User-Input Dependence Analysis via Graph Reachability

Bernhard Scholz, Chenyi Zhang,
and Cristina Cifuentes

User-Input Dependence Analysis via Graph Reachability

Bernhard Scholz, Chenyi Zhang, Cristina Cifuentes

SMLI TR-2008-171 March 2008

Abstract:

Security vulnerabilities are software bugs that are exploited by an attacker. Systems software is at high risk of exploitation: attackers commonly exploit security vulnerabilities to gain control over a system, remotely, over the internet. Bug-checking tools have been used with fair success in recent years to automatically find bugs in software. However, for finding software bugs that can cause security vulnerabilities, a bug checking tool must determine whether the software bug can be controlled by user-input.

In this paper we introduce a static program analysis for computing user-input dependencies. This analysis is used as a pre-processing filter to our static bug checking tool, currently under development, to identify bugs that can be exploited as security vulnerabilities. Runtime speed and scalability of the user-input dependence analysis is of key importance if the analysis is used for large commercial systems software.

Our user-input dependency analysis takes both data and control dependencies into account. We extend Static Single Assignment (SSA) form by augmenting phi-nodes with control dependencies of its arguments. A formal definition of user-input dependency 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 version of the analysis. The paper compares their efficiency and effectiveness for various systems codes.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
bernhard.scholz@sun.com
chenyi.zhang@sun.com
cristina.cifuentes@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

User-Input Dependence Analysis via Graph Reachability

Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes

Sun Microsystems Laboratories
Brisbane, Australia
{Bernhard.Scholz,Chenyi.Zhang,Cristina.Cifuentes}@sun.com

Abstract. Security vulnerabilities are software bugs that can be exploited by an attacker. Systems software is at high risk of exploitation: attackers commonly exploit security vulnerabilities to gain control over a system, remotely, over the internet. Bug-checking tools have been used with fair success in recent years to automatically find bugs in software. However, for finding software bugs that can cause security vulnerabilities, a bug checking tool must determine whether the software bug can be controlled by user-input.

In this paper we introduce a static program analysis for computing user-input dependencies. This analysis is used as a pre-processing filter to our static bug checking tool, currently under development, to identify bugs that can be exploited as security vulnerabilities. Runtime speed and scalability of the user-input dependence analysis is of key importance if the analysis is used for large commercial systems software.

Our user-input dependency analysis takes both data and control dependencies into account. We extend Static Single Assignment (SSA) form by augmenting phi-nodes with the control dependencies of their arguments. A formal definition of user-input dependency 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 version of the analysis. The paper compares their efficiency and effectiveness for various systems applications.

1 Introduction

Systems software is at high risk of exploitation. 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 [1] and the Sun Telnet worm [2], exploit security vulnerabilities in software and can compromise hundreds of thousands of computers in 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 [3–7]. 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 [8] 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 [9, 10]. 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, not just those paths exercised at run-time. In this paper we propose a new static program analysis technique for locating user-input dependencies in programs based on SSA form [11]. This analysis is used as a pre-processing pass to our static bug checking tool for finding the relevant statements in a program that are prone for vulnerabilities. Runtime speed and scalability of the analysis is of importance when used for large commercial systems software. The contributions of this work are as follows: (1) the solution of user-input dependency 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, and (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. Sec. 2 demonstrates our approach based on a motivating example, Sec. 3 introduces the notation used throughout the paper, Sec. 4 presents the user-input dependence analysis, and Sec. 5 describes the implementation and the experiments. In Sec. 6 we survey related work. In the last section we draw our conclusions.

2 Motivating Example

To illustrate our approach to user-input dependence analysis, we present the C-code fragment for `copy_to_utf` in Fig. 1(a), a function that copies a character-string to a Unicode Transformation Format (UTF) array. The function declares two character arrays `x` and `y` of the same fixed size (`BUFSIZ`). The variable `n` has value zero before entering the code fragment and is passed on to the 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 the 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`, character by character, with an interleaving 0, creating the UTF representation of the character.

In the example, a **buffer overflow** may occur on line 13 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 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 the `in()` function, the return value of the library function `getchar` is user-input dependent because the user provides the input. Hence, the result of the function `in()` is dependent on user-input. The buffer overflow poses a potential security vulnerability because it can be exercised via user-input. In the example, a **read outside the bounds of an array** may occur in line 14 if the index `i` is greater or equal to `BUFSIZ`—the size of array `x`. Since `i` is dependent on user-input, it too poses a potential security vulnerability.

The aSSA form of our motivating example is given in Fig. 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 Sec. 4). Note that the example makes use of `load` and `store` instructions to denote the load of a value from memory and a store of a value to memory, including all address computation that needs to be performed to reach the particular memory location of interest.

To intuitively illustrate the augmented phi-nodes, consider the assignments to $\mathtt{i}_1$ and $\mathtt{j}_1$. Both nodes are augmented phi-nodes that decide whether the variable values are taken from inside or outside the loop depending on predicate $\mathtt{p}_1$. Note that predicate $\mathtt{p}_0$ of the outer if-statement is not involved in the selection, though both statements are only executable if $\mathtt{p}_0$ holds. Similarly, the augmented phi-node $\mathtt{a}_2$ selects between the value $\mathtt{a}_0$ and $\mathtt{a}_1$ depending on predicate $\mathtt{p}_2$.

We map the user-input dependency test to a graph reachability problem in a rooted directed graph. The graph reachability problem 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

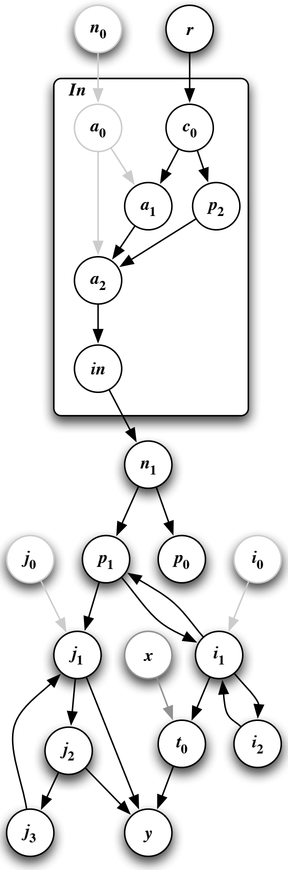
```
void copy_to_utf()
{
 int n,
     i,
     j;
 char x[BUFSIZ],
      y[BUFSIZ];
 ...
 n = in(n);
 if (n > 0) {
  j = 0;
  for (i=0;i<n;i++){
   y[j++] = x[i];
   y[j++] = 0;
  }
 }
 ...
}
int in(int a)
{
 int c = getchar();
 if (isdigit(c)) {
  a = a + c - ’0’;
 }
 return a;
}
```

(a) Input Program

```
      ...
      n_1:=in(n_0);
      p_0:=(n_1>0);
      if(¬p_0) goto ex;
      j_0:=0;
      i_0:=0;
for:  i_1:=ϕ′(i_0,i_2;p_1);
      j_1:=ϕ′(j_0,j_3;p_1);
      p_1:=(i_1<n_1);
      if(¬p_1) goto ex;
      t_0:=load x(i_1);
      store y(j_1),t_0;
      j_2:=j_1+1;
      store y(j_2),0;
      j_3:=j_2+1;
      i_2:=i_1+1;
      goto for;
ex:   …

int in(int a_0){
      c_0:=getchar();
      p_2:=isdigit(c_0);
      if(¬p_2) goto br;
      a_1:=a_0+c_0-10;
br:   a_2:=ϕ′(a_0,a_1;p_2);
      return a_2;
}
```

(b) aSSA



(c) Reachability Graph

Fig. 1. Motivating example

may control the value of the node and the value becomes user-input dependent. In Fig. 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 $\texttt{i}_2\texttt{:=i}_1\texttt{+1}$ that has a single data dependency on its right-hand side, i.e., $\texttt{i}_1$. There is an edge between $\texttt{i}_1$ and $\texttt{i}_2$. 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 $\texttt{i}_1\texttt{:=}\phi'\texttt{(i}_0\texttt{,i}_2\texttt{;p}_1\texttt{)}$ depends on $\texttt{i}_0$ and $\texttt{i}_2$ by data dependencies but also on $\texttt{p}_1$ by control dependency. Therefore, node $\texttt{i}_1$ has incoming edges from $\texttt{i}_0$, $\texttt{i}_2$ and $\texttt{p}_1$. 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 $\mathtt{n}_0$ is connected to the formal argument $\mathtt{a}_0$ and the return value $\mathtt{a}_2$ is connected to the function node `in`. Variable $\mathtt{n}_1$ 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 $\mathtt{c}_0$ directly 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 Fig. 1(c) the array indices $\mathtt{i}_1$ and $\mathtt{j}_1$ 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 Fig. 1(a).

3 Background

A flowgraph is a directed graph (N, E, s, e), where N is the set of nodes and $E \subseteq N \times N$ the set of edges. If $(u, v) \in E$, v is a successor of u, and u is a predecessor of v, we write $\mathit{preds}(v)$ for the set $\{u \in N \mid (u, v) \in E\}$. We assume that there is a distinguished start node s with in-degree zero (i.e., no predecessors) and an exit node e with out-degree zero (i.e., no successors). The source of edge (u, v) is node u, written $\mathit{src}(u, v)$. The source $\mathit{src}(E')$ of a set of edges $E' \subseteq E$ is $\{u \mid (u, v) \in E'\}$. A *path* of length k is a sequence of nodes $(u_0, \ldots, u_k)$ such that (u_i, u_{i+1}) is an edge for all $0 \leq i \leq k-1$. An *empty path* is a path of length zero. We write $\alpha : u \xrightarrow{*} v$ for a path $\alpha = (u, ..., v)$ from node u to node v. We also write $v \in (u_0, \ldots, u_k)$ if $v = u_i$ for some $i = 0 \ldots k$. The set of paths from node u to v is denoted by $\mathit{Path}(u, v)$. For path $\alpha = (u_0, \ldots, u_k)$ and $\beta = (v_0, \ldots, v_l)$, we write $\alpha \circ \beta$ for the path $(u_0, \ldots, u_k, v_1, \ldots v_l)$ if $u_k = v_0$. Given $\mathit{\Pi}$, $\mathit{\Pi}'$ as two sets of paths, we write $\mathit{\Pi} \circ \mathit{\Pi}'$ for the set $\{\alpha \circ \beta \mid \alpha \in \mathit{\Pi} \wedge \beta \in \mathit{\Pi}'\}$.

A node u *dominates* a node v, written as $u\ \mathit{dom}\ v$, if all paths from the start node s to v include u. The dominators $\mathit{dom}(u)$ of node u is the set of nodes that dominate u. A node u *strictly dominates* a node v, written $u\ \mathit{sdom}\, v$, if u dominates v and $u \neq v$, and we write $\mathit{sdom}(u)$ for the set of nodes that strictly dominate u. The *immediate dominator* $\mathit{idom}(u)$ of a node u is the unique node that strictly dominates u but does not strictly dominate any other strict dominator of u. The *dominance frontier* $\mathit{DF}(u)$ of node u is a set of nodes $N' \subseteq N$ such that for all $v \in N'$, u dominates a predecessor of v but does not strictly dominate v; i.e., $\mathit{DF}(u) = \{v \mid \exists w \in \mathit{preds}(v) : u\ \mathit{dom}\ w \wedge \neg(u\ \mathit{sdom}\, v)\}$. The dominance frontier of a set of nodes $N' \subseteq N$ is defined as $\mathit{DF}(N') = \bigcup_{u \in N'} \mathit{DF}(u)$, and the iterated dominance frontier is defined as $\mathit{IDF}(N') = \bigcup_{i \in \mathbb{N}} \mathit{DF}^i(N')$, where $\mathit{DF}^0(N') = \mathit{DF}(N')$ and $\mathit{DF}^{i+1}(N') = \mathit{DF}(\mathit{DF}^i(N'))$ for $i \in \mathbb{N}$. We define the reverse graph of a flowgraph where the edges are reversed, and the start and the end nodes swapped. A node u *post-dominates* v, written as $u\,\mathit{pdom}\, v$, if u dominates v in the reversed flowgraph. The *post iterated dominance frontier* $\mathit{PIDF}(N')$ is the iterated dominance frontier of $N' \subseteq N$ in the reversed flowgraph [12].

4 User-Input Dependence Analysis

4.1 Augmenting Phi-Nodes with Control Dependencies

SSA form provides an efficient representation of the def-use relation on data dependencies. A program represented in SSA form does not have any false data dependencies [11]. However, SSA form does not expose any control dependencies [12, 13]. To represent control dependencies, we extend SSA phi-nodes with control dependencies,

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

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 [14, 15].[1]

We define the set P_x on arbitrary reducible flow graphs. The following definitions are required for our definition of control dependencies. Let $x = \phi(y_1, \ldots, y_k)$, we say x selects $y_i \in Y_x$ in a path $\alpha = (s, \ldots, x)$ if $y_i \in \alpha$, and for all $y_j \in \alpha$ with $i \neq j$, we have $y_j \in \alpha$ implies $y_j \leq_\alpha y_i$; where $v \leq_\alpha v'$ if the rightmost occurrence of v has a smaller index than the rightmost occurrence of v' in α. A path $(u_1, u_2, \ldots u_n)$ is *forward* if for all u_i, u_j with $1 \leq i < j \leq n$, we have $\neg(u_j \mathit{dom}\, u_i)$.

There are two different control dependency relations that we need to explain separately, as follows. If p is a controlling node that splits the flow into two different nodes that define y_1 and y_2, which later merge at $x = \phi(y_1, y_2)$, we say p controls x because the value of p decides whether the y_1 or y_2 is to be taken at x. However, if it is the case that $x\, \mathit{dom}\, p$, then it is possible to allow p to join the selection by the existence of a path from p to x, and consequently p and x are included in a loop. We distinguish these two cases by using the term forward-control dependency and loop-control dependency respectively.

Definition 1. *A controlling node p with successors q and r forward-controls a phi-node x, if for all path $\gamma = (s, \ldots, p)$, there exist forward paths $\alpha = (q, \ldots, x)$ and $\beta = (r, \ldots, x)$, such that $\alpha \cap \beta = \{x\}$, and x selects y_i from $\gamma \cdot \alpha$ and y_j from $\gamma \cdot \beta$ such that $i \neq j$.*

Write $FC(x)$ for the set of forward-controlling nodes of x. The intuition behind this is that a controlling node p affects the value selection of x if and only if both branches of p may reach x, and with different selection outcomes. If a phi-node x is unreachable by either branch of a controlling node p by a forward path, p is not regarded as forward controlling x. A typical forward-control dependency has been sketched in Fig 2 where x_2 is forward-control dependent on p, but not forward-control dependent on q.

We adapt the classical loop definition for reducible graphs. An edge (u, v) is a *back edge* if $v\, \mathit{dom}\, u$. Given a node u, define the set $B_u = \{v \mid (v, u) \text{ is a back edge}\}$.

[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 together with switches.

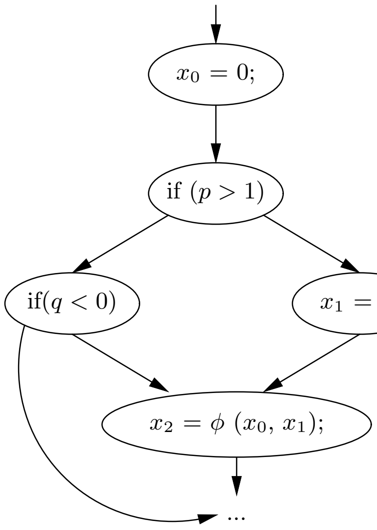


Fig. 2. An example where p forward-controls x but q does not.

If $B_u \neq \emptyset$ we define $L_u = \{w \mid \exists \alpha = (w, \ldots, v) : v \in B_u \wedge u \notin \alpha\}$, and u is the loop header of L_u.

Definition 2. *A controlling node p loop-controls a phi-node x, if there is a loop L containing p and x, a definition node $y_i \in Y_x \cap L$ and another definition node $y_j \notin L$, $x\ dom\, p$, and there is a successor q of p with $q \notin L$.*

Write $LC(x)$ for the set of loop-controlling nodes of x. Since p partially decides the number of iterations in the loop, it may induce the program execution to x via a path in which x selects the definition y with a different value for each iteration. In the example which is part of a reducible graph shown in Fig 3, the controlling node q loop-controls x_1, and the controlling node p forward-controls x_3. Finally, we combine the above two definitions as the set of controlling nodes P_x.

Definition 3.

$$P_x = FC(x) \cup LC(x)$$

The definitions for forward- and loop-control dependencies are purely flow graph-oriented. However, it is difficult to identify the set of controlling nodes given a phi-node x. Therefore, we apply a conservative approach to calculate P_x, which simply takes the post iterated dominance frontier of the arguments of a phi-node x less those that are not dominated by x's immediate dominator. The correctness of this approximation is guaranteed by the following proposition, and for structured control flow the approximant coincides with the predicate set of an augmented phi-node. The proof is presented in the appendix.

Proposition 1. *For each phi-node x, we have*

$$P_x = FC(x) \cup LC(x) \subseteq PIDF(Y_x) \cap dom^{-1}(\, idom\,(x))$$

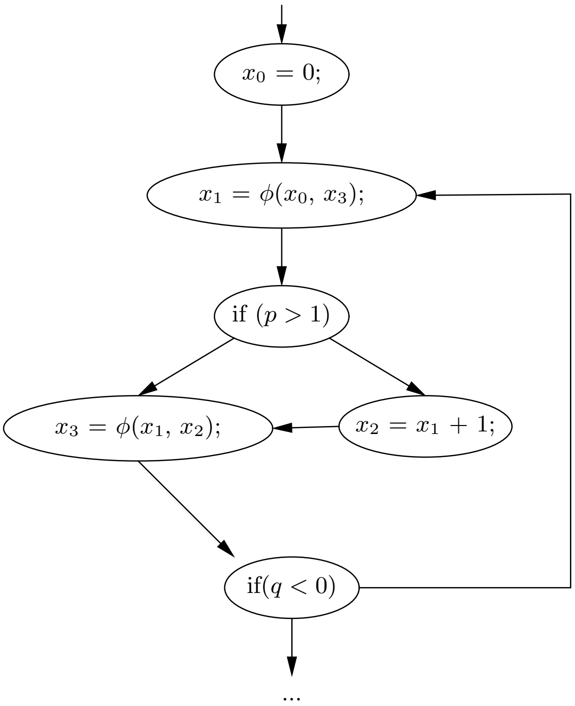


Fig. 3. An example where q loop-controls x_1

4.2 Definition of Tainted Values

The user input dependence analysis obtains the information whether a variable in a program is potentially tainted or strictly untainted. 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 $\vartriangle$ representing the untainted value. The meet operation is defined by:

$$a \sqcap b = \begin{cases} \vartriangle, & \text{if } a = \vartriangle \text{ and } b = \vartriangle, \\ \blacktriangle, & \text{otherwise.} \end{cases} \tag{2}$$

Semi-lattice $(\mathcal{L}, \sqcap)$ is isomorphic to the boolean semi-lattice $(\mathbb{B}, \vee)$ assuming element $\blacktriangle$ is 1 and element $\vartriangle$ 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 $\vartriangle$ 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 $\vartriangle$ is the top and $\blacktriangle$ is the bottom element.

We define an information lattice $(\mathcal{L}^n, \sqcap, \blacktriangle^n, \vartriangle^n)$ where n is the number of variables. An element $\boldsymbol{c} \in \mathcal{L}^n$ 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 ($i_x = 1 \dots n$) to a variable $x \in \mathit{Var}$ that denotes the position of x in a vector of size n. The result of meet operation $\boldsymbol{a} \sqcap \boldsymbol{b}$ is vector $\boldsymbol{c}$ with element $c_i = a_i \sqcap b_i$ for all $i = 1 \dots n$. The top and bottom elements are $\vartriangle^n = \langle \vartriangle, \dots, \vartriangle \rangle$ and $\blacktriangle^n = \langle \blacktriangle, \dots, \blacktriangle \rangle$, respectively. For the sake of readability, we use the notation $\boldsymbol{c}(x)$ for element $\boldsymbol{c}_{i_x}$, and $\boldsymbol{c}_{[x \leftarrow a]}$ is a configuration identical to $\boldsymbol{c}$ except for $\boldsymbol{c}_{[x \leftarrow a]}(x) = a$.

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

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

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 $\pi = (u_1, \dots, u_k)$. 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., $M[\![\texttt{nop}]\!](\boldsymbol{c}) = \boldsymbol{c}$.
- **Read Operation:** taints variable x, i.e., $M[\![x := \texttt{read}]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \blacktriangle]}$.
- **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., $M[\![x := \mathit{op}(y_1, \dots, y_k)]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \sqcap_{1 \le i \le k} \boldsymbol{c}(y_i)]}$; if there are no variables on the right-hand side, then $M[\![x := \mathit{op}()]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \vartriangle]}$.
- **Augmented Phi-Node:** If one of the arguments or one of the predicates is tainted, then the result will be tainted, i.e., $M[\![x := \phi'(y_1, \dots, y_k; p_1, \dots, p_l)]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \sqcap_{1 \le i \le k} \boldsymbol{c}(y_i) \sqcap \sqcap_{1 \le i \le l} \boldsymbol{c}(p_i)]}$, where predicate p_i ($i = 1 \dots l$) 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 4. *Variable $v \in Var$ in aSSA form is untainted, if $\mathit{mop}(v)(v)$ is $\vartriangle$.*

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

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

Note that variable $\boldsymbol{z}_u$ 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 $\mathit{mfp}(F)$. We write $\mathit{mfp}(u)$ for the maximum fixed-point of F on node u.

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

Proof. It is easy to see that $\mathcal{L}^n$ is a semi-lattice and has finite height. The associated function space of the data flow analysis framework is distributive, i.e., $\forall \boldsymbol{c}_1, \boldsymbol{c}_2 \in \mathcal{L}^n$: $\forall u \in N : M(u)(\boldsymbol{c}_1 \sqcap \boldsymbol{c}_2) = M(u)(\boldsymbol{c}_1) \sqcap M(u)(\boldsymbol{c}_2)$, and closed under composition [16].

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., $F \circ \cdots \circ F(\triangle^{|N| \times n})$.

4.3 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 $x \in \mathit{Var}$, there is a unique $u \in N$ such that x is defined at u. We use $u \in N$ as a synonym for $x \in \mathit{Var}$ if u defines x, and vice versa.
2. Every node $x := \mathit{op}(y_1, \ldots, y_k)$ 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 $y \in Y_x$, i.e., there is a path $\alpha = (y \xrightarrow{*} x)$ 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 $x := \mathit{op}(y_1, \ldots, y_k)$, each y_i is equal to the value $\mathit{mop}(x)(y_i)$ 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 $p \in P_x$.

Lemma 1. *Given nodes $u, v \in N$, and $u \xrightarrow{*} v$,*

$$\left[\bigsqcap_{\{\pi \in \mathit{Path}(u,v) | y \notin \pi\}} M(\pi)(c) \right] (y) = c(y). \tag{5}$$

Proof. By induction on the length of arbitrary paths that do not include y. □

Lemma 2. *For all $y \in Var$ and $u \in N$, $(y \xrightarrow{*} u)$ implies $\mathit{mop}(u)(y) = \mathit{mop}(y)(y)$.*

Proof. If y is equal to u, then Lemma is immediate. Otherwise, startnode s reaches y, and y reaches u. Therefore, there exists a path from s to u via y.

$$\begin{aligned}
\mathit{mop}(u)(y) &= \left[\bigsqcap_{\pi\in \mathit{Path}(s,u)} M(\pi)(\triangle^n)\right](y) && (6)\\
&= \left[\bigsqcap_{\{\pi\in \mathit{Path}(s,u)\mid y\in\pi\}} M(\pi)(\triangle^n)\right](y) \sqcap \left[\bigsqcap_{\{\pi\in \mathit{Path}(s,u)\mid y\notin\pi\}} M(\pi)(\triangle^n)\right](y) && (7)\\
&= \left[\bigsqcap_{\pi\in \mathit{Path}(s,y)\circ \mathit{Path}^\sharp(y,u)} M(\pi)(\triangle^n)\right](y) \sqcap [\triangle^n(y)] && (8)\\
&= \left[\bigsqcap_{\pi\in \mathit{Path}(s,y)\circ \mathit{Path}^\sharp(y,u)} M(\pi)(\triangle^n)\right](y) && (9)\\
&= \left[\bigsqcap_{\pi\in \mathit{Path}^\sharp(y,u)} M(\pi)\left(\bigsqcap_{\pi'\in \mathit{Path}(s,y)} M(\pi')(\triangle^n)\right)\right](y) && (10)\\
&= \left[\bigsqcap_{\pi'\in \mathit{Path}(s,y)} M(\pi')(\triangle^n)\right](y) && (11)\\
&= \mathit{mop}(y)(y) && (12)
\end{aligned}$$

where $\mathit{Path}(u,v)^\sharp$ is the set of paths from u to v with a single occurrence of u. From step (7) to (8) and step (10) to (11) we apply Lemma 1. Note that if there is no path in set $\{\pi \in \mathit{Path}(s,u) \mid y \notin \pi\}$, then the meet operation in (7) reduces to $\triangle^n$. $\square$

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

$$\widehat{z}_x = \begin{cases} \blacktriangle, & \text{if } x := \texttt{read},\\ \triangle, & \text{if } x := \mathit{op}(),\\ \bigsqcap_{1\le i\le k} \widehat{z}_{y_i}, & \text{if } x := \mathit{op}(y_1,\dots,y_k),\\ \bigsqcap_{1\le i\le k} \widehat{z}_{y_i} \sqcap \bigsqcap_{1\le j\le l} \widehat{z}_{p_j}, & \text{if } x := \phi'(y_1,\dots,y_k;p_1,\dots,p_l). \end{cases} \quad (13)$$

Let $\widehat{\boldsymbol{z}} \in \mathcal{L}^n$ 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 $\mathit{mfp}(\widehat{F})$. We write $\widehat{\mathit{mfp}}(x)$ for the maximal solution of $\widehat{F}$ on variable x.

Lemma 3. *Vector $\boldsymbol{z} = \langle \mathit{mop}(x_1)(x_1),\dots,\mathit{mop}(x_n)(x_n)\rangle$ is a fixed point of $\widehat{F}$.*

Proof. We show for each element x in vector $\boldsymbol{z}$ that $\widehat{F}(\boldsymbol{z}) = \boldsymbol{z}$. For every x we have four cases:

1. **Read operation** ($x := \texttt{read}$):

$$mop(x)(x) = \left[\bigsqcap_{\pi \in Path(s,u)} M(\pi)(\triangle^n) \right] (x) \tag{14}$$

$$= \left[M[\![x := \texttt{read}]\!] \left(\bigsqcap_{u \in preds(x)} \bigsqcap_{\pi \in Path(s,u)} M(\pi)(\triangle^n) \right) \right] (x) \tag{15}$$

$$= \blacktriangle. \tag{16}$$

2. **Assignment without operands** ($x := op()$): Similar to read operation as above, i.e., $mop(x)(x) = \triangle$.
3. **Assignment with operands** ($x := op(y_1, \dots, y_k)$):

$$mop(x)(x) = \left[\bigsqcap_{\pi \in Path(s,u)} M(\pi)(\triangle^n) \right] (x) \tag{17}$$

$$= \left[M[\![x := op(y_1, \dots, y_k)]\!] \left(\bigsqcap_{u \in preds(x)} \bigsqcap_{\pi \in Path(s,u)} M(\pi)(\triangle^n) \right) \right] (x) \tag{18}$$

$$= \left[M[\![x := op(y_1, \dots, y_k)]\!] \left(\bigsqcap_{u \in preds(x)} mop(u) \right) \right] (x) \tag{19}$$

$$= \bigsqcap_{1 \le i \le k} \left(\bigsqcap_{u \in preds(x)} mop(u) \right) (y_i) \tag{20}$$

$$= \bigsqcap_{1 \le i \le k} mop(y_i)(y_i) \tag{21}$$

From step (20) to step (21), we have two conditions for predecessor u for a given y_i: (1) Node u is reachable from y_i. Therefore, $mop(u)(y_i)$ is equal to $mop(y_i)(y_i)$ by Lemma 2. (2) Node u is not reachable from y_i. Therefore, $mop(u)(y_i) = \triangle$ as a consequence of Lemma 1. There is at least one predecessor u which is reachable from y_i, since x would not be reachable from y_i otherwise.
4. **Augmented phi-node** ($x := \phi'(y_1, \dots, y_k; p_1, \dots, p_l)$): Similar to assignment with operands as above. All definitions and predicates reach node x, hence, $mop(x)(x) = \bigsqcap_{1 \le i \le k} mop(y_i)(y_i) \sqcap \bigsqcap_{1 \le j \le l} mop(p_j)(p_j)$.

As a consequence of Lemma 3, for all $x \in Var$, $mop(x)(x) \sqsubseteq \widehat{mfp}(u)$. To show the inverse, we define functions $F^i = \underbrace{F \circ \dots F}_{i}(\triangle^{|N| \times n})$, and $\widehat{F}^i = \underbrace{\widehat{F} \circ \dots \circ \widehat{F}}_{i}(\triangle^n)$.

Lemma 4. *For all $x \in Var$ and for all $i \ge 0$, $\widehat{F}^i(x) \sqsubseteq F^i(x)(x)$.*

Proof. By a straightforward induction we show that $F^i(u)(y) \sqsupseteq F^i(y)(y)$ for all $u \in N$, for all $y \in \mathit{Var}$, and $i \geq 0$. We prove the lemma by induction. The base case is immediate. For the induction step we have four different cases for $x \in \mathit{Var}$.

1. **Read operation** ($x :=$ `read`) : $\widehat{F}^i(x) = F^i(x)(x) = \blacktriangle$ for all $i > 0$.
2. **Assignment without operands** ($x := \mathit{op}$): Similar as above.
3. **Assignment with operands** ($x := \mathit{op}(y_1, \ldots, y_k)$):

$$
\begin{aligned}
F^{i+1}(x)(x) &= \left[M(x) \left(\bigsqcap_{u \in \mathit{preds}(x)} F^i(u) \right) \right](x) && (22)\\
&= \bigsqcap_{y \in Y_x} \left[\left(\bigsqcap_{u \in \mathit{preds}(x)} F^i(u) \right)(y) \right] && (23)\\
&= \bigsqcap_{y \in Y_x} \bigsqcap_{u \in \mathit{preds}(x)} F^i(u)(y) && (24)\\
&\sqsupseteq \bigsqcap_{y \in Y_x} \bigsqcap_{u \in \mathit{preds}(x)} F^i(y)(y) && (25)\\
&= \bigsqcap_{y \in Y_x} F^i(y)(y) && (26)\\
&\sqsupseteq \bigsqcap_{y \in Y_x} \widehat{F}^i(y) && (27)\\
&= \widehat{F}^{i+1}(x) && (28)
\end{aligned}
$$

   From step (24) to step (25) we apply $F^i(u)(y) \sqsupseteq F^i(y)(y)$, and the induction hypothesis is used from step (26) to step (27).
4. **Augmented phi-node** ($x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l)$) : Similar as above.

After a finite number of steps both function converge to a fixed point because the lattice has finite height. Hence, $\widehat{\mathit{mfp}}(x) \sqsubseteq \mathit{mfp}(x)(x)$, for all $x \in \mathit{Var}$. Since, $\mathit{mfp}(u)$ coincides with $\mathit{mop}(u)$ (cf. Theorem 1), $\widehat{\mathit{mfp}}(x) \sqsubseteq \mathit{mop}(x)(x)$. Combining this result with Lemma 3, the following theorem is implied.

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

4.4 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 $a \leq b$ if $a \vee b = a$. In this partial order 0 is the top element and 1 is the bottom element. The partial order $\leq$ is further extended to vectors, i.e., $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{B}^n$, $\boldsymbol{a} \leq \boldsymbol{b}$, if $a_i \leq b_i$, for all $i = 1, \ldots, n$.

Definition 5. *Given $A \in \mathbb{B}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{B}^n$ in the boolean lattice $(\mathbb{B}, \vee, \wedge, 1, 0)$, define $\boldsymbol{x} \in \mathbb{B}^n$ the maximal solution of the boolean equation system*

$$x_i = (\bigvee_{j=1}^{n} a_{ij} \wedge x_j) \vee b_i, \quad \textit{for } i = 1, \ldots, n. \tag{29}$$

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

We associate to Eqs. (29) 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 $\boldsymbol{x}$, an element x_i has value 1 iff $v_i \in R$.*

Proof. We restate the theorem as an element x_i in the maximal solution has value 0 if $v_i \notin R$, and value 1 if $v_i \in R$. Hence, we define a vector with elements x_i such that

$$x_i = \begin{cases} 1, \text{ if } v_i \in R \\ 0, \text{ otherwise} \end{cases}, \quad \text{for } i = 1, \ldots, n. \tag{30}$$

for which we show that $\boldsymbol{x}$ is a fixed point. We make following observations showing relations between the reachability graph and the linear boolean equation system:

(O1) If $v_i \notin R$, then $b_i = 0$.
(O2) If $v_i \notin R$, then for all j $(1 \leq j \leq n)$ with $a_{ij} = 1$, $v_j \notin R$.
(O3) If $v_i \in R$ and $b_i = 0$, then there exists a node $v_j \in R$ and $a_{ij} = 1$.

Let $\boldsymbol{z} = F(\boldsymbol{x})$. For $i = 1, \ldots, n$, we have four cases,

$$z_i = \begin{cases} 1, & \text{if } v_i \in R \wedge b_i = 1 & \text{(Case 1)} \\ \bigvee_j a_{ij} \wedge x_j, & \text{if } v_i \in R \wedge b_i = 0 & \text{(Case 2)} \\ 0, & \text{if } v_i \notin R \wedge \forall j : a_{ij} = 0 & \text{(Case 3)} \\ \bigvee_{v_j \notin R} a_{ij} \wedge x_j, & \text{otherwise} & \text{(Case 4)} \end{cases} \tag{31}$$

Case 1: Immediate. Case 2: By Observation (O3) there exists $v_j \in R$ and $a_{ij} = 1$. Therefore, x_j reduces equation to 1. Case 3: By Observation (O1), $b_i = 0$ and all a_{ij} are 0 reducing Equation (29) to 0. Case 4 (if $v_i \notin R \wedge \exists a_{ij} = 1$) : By Observation (O1), $b_i = 0$. By Observation (O2) for all $a_{ij} = 1$, $x_j = 0$ that reduces the equation to 0. Hence, $\boldsymbol{z} = \boldsymbol{x}$.

In the sequel, we show that $\boldsymbol{x}$ is the maximum fixed point. Suppose there is a fixed point $\boldsymbol{z} \nleq \boldsymbol{x}$, then there exists an index i such that $z_i > x_i$. This is only the case if z_i is 0 and x_i is 1, and v_i is reachable. Therefore, node $v_i \in R$, then there exists a simple path $\pi = (r, u_{j_1}, \ldots, u_{j_k})$ with $u_{j_k} = v_i$. Given vector $\boldsymbol{z}$ as a solution of equation (29), element z_{j_1} in vector $\boldsymbol{z}$ associated to node u_{j_1} has an edge (r, u_{j_1}) and therefore, b_{j_1} holds. If b_{j_1} holds, then z_{j_1} reduces to 1. We show that z_i is 1. For an edge $(u_{j_l}, u_{i_{l+1}}) \in \mathit{Arc}$ in π, $a_{j_{l+1} j_l}$ is 1 in equation $z_{j_{l+1}}$. Hence, Equation $z_i = z_{j_k} = \cdots \vee (a_{j_{l+1} j_l} \wedge z_{j_l}) \vee \ldots$ reduces to 1, contradicting our assumption. □

The next theorem connects the solution $\boldsymbol{x}$ of the linear boolean equation system with the MOP solution, where we assume there is an one-to-one mapping from each variable in the compressed equation system of Eqs. (13) 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 $\boldsymbol{x}$ of Eqs.* (29)*, we have for all $i = 1, \ldots, n$,*

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

Proof. The boolean lattice and the taint lattice are isomorphic. The partial orders $\sqsubseteq$ and $\leq$ are isomorphic. By simple algebraic transformation of the linear boolean equation system, the equivalence is established. □

4.5 Inter-procedure 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. Sec. 5). The call-insensitive analysis maps the whole program to a single reachability graph using the mapping as sketched in the motivating example in Fig. 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 $\mathit{mop}(u)(f)$ reflects whether the return value of f is tainted at node u. The transfer function of a call-site is $M[\![x := \texttt{call}\, f\,(y_1, \ldots, y_k)]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \boldsymbol{c}(f), a_1 \leftarrow \boldsymbol{c}(y_1), \ldots, a_k \leftarrow \boldsymbol{c}(y_k)]}$ 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.). For the first case, we use the connection scheme as used for the insensitive-analysis. In the second case, we wire the dependencies as given in the summary function. For external functions, we rely on specifications as shown in Fig. 5. After constructing the reachability graph we probe 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. 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.

For an array a we introduce a new variable in Var. The meaning of $\mathit{mop}(u)(a)$ reflects whether the contents of a are 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 $M[\![x := \texttt{load}\, a(y)]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \boldsymbol{c}(a) \sqcap \boldsymbol{c}(y)]}$, i.e., the result of 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 $M[\![\texttt{store}\, a(y), x]\!](\boldsymbol{c}) = \boldsymbol{c}_{[a \leftarrow \boldsymbol{c}(x) \sqcap \boldsymbol{c}(y)]}$, 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., $M[\![x := \texttt{load}\, g]\!](\boldsymbol{c}) = \boldsymbol{c}_{[x \leftarrow \boldsymbol{c}(g)]}$ and $M[\![\texttt{store}\, g, x]\!](\boldsymbol{c}) = \boldsymbol{c}_{[g \leftarrow \boldsymbol{c}(x)]}$.

For pointers we encode a simple may-alias 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., "$\texttt{store}\ p, x$" adds an edge from x to p, and "$x := \texttt{load}\ p$" 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. The Figure 4 shows the translation of LLVM pointer primitives to the reachability graph. The LLVM framework is used for the experimental results demonstrated in the next section of this paper.

5 Implementation and Experiments

We have implemented the call-insensitive and 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 has been designed for a virtual architecture that avoids machine specific constraints, and its instruction set is strictly typed. Every value or memory location has an associated type and all instructions obey strict type rules. LLVM code is represented in SSA form. LLVM provides support for alias analysis and assumes all clients to be flow-insensitive. Implementations of Andersen's and Steensgaard's interprocedural alias analyses are in place. Function

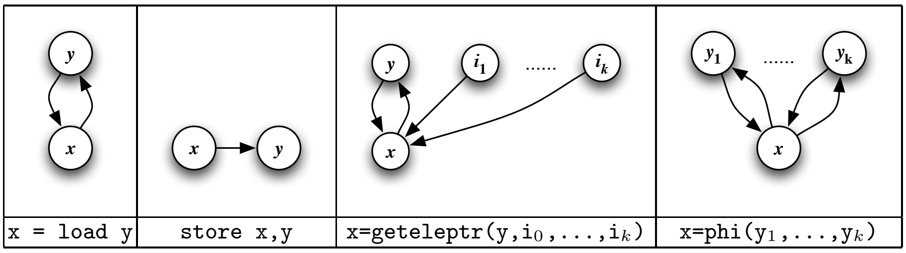


Fig. 4. Reachabilty Graph Mapping of Pointer Operations

```
_input_ stdin;    /* user-input dependent vars or args/results */
_input_ int scanf(const char *, _input_ ...);
_input_ int getchar(void);
_input_ char *gets(_input_ char *);
int main(_input_ int argc, _input_ char *argv);

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

Fig. 5. Excerpt of configuration file

alias analysis is not provided by LLVM; however, we implemented a may-function alias analysis to better support the accuracy of our user-dependency analysis. The user-input dependence analysis is a fundamental component of our security bug checking tool for systems code that is built upon the LLVM framework, with its implementation currently underway. In our bug checking tool, the inter-procedural user-input dependence analysis is used as a pre-processing pass to identify potential security vulnerabilities. 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. 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. The user-input dependence analysis not only identifies potential security vulnerabilities, but also reduces the problem size, e.g., not all array accesses need to be checked only those that are dependent on user input values.

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 Fig. 5. 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`)

	Problem Size		Dependence		Array Access							
Progs	#loc	#inst	%uii$_i$	%uii$_s$	#cr	#cw	#ncr	#ncw	%uir$_i$	%uir$_s$	%uiw$_i$	%uiw$_s$
sendmail	179753	169166	75.5	66.5	15925	7300	4301	1152	92.9	90.3	81.8	77.1
httpd	103066	164162	85.1	77.7	11849	6610	3113	690	91.7	90.8	91.7	90.6
perlbmk	85464	176866	75.8	73.4	19333	9139	1823	1590	98.2	97.9	94.9	94.6
vortex	67220	65685	73.5	69.2	4512	2473	1106	398	93.9	90.9	95.2	95.0
pppd	27048	32540	60.7	42.1	1899	1409	1380	731	41.4	28.8	31.6	18.9
sshd	20729	18489	72.4	64.7	1644	736	273	123	83.2	63.7	74.8	54.5
mailx	14609	25717	72.8	56.5	880	713	843	254	91.9	83.0	83.5	75.6
zoneadmd	7485	7835	69.0	67.5	239	247	163	28	95.1	95.1	85.7	85.7
mail	6934	7286	60.0	52.0	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, *#fn* is the number of functions, *#glob* is the number of global variables, *#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.

that are controlled by the user. Therefore, in the declaration, the qualifier `_input_` is added. Summary functions 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 analysis, we use benchmarks including programs from the OpenSolaris$^{\text{TM}}$ operating system, the Apache httpd server (v2.2.6), and programs from SPEC CINT2000. Tab. 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 Tab. 1). For array accesses, the number of user-input dependent accesses is small for both analysis. Constant array accesses cannot ever be dependent on user-input data, 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 Tab. 2, the insensitive analysis is linear on the number of instructions in the

[2] A portion of standard C library is not fully specified in the configuration file and worst-case assumption were made. With a fully specified library, we expect smaller percentages of user-input dependent instructions.

	Problem Size			Insensitive		Sensitive			Runtime	
Progs	#inst	#fn	#glob	#nod$_i$	#edg$_i$	#scc	#nod$_s$	#edg$_s$	τ_i	τ_s
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.

intermediate representation (IR). 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 Tab. 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.

6 Related Work

User-Input Dependence Analysis. Static taint analysis [18–20] and user-input dependence analysis [9, 10] 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. There are two mechanisms used for sanity checks: (1) empirical assumptions and (2) user annotations. For example, in some static taint analysis approaches it is assumed that data after a procedure call is sanitized and therefore untainted. This assumption may introduce false negatives. User annotations [18] overcome this issue, however, the programmer has to provide "correct" annotations, which are expensive in terms of manpower in large systems. It is estimated [21], that approximately one hour is required to annotate 300-600 lines of Java code. Hence, adding annotations to large legacy systems written in C (e.g. the Solaris$^{\text{TM}}$ operating system has 20 millions of lines of code) is not viable.

Dynamic taint analysis [22–24] is related to testing and considers a single execution trace of a program. However, since dynamic analysis alone is inherently unable to precisely apply control-dependencies, all these dynamic frameworks need static means to sketch control-dependency, such as post-dominator tree applied in [22]. Since we need a user-input dependence analysis to serve as a pre-processing pass, it is not viable to apply dynamic analysis to generate such a set of variables which are potentially vulnerable to user-input by exhausting all possible paths.

Other static approaches for taint analysis or user-input dependence analysis include type systems [20], data flow analysis [19], and Program Dependence Graph (PDG) [9, 10] 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. However, since PDG does not make the definition-use relation explicit, false data-dependencies still exist. Program Dependence Web (PDW) [15, 14] achieves both control and data-dependencies, but its construction and representation is more elaborate. Programs in PDW are represented in 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. Phi-nodes in Augmented Static Single Assignment (aSSA) form are abstractions of gating functions, resulting in a *simpler and less expensive* representation. In general our aSSA-based approach is *scalable* because it is reduced to a reachability problem exhibiting a linear complexity, so that large commercial systems software can be handled. It is *precise* because it takes both def-use and control dependencies into account, in an inter-procedural way. In our work, both call-sensitive and call-insensitive approaches have been studied, and the results have been analyzed in terms of a trade-off between speed and precision.

Taint Analysis and Information Flow. Information Flow is a notion concerned with confidentiality that tracks information passage between variables or communication channels inside a program. By adopting the famous multi-level security policies [25], 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 [26], various work has been done to extend this notion, including by secure type systems, static program analysis, or formal verification. Examples include [27, 9]. In these approaches information flow is tracked by assignment, where the flow is formed from the right-hand side of the assignment to the left-hand and from predicates of control structures (e.g., if, while, etc.) to the variables defined inside the control structures. The user-input dependence analysis presented in this paper is closely related to Information Flow Analysis, in that its methodologies are analogous to the ways 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 more strict than what has been applied on information flow analysis, where a condition only contributing to the reachability of a phi-node x is *not* regarded as controlling x. (Note in Fig 1(b), i_1 and j_1 is not control-dependent on p_0, although they are enclosed in a big branch following p_0.) Besides, our security concern is more safety-oriented than information-flow-oriented, as we are only interested in problems such as "whether an array access has its index *value* dependent on user-input". We do not take into account the notion of security policy, either.

Data Flow Analysis. The theory of monotone data flow analysis frameworks was established in [16]. Reps et al. [28] 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.

7 Conclusion

In this paper we introduced a new user-input dependence analysis for programs, 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. 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 experiment result confirms that our approach is viable for large-scale bug checking.

Acknowledgements

We would like to thank Jan-Willem Maessen and Philip Yelland for their helpful comments. Particular thanks to Kirsten Winter for carefully reading the manuscript and Nathan Keynes for debugging of the final algorithms. We also received input from Nigel Horspool and Eduard Mehofer. We would like to thank Wei-ying Ho for proof-reading the manuscript.

References

1. Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the Slammer Worm. IEEE Security and Privacy **1**(4) (2003) 33–39
2. US-CERT: Vulnerability Note VU#881872, Sun Solaris telnet authentication bypass vulnerability. http://www.kb.cert.org/vuls/id/881872
3. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic prog. errors. Software—Practice & Experience **30** (2000) 775–802
4. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific programmer-written compiler extensions. In: Proc. of the Symp. on Operat. Syst. Design and Impl. (October 2000) 23–25
5. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. of the Symp. on Princ. of Prog. Lang. (January 2002) 1–3
6. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proc. of the Conf. on Prog. Lang. Design and Impl. (June 2002)
7. Deutsch, A.: Static verification of dynamic properties. PolySpace White Paper
8. Christiansen, T.: Perl security. http://www.perl.com/doc/manual/html/pod/perlsec.html Taint module available November 1997
9. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: Proc. of the Int. Symp. on Secure Software Engineering. (2006)
10. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Trans. on Softw. Eng. and Meth. **15**(4) (October 2006) 410–457
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. on Program. Lang. and Syst. **13**(4) (October 1991) 451–490
12. Weiss, M.: The transitive closure of control dependence: the iterated join. ACM Lett. Program. Lang. Syst. **1**(2) (1992) 178–190
13. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3) (1987) 319–349
14. Tu, P., Padua, D.: Efficient building and placing of gating functions. In: Proc. of the Conf. on Prog. Lang. Design and Impl. (1995) 47–55
15. Ottenstein, K.J., Ballance, R.A., MacCabe, A.B.: 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. (1990) 257–271
16. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. J. ACM **23**(1) (1976) 158–171
17. Lattner, C., Adve, V.: 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. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injec-tion vulnerabilities. In: Proc. of the Conf. on Prog. Lang. Design and Impl. (2007)
19. Pistoia, M., Flynn, R.J., Koved, L., Sreedhar, V.C.: Interprocedural analysis for privileged code placement and tainted variable detection. In: Proc. of the European Conf. on Object-Oriented Prog. (July 2005) 362–386
20. Foster, J., Fahndrich, M., Aiken, A.: A theory of type qualifiers. In: In Proc. of the Conf. on Prog. Lang. Design and Impl. (1999) 192–203

21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proc. of the Conf. on Prog. Lang. Design and Impl. (2002) 234–245
22. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proc. of the Symp. on Software Testing and Analysis. (2007) 196–206
23. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: Proc. of the Conf. on USENIX Security Symp. (2006) 9–9
24. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the Network and Distributed System Security Symp. (2005)
25. Bell, D.E., LaPadula, L.J.: Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA 01730 (March 1976)
26. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**(7) (July 1997) 504–513
27. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1) (January 2003) 1–15
28. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the Symp. on Princ. of Prog. Lang. (1995) 49–61

A Appendix

Proposition 1. Given $x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l)$, we have

$$P_x \subseteq \mathit{PIDF}(Y_x) \cap \ \mathit{dom}^{-1}(\mathit{idom}(x))$$

Proof.

- We first show $FC(x) \subseteq \mathit{PIDF}(Y_x) \cap \ \mathit{dom}^{-1}(\mathit{idom}(x))$, let $p \in FC(x)$ with successors q and r, then there exists $\alpha = (q, \ldots, x)$, $\beta = (r, \ldots, x)$ and $\gamma = (s, \ldots, p)$ such that $\gamma \cdot \alpha$ selects y_i and $\gamma \cdot \beta$ selects y_j for some $y_i, y_j \in Y_x$. Then either $y_i \in \alpha$ or $y_j \in \beta$, otherwise we can not have the selection as defined. To show $p \in \ \mathit{dom}^{-1}(\mathit{idom}(x))$, if it is not the case, then there exists some $\gamma = (s, \ldots, p)$ with $\mathit{idom}(x) \notin \gamma$, then by the existence of α and β we have $\mathit{idom}(x) \in \alpha$ and $\mathit{idom}(x) \in \beta$, contradicting the fact that $\alpha \cap \beta$ is a singleton x and $\mathit{idom}(x) \neq x$.
- To show $LC(x) \subseteq \mathit{PIDF}(Y_x) \cap \ \mathit{dom}^{-1}(\mathit{idom}(x))$, let $p \in LC(x)$, then there is a loop L containing p and x, a definition node $y_i \in Y_x \cap L$, and there is a successor q of p with $q \notin L$. It is obvious that $p \in \mathit{PIDF}(y_i)$ from p, y_i both in loop L, so $p \in \mathit{PIDF}(Y_x)$. Also from $x dom p$ we have $\mathit{idom}(x) dom p$, i.e., $p \in \ \mathit{dom}^{-1}(\mathit{idom}(x))$.

□

B Glossary

Symbols	Description
N	Nodes of the control flowgraph
u, v, w	Nodes of the control flowgraph
s	Start node of control flowgraph
e	End node of control flowgraph
(u, v)	Edge from u to v in flowgraph
E	Edges of flowgraph
$\mathit{preds}(u)$	Set of predecessors of node u
$\mathit{src}(u, v)$	Source u of edge (u, v)
$\mathit{src}(E')$	Source of a set of edges E'
$\alpha : u \xrightarrow{*} v$	Path α from u to v
$u \in \alpha$	Node u occurs in path α
$\mathit{Path}(u, v)$	Set of path from node u to node v
$\mathit{Path}^{\sharp}(u, v)$	Set of path from u to v with single occurrence of u
$\alpha \circ \beta$	Path concatenation of path α and β
$u \, \mathit{dom} \, v$	Node u dominates node v
$\mathit{dom}\,(u)$	Dominators of u
$\mathit{idom}\,(u)$	Immediate dominator of u
$u \, \mathit{sdom} \, v$	Node u strictly dominates v
$\mathit{DF}(u)$	Dominance frontier of node u
$\mathit{IDF}(N')$	Iterated dominance frontier of set N'
$u \, \mathit{pdom} \, v$	Node u post dominates node v
$\mathit{PIDF}(N')$	Post iterated dominance frontier of set N'
$(u, v) \prec\!\!\prec w$	Edge (u, v) is controlling node w
$(u, v) \prec\!\!\prec^{+} w$	Edge (u, v) is transitively controlling node w
$\mathit{ctrl}^{*}(u)$	Set of controlling edges of node u
Var	Set of variables
x, y	Program variables
n	Number of program variables
`nop`	Statement with no side-effects
$x := \texttt{read}$	Read statement
$x := \mathit{op}(y_1, \ldots, y_k)$	Assignment with operands
$x := \mathit{op}()$	Assignment without operands
$x := \phi(y_1, \ldots, y_k)$	Phi-node
$x := \phi'(y_1, \ldots, y_k; p_1, \ldots, p_l)$	Augmented phi-node
Y_x	Definitions $\{y_1, \ldots, y_k\}$ of a phi-node
P_x	Predicates $\{p_1, \ldots, p_l\}$ of an augmented phi-node
$\boldsymbol{z}_u$	Vector of variables in the DFA equation system
Z	Matrix of variables in the DFA equation system
z_u	Variable (a vector in $\mathcal{L}$) of node u
$\widehat{\boldsymbol{z}}$	Vector of variables in compressed DFA equations
$\widehat{\boldsymbol{z}}_x$	Variable of x in compressed DFA equations
F	RHS of DFA equations
F^i	Repeated application of function F
$\widehat{F}$	RHS of compressed DFA equations
$\mathit{mfp}(F)$	Maximum fixed point of function F
$\mathit{mfp}(u)$	MFP solution for node u of DFA equations

$\widehat{mfp}(u)$	MFP solution for node u of $\widehat{F}$
$mfp(u)(x)$	MFP solution of variable x in node u
$mop(u)$	Meet Over all Paths (MOP) solution of node u
$mop(u)(x)$	MOP solution of variable x in node u
$M(u)(\boldsymbol{c})$	Transfer function of node u
$M(\pi)(\boldsymbol{c})$	Transfer function of path π
$\vartriangle$	Untainted value
$\blacktriangle$	Tainted value
$\mathcal{L}$	Simple semi-lattice containing $\vartriangle$ and $\blacktriangle$
$\sqcap$	Meet operator in $\mathcal{L}$ and $\mathcal{L}^n$
$\sqsubseteq$	Partial order in $\mathcal{L}$ and $\mathcal{L}^n$
$\mathcal{L}^n$	Information Lattice
$\blacktriangle^n$	Bottom element in the information lattice
$\vartriangle^n$	Top element in the information lattice
$\boldsymbol{c}$	Configuration (element in $\mathcal{L}^n$)
$\boldsymbol{c}(x)$	Value of x in configuration $\boldsymbol{c}$
$\boldsymbol{c}_{[x \rightarrow a]}$	Changing value of x in $\boldsymbol{c}$
$\mathbb{B}$	Boolean lattice
G	Reachability graph
V	Set of nodes in reachability graph
R	Set of reachable nodes in reachability graph
r	Root node in reachability graph
m	Number of edges in the reachability graph
Arc	Set of edges in reachability graph
$a \leq b$	Partial order $a \vee b = a$ in $\mathbb{B}$

About the Authors

Bernhard Scholz is a Visiting Professor at Sun Microsystems Laboratories in Brisbane, Australia, where he investigates techniques in the area of static program analysis. He is a Senior Lecturer at The University of Sydney and has previously served on the faculty of the Technical University of Vienna and the University of Vienna. Before pursuing an academic career, Bernhard worked in industry at Baring Asset Management, London, UK.

Chenyi Zhang is a Graduate Intern at Sun Microsytems Laboratories in Brisbane, Australia, where he contributes to the Parfait project. He is a PhD student in the School of Computer Science and Engineering at the University of New South Wales. He also is a joint student in the formal methods group at National ICT Australia (NICTA). His current research topics are computer security focusing on noninterference properties on the traditional automata models and probabilistic process algebra.

Cristina Cifuentes is a Senior Staff Engineer at Sun Microsytems Laboratories in Brisbane, Australia, where she investigates static program analysis techniques for bug checking. Her research interests include program transformations in the areas of compiler construction, binary translation, program comprehension, software maintenance, reverse engineering, and decompilation. Cristina is the principal investigator of the Parfait project. Prior to joining Sun Microsystems Laboratories in July 1999, she held academic positions at The University of Queensland and The University of Tasmania in Australia. Cristina is the treasurer of the ACM special interest group on programming languages, SIGPLAN.