# Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs

V. Benjamin Livshits and Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

{livshits, lam}@cs.stanford.edu

## ABSTRACT

This paper proposes a pointer alias analysis for automatic error detection. State-of-the-art pointer alias analyses are either too slow or too imprecise for finding errors in real-life programs. We propose a hybrid pointer analysis that tracks actively manipulated pointers held in local variables and parameters accurately with path and context sensitivity and handles pointers stored in recursive data structures less precisely but efficiently. We make the unsound assumption that pointers passed into a procedure, in parameters, global variables, and locations reached by applying simple access paths to parameters and global variables, are all distinct from each other and from any other locations. This assumption matches the semantics of many functions, reduces spurious aliases and speeds up the analysis.

We present a program representation, called IPSSA, which captures intraprocedural and interprocedural definition-use relationships of directly and indirectly accessed memory locations. This representation makes it easy to create demand-driven path-sensitive and context-sensitive analyses.

We demonstrate how a program checker based on IPSSA can be used to find security violations. Our checker, when applied to 10 programs, found 6 new violations and 8 previously reported ones. The checker generated only one false warning, suggesting that our approach is effective in creating practical and easy-to-use bug detection tools.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.3 [**Operating Systems**]: Security and Protection

## General Terms

Algorithms, Languages, Software Security.

This material is based upon work supported in part by the National Science Foundation under Grant No. 0086160.


*ESEC/FSE'03,* September 1–5, 2003, Helsinki, Finland.


## Keywords

Program analysis, program representation, pointer analysis, error detection, SSA representation, context-sensitive analysis, path-sensitive analysis, security flaws, buffer overruns.

## 1. INTRODUCTION

A number of practical auditing and error detection tools have been shown to be effective in finding errors in existing software systems[2, 11]. Among the errors detected are buffer overruns and format string vulnerabilities, which account for a large number of reported security attacks[23].

A common requirement for such tools is the ability to follow the flow of data efficiently: from a definition to all its possible uses or from a use to all its possible definitions. What makes this problem difficult for languages like C is the presence of pointers; without performing some sort of pointer analysis it is impossible to say what locations an indirect load or store will access.

The current state-of-the-art pointer alias analyses are either too imprecise or too slow to use for bug detection. Pointer alias analysis used in program optimizations is necessarily sound: two pointers are considered to be possibly aliased if it cannot be proven otherwise. While flow-insensitive and context-insensitive analyses are fast[13, 20], they generate many spurious aliases. Program checkers built upon these techniques would raise too many false alarms and would thus be unusable. On the other hand, flow and context-sensitive algorithms have been proposed[17, 24, 26], but are too slow to be used on real-life programs.

Practical auditing tools often use unsound pointer alias analyses instead[2, 4, 11, 23]. Not only are the analyses flow and context-sensitive, they are even path-sensitive. While they track memory locations held by local variables and parameters with precision, they often assume unsoundly that all other indirect memory references are unaliased. Because these tools report errors based on information that is most likely to be true, they generate fewer false warnings. Unfortunately it is sometimes hard to understand which errors go undetected because of the ad hoc techniques used.

### 1.1 Contributions

Our goal is to develop a pointer alias analysis expressly to be used for software auditing and error detection. The following outlines the major contributions of this work.

### 1.1.1 A Hybrid Pointer Alias Analysis

We cannot afford the expense of a context-sensitive and path-sensitive analysis throughout the whole program, nor can we afford the inaccuracy of a flow-insensitive analysis. Thus, instead of using a uniform algorithm to analyze all pointers in a program, we use a hybrid approach. First, we propose a precise path and context-sensitive alias analysis to track locations referred to by simple access paths originating from parameters and local variables. Distinguishing between different field structures is crucial in C to achieve good precision[23], thus fields are kept separate in our representation. Second, we use an efficient, but imprecise pointer alias analysis to handle all the other references. We currently use Steensgaard's unification-based analysis[20], which is flow and context-insensitive and does not distinguish between fields within structures. Our design is geared toward catching errors that arise from inconsistencies around procedure boundaries and along exceptional control flow paths without producing too many false alarms.

