for indirect call-sites that gives a set of possibly invoked functions for a call-site. If this set cannot be determined, we make the worst-case assumption that the arguments and the result become user-input dependent.

The call-sensitive approach is call-sensitive in the absence of immediate recursive functions. If the call graph contains SCCs with more than one function, we construct a single reachability graph for the strongly connected component. Hence, the functions inside the SCC are treated in a call-insensitive fashion. To implement a call-sensitive approach for immediate recursive functions, a more sophisticated solver than reachability would be required. For example, a recursive boolean function solver [14] could be used to solve immediate recursive functions in a call-sensitive fashion at the expense of exponential worst-case time complexity.

Arrays: for an array a we introduce a new variable in *Var*. The meaning of $mop(u)(a)$ reflects whether the contents of a is tainted in node u. We have a transfer function for reading an element and a transfer function for writing an element in the array. The transfer function for a read is $M[\![x := \texttt{load}\, a(y)]\!](\vec{c}) = \vec{c}_{[x \leftarrow \vec{c}(a) \sqcap \vec{c}(y)]}$, i.e., the result of

	Problem Size		Dependence		Array Access							
Progs	#loc	#inst	%uii$_i$	%uii$_s$	#cr	#cw	#ncr	#ncw	%uir$_i$	%uir$_s$	%uiw$_i$	%uiw$_s$
mysqld-4.1.22	1111594	1321523	81.2	76.6	95617	62938	11168	5021	90.7	88.9	93.4	91.6
mysqld-3.22.32	218531	202847	74.3	68.8	14342	7005	2774	1176	86.1	83.4	89.4	88.6
sendmail	179753	169166	72.4	63.8	15923	7300	4303	1152	92.8	90.2	81.8	77.1
httpd	103066	164162	81.0	74.0	11849	6610	3113	690	91.7	90.8	91.7	90.6
perlbmk	85464	176866	74.4	72.0	19333	9137	1823	1592	98.2	97.9	94.8	94.5
vortex	67220	65685	70.7	66.6	4512	2473	1106	398	93.9	90.9	95.2	95.0
pppd	27048	32540	56.4	39.1	1899	1409	1380	731	41.4	28.8	31.6	18.9
sshd	20729	18489	66.7	59.6	1644	736	273	123	83.2	63.7	74.8	54.5
mailx	14609	25717	69.8	54.2	880	713	843	254	91.9	83.0	83.5	75.6
zoneadmd	7485	7835	64.4	63.0	239	247	163	28	95.1	95.1	85.7	85.7
mail	6934	7286	55.6	48.2	272	148	220	73	92.7	87.7	79.5	78.1

Table 1. Experiment: problem size and percentages of user-input dependent instructions. *#loc* is the number of lines of code, *#inst* is the number of instructions in LLVM's IR, *%uii* is the percentages of user-input dependent instructions, *#cr* is the number of constant read array accesses, *#cw* is the number of constant write array accesses, *#ncr* is the number of non-constant read array accesses, *#ncw* is the number of non-constant write array accesses, *%uir* and *%uiw* are the percentages of user-input dependent read and write accesses as a percentage of non-constant array accesses. Note the subscripts 'i' and 's' indicate the call-insensitive case and the call-sensitive case, respectively.

the read access becomes tainted if either the index is tainted or the contents of the array is tainted. The transfer function for a write is $M[\![\texttt{store}\, a(y), x]\!](\vec{c}) = \vec{c}_{[a \leftarrow \vec{c}(x) \sqcap \vec{c}(y)]}$, i.e., the array becomes tainted if either the index or the value is tainted. Similar to functions, all write accesses are joined with a meet operation in the compressed equation system. For global variables we have two transfer functions for the read and write access, i.e., $M[\![x := \texttt{load}\, g]\!](\vec{c}) = \vec{c}_{[x \leftarrow \vec{c}(g)]}$ and $M[\![\texttt{store}\, g, x]\!](\vec{c}) = \vec{c}_{[g \leftarrow \vec{c}(x)]}$.

Pointers: for pointers we encode a simple may points-to analysis in the reachability graph. We consider the load and store operations for pointers separately. Both store and load operations might taint data of the program, e.g., "`store` p, x" adds an edge from x to p, and "$x :=$ `load` p" adds an edge from p to x. To handle the effect that an address value is loaded into a variable, we add reverse edges to load operations (and phi-nodes) such that all possible memory objects that might be referenced in the store operations become connected. For pointer arguments the mapping of call-sites needs to be extended as well. A reverse edge is added between the actual and formal argument to ensure that taint information can traverse from the callee to the caller through the pointer arguments.

4. Preliminary Results

We have implemented the call-insensitive and call-sensitive user-input dependence analysis in the LLVM framework [17], which is a low-level virtual machine for the C programming language family. Its instruction set is strictly typed and has been designed for a virtual architecture that avoids machine specific constraints. Every value or memory location has an associated type and all instructions obey strict type rules. LLVM code is represented in SSA form. We implemented a may-alias analysis for function pointers to better support the accuracy of our user-input dependence analysis. We also use our own may-alias analysis in the reachability graph.

The result of the user-input dependence analysis are annotations in the intermediate representation of LLVM, denoting which variables and statements are user-input dependent. When used in combination with the Parfait bug checker [5], these annotations guide other program analyses in our bug checking tool to find security bugs. For example, for checking security-relevant array accesses that are out of bound, the bug checking tool analyzes only the array accesses that are dependent on user inputs; reducing the number of array accesses that are checked.

The user-input dependence analysis reads the input program as an LLVM bytecode file and a configuration file that specifies which external global variables, arguments and results of functions are user-input dependent, as well as output dependencies on inputs to a function. An excerpt of a configuration file for functions and globals in the C library is listed in Figure 3. The qualifier `_input` declares a global variable, arguments or results of a function as user-input dependent. For example, the `main` function has two user-input dependent arguments (`argc` and `argv`) that are controlled by the user. Therefore, in the declaration, the qualifier `_input` is added. Summary functions

	Problem Size			Insensitive		Sensitive			Runtime	
Progs	#inst	#fn	#glob	#nod$_i$	#edg$_i$	#scc	#nod$_s$	#edg$_s$	τ_i	τ_s
mysqld-4.1.22	1111594	6151	31580	1370632	1903894	5737	1465480	2232975	153.4	361.29
mysqld-3.22.32	218531	1453	4987	211073	314668	1025	212886	324410	6.49	8.74
sendmail	169166	1002	4416	176242	499148	644	254541	4508623	6.51	71.01
httpd	164162	966	5348	172351	318029	792	186347	376830	5.02	8.95
perlbmk	176866	841	1964	180301	333772	520	492883	3154620	3.28	2018.19
vortex	65685	288	1342	68255	124067	218	87048	375277	0.43	2.42
pppd	32540	433	1714	35014	67045	272	44406	98754	0.24	0.64
sshd	18489	582	1105	20074	32805	165	22546	35851	0.12	0.29
mailx	25717	276	682	26820	61359	169	31331	77000	0.21	0.44
zoneadmd	7835	324	439	8396	16900	31	8773	17996	0.07	0.11
mail	7286	131	465	7860	40925	35	8321	42281	0.2	0.25

Table 2. Experiment: reachability graph size and the running time. *#inst* is the number of instructions in the IR, *#fn* is the number of functions in the program, *#glob* is the number of globals in the program, *#scc* is the number of strongly connected components in the call-sensitive case (in the call-insensitive case it is always 1), *#nod* and *#edg* are the numbers of nodes and edges in the reachability graph, and τ is the analysis time in seconds. Note the subscripts 'i' and 's' indicate the call-insensitive case and the call-sensitive case, respectively.

express dependencies rather than operations per se. For example, the `strcpy` summary function states that the argument `str1` and the result of the function depends on the argument `str2`, i.e., if `str2` is user-input dependent the result of `strcpy` and the actual argument `str1` will become user-input dependent.

To evaluate our user-input dependence analyses, we use benchmarks including programs from the OpenSolaris™ operating system, two versions of the MySQL ™ database, the Apache httpd server (v2.2.6), and programs from SPEC CINT 2000. Table 1 gives the problem sizes and a comparison with respect to the percentage of instructions that are user-input dependent in the benchmarks. The percentage of user-input dependent instructions ranges from 60% to 85%[2] for the insensitive analysis and from 42% to 78% for the sensitive analysis (see column %uii$_i$ and %uii$_s$ in Table 1). For array accesses, the number of user-input dependent accesses is small for both analysis. Constant array accesses are usually not regarded as security vulnerabilities, therefore, the security bug checking tool will only analyze a small fraction of all array accesses in the program. The results show that, on average, 83% of non-constant array accesses are user-input dependent for the insensitive analysis and 78% for the context sensitive analysis.

The runtime of the insensitive and sensitive analysis varies significantly. As seen in Table 2, the insensitive analysis is linear on the number of instructions in the intermediate representation. The sensitive analysis uses significantly more time, since for each strongly connected component in the call graph, the dependencies between global variables, arguments and results need to be re-computed separately (as shown in column τ_s in Table 2). The analysis was executed on a SUN Fire X4600 (16GB Ram, 4xOpteron 8220) under light load. For the perl benchmark it takes the sensitive analysis more than 33 minutes, this is because the sensitive approach considers too many global variables in different strongly connected components, which results in a much longer runtime. More experimental data and comparisons are needed for this work, as outlined in Section 6.

[2] A portion of standard C library is not fully specified in the configuration file and worst-case assumption were made.

5. Related Work

In this section we review the literature in the areas of user-input dependence analysis, taint analysis, information flow, slicing, and data flow analysis.

User-Input Dependence Analysis. Static taint analysis [29, 20, 12] and user-input dependence analysis [13, 24] are concerned with tracking user-input data in source code. In contrast to static taint analysis, user-input dependence analysis does not have any notion of sanitization, which is a mechanism used to untaint data after it has been sanity checked.

Static approaches for taint analysis and user-input dependence analysis include type systems [12], data flow analysis [20], and Program Dependence Graph (PDG) [13, 24] with path conditions. Type checking is efficient, but in practice it is potentially less precise than our approach. PDG represents control-dependencies explicitly, but in practice their control relation is too strong for our purpose (in our approach only those conditions that join the selection of a

value at a confluence point are taken into account). The path conditions are helpful to eliminate infeasible paths which makes the analysis more precise. Def-use relations are explicit through use of an SSA representation.

Program Dependence Web (PDW) achieves both control and data-dependencies [19, 27], but its construction and representation is more elaborate. Programs in PDW are represented as Gated Single Assignment (GSA) form, where phi-nodes in SSA are replaced by gating functions that explicitly decide how to make a selection from the potential values. GSA form uses path expressions to represent the condition under which an operand of a phi-node is selected. Path expressions are regular expressions whose alphabet is the set of edges in the control flow graph. Without reusing re-occurring regular expression in its representation, the space complexity of GSA form is in the worst case $\mathcal{O}(n^3 * 4^n)$ [15]. By reusing common sub-expressions in GSA form (which might be less preferred if conditions need to be traversed separately), the space complexity is reduced between $\mathcal{O}(m\alpha(m,n)+t)$ and $\mathcal{O}(n^3)$ depending on which algorithm is used where n is the number of nodes, m is the number of edges in the control flow graph, and t is a parameter determined by the topology of the control flow graph [25].

aSSA form uses a subset of nodes for its representation. The representation of aSSA has a space complexity of $\mathcal{O}(n^2)$ since for each phi-node n nodes could be controlling nodes. The algorithm for computing aSSA rely on fast dominator algorithms running in linear time. Hence, aSSA has an advantage in terms of speed and space for larger programs though it abstracts away the exact conditions under which the operands of a phi-node are taken.

Information Flow. Information Flow is a notion concerning confidentiality that tracks information passage between variables or communication channels inside a program. By adopting the well-known multi-level security policies [2], a program is regarded as secure if there is no information from any variable v to any other variables that are not dominated by v's security level. Since it was first proposed as a program analysis method in [8], various work has been done to extend this notion, including by secure type systems, static program analysis, or formal verification, e.g., [22, 13]. Information flow is tracked by assignment, whereby information flows from its right-hand side variables to the left-hand side variable, and by control, whereby information flows from its predicate to the variables defined inside the control structure.

The user-input dependence analysis presented in this paper is closely related to information flow analysis. Its methodologies are analogous in respect of tracking information flow, i.e., information is propagated from user-input (as high security level) throughout a program via control- and data-dependencies. However, our control dependency is stricter than the notion of control dependency used for information flow. In our approach conditions contribute to the reachability of a phi-node only, if it is *not* regarded as controlling. Note in Figure 1(b), i_1 and j_1 are not control-dependent on p_0, although they are executed in the branch of condition p_0.

In the terminology of information flow, we consider integrity but not confidentiality, and we do not have the notion of a security policy.

Data Flow Analysis. The theory of monotone data flow analysis frameworks was established in [16]. Reps et al. [21] maps an inter-procedural data flow analysis to a reachability graph. The mapping requires an exploded control flowgraph, i.e., a graph that encodes the data flow facts and the transfer functions as a reachability graph. In this paper we compress the reachability graph by exploiting the properties of SSA form resulting in a fast algorithm. In our approach we have $\mathcal{O}(n)$ nodes in the reachability graph where n is the number of variables. In contrast, the approach of Reps et al. has $\mathcal{O}(n \times |N|)$ nodes in the reachability graph where $|N|$ is the number of basic blocks. Note that Reps' et al. work is a general framework for solving instances of separable data flow analysis problems.

Slicing [26] is a related technique to user-input dependence analysis. Most slicing approaches rely on PDGs introducing imprecision due to false data dependencies. Forward and backward slicing is concerned with computations and their immediate or intermediate dependent/depending computations. In contrast, we use a more precise representation to find dependencies on user-inputs.

6. Future Work

The efficiency of the algorithm meets the requirement of quickly analyzing large code bases in the millions of lines of source code. However, the results presented in this work are preliminary as our treatment of pointers is too conservative. We plan to use a more precise may-alias analysis as provided by LLVM. Further, to better understand precision and speed trade-offs, we would like to conduct a comparison study with the path-conditition approach of Snelting et al. [24].

7. Conclusion

In this paper we introduced a new user-input dependence analysis, which takes both data and control dependencies into account. The underlying program representation for our analysis is Static Single Assignment form, which we extend to Augmented Static Single Assignment (aSSA) form to capture control dependencies. aSSA is a simplified version of Gated Single Assignment (GSA) form and a more

compact representation than the combined Program Dependence Graph (PDG) and SSA representation. We exploit properties of aSSA form to reduce the classic Meet-Over-all-Paths solution into a simplified graph reachability problem. This reduction is novel to our knowledge and results in a fast algorithm for solving the user-input dependence analysis.

The preliminary results confirm that our approach is viable as a filtering/pre-processing phase of our bug-checking tool when run on large code bases. The analysis pin-points, in a runtime efficient manner, which statements in the code are dependent on user-inputs. If the further analysis performed by the bug-checking tool on such statements determines that a bug exists, then that bug is potentially a security vulnerability as it can be exploited externally via user input.

Acknowledgments

We would like to thank Nathan Keynes, Erica Mealy, Kirsten Winter, Eduard Mehofer, Nigel Horspool and Lian Li for their input to improve the presentation of this paper.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of the Symp. on Princ. of Prog. Lang.*, pages 1–3, January 2002.

[2] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA 01730, March 1976.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic prog. errors. *Software—Practice & Experience*, 30:775–802, 2000.

[4] T. Christiansen. Perl security. `http://www.perl.com/doc/manual/html/pod/perlsec.html` Taint module available November 1997.

[5] C. Cifuentes and B. Scholz. Parfait – designing a scalable bug checker. In *Proceedings of the ACM SIGPLAN Static Analysis Workshop*, 12 June 2008.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Program. Lang. and Syst.*, 13(4):451–490, October 1991.

[7] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of the Conf. on Prog. Lang. Design and Impl.*, June 2002.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1997.

[9] A. Deutsch. Static verification of dynamic properties. PolySpace White Paper.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *Proc. of the Symp. on Operat. Syst. Design and Impl.*, pages 23–25, Oct. 2000.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Syst.*, 9(3):319–349, Jul. 1987.

[12] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *In Proc. of the Conf. on Prog. Lang. Design and Impl.*, pages 192–203, 1999.

[13] C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *Proc. of the Int. Symp. on Secure Software Engineering*, 2006.

[14] B. Herlihy, P. Schachte, and H. Sondergaard. Un-kleene boolean equation solver. *International Journal on Foundations of Computer Science*, 18(2):227–250, 2007.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2 edition, 2000.

[16] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[18] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[19] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of the conf. on Prog. Lang. Design and Impl.*, pages 257–271, 1990.

[20] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *Proc. of the European Conf. on Object-Oriented Prog.*, pages 362–386, July 2005.

[21] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the Symp. on Princ. of Prog. Lang.*, pages 49–61, 1995.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):1–15, January 2003.

[23] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. Technical Report SMLI TR-2008-171, Sun Microsystems Laboratories, 16 Network Circle, Menlo Park, CA 94025, March 2008.

[24] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. on Softw. Eng. and Meth.*, 15(4):410–457, October 2006.

[25] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[27] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proc. of the Conf. on Prog. Lang. Design and Impl.*, pages 47–55, 1995.

[28] US-CERT. Vulnerability Note VU#881872, Sun Solaris telnet authentication bypass vulnerability. `http://www.kb.cert.org/vuls/id/881872`.

[29] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injec-tion vulnerabilities. In *Proc. of the Conf. on Prog. Lang. Design and Impl.*, 2007.

[30] M. Weiss. The transitive closure of control dependence: the iterated join. *ACM Lett. Program. Lang. Syst.*, 1(2):178–190, 1992.