### 1.1.2 An Unsound Assumption on Aliases

Unsoundness should be introduced in such a way that still allows users to reason about the results and to understand when the results may be incorrect. The unsound assumption we make has the dual benefit of speeding up the analysis and suppressing warnings that are likely to be false.

We assume that pointers passed into a procedure, in parameters, global variables, and locations reached by applying simple access paths to parameters and global variables, are all distinct from each other and from any other locations. Such an assumption reduces the complexity of our pointer analysis, as it allows the effects of each procedure to be summarized succinctly. Handling potential aliases between parameters precisely has proven to be difficult. Previous approaches either disallow strong updates[24], which would not be precise enough, or use techniques such as partial transfer functions[26, 25] to create summaries only for observed contexts. None of the flow-sensitive and context-sensitive techniques have been demonstrated to scale to large programs.

This assumption also matches well with how most programs are written. For modularity, the semantics of a function is often independent of the presence of possible aliases among incoming parameters. If that is not the case, a defensive programmer would insert explicit tests in the code to ensure that the potential aliases are handled properly. In these cases, our unsound assumption would not cause a fully path-sensitive analyzer to produce any inaccurate result. Our system issues a warning if it concludes the unsound assumption is *definitely* violated and that the results of the analysis are necessarily incorrect. A low-level warning is also reported if the assumption *may* be violated, but we expect that the low-level warnings are too numerous to be helpful.

### 1.1.3 Handling Paths and Contexts Efficiently

Since analyzing all potential paths in a program is infeasible, simulation-based approaches tend to use heuristics and ad hoc solutions to limit the number of paths explored[2, 11]. Our solution is to use a demand-driven approach to concentrate the resources on those paths found to be of interest.

For handling contexts, we first analyze the program to create summaries of the effect of callees on callers. An efficient whole-program context-sensitive analysis is made possible by our unsound assumption that incoming parameters are unaliased. Once this information is available, we can analyze certain context-sensitive paths efficiently on demand.

Similarly, our algorithm first performs a whole program analysis to find all the potential definition-use relationships in a flow-sensitive but path-insensitive manner. On demand, the predicates of the path of interest are analyzed.

### 1.1.4 IPSSA: A Representation for Bug Detection

Because all auditing and bug detection tools in C need to handle indirect memory accesses, we propose to analyze pointers in an application-independent manner and make the results accessible to various tools. Our representation, called IPSSA, extends the basic concept of SSA[7] to include definition-use relationships due to pointer dereferences and procedure calls.

### 1.1.5 Empirical Results: Security Violations

We demonstrate the practicality of our approach by building a tool that finds buffer overruns and format string violations using the IPSSA representation. Our tool found 14 security vulnerabilities in 10 application programs. More importantly, it reported only one false warning, which is significantly fewer in number than existing tools. These preliminary results suggest that our approach is effective in creating practical and easy-to-use bug detection tools.

## 1.2 Paper Organization

Section 2 presents an overview and design rationale of our approach. Section 3 describes our algorithm for constructing the IPSSA representation. Section 4 presents our tool for detecting security vulnerabilities and our experience in using the tool. Section 5 discusses related work and Section 6 concludes.

# 2. AN OVERVIEW OF IPSSA

The IPSSA representation connects definitions and uses in a program, even if they reside in different functions or involve indirect accesses. A use may have multiple potential definitions for four reasons:

1. alternatives in control flow,
2. accesses to dynamically determined memory locations,
3. a procedure may be called from different call sites, and
4. a call site may invoke different callees indirectly.

Gated SSA[1, 21] handles the first of these reasons; this paper proposes techniques for handling the rest.

## 2.1 Terminology

Our analysis is designed to be used for C programs. To keep the discussion simple, we will focus only on a subset of the language. A program consists of a set of functions $F$, a set of variables $V$, and a set of statements $S$. Statements are broken down to simpler statements such that indirect memory accesses occur only in one of two kinds of statements:

1. a LOAD statement loads the contents of an indirectly accessed location into a simple variable, and
2. a STORE statement stores the contents of a simple variable into an indirectly accessed location.

There are three kinds of indirect access operators: dereference (e.g. `*v`), field access (e.g. `v.fld`), and array index (e.g. `v[i]`). Structures are treated as a collection of scalar variables and fields are treated like individual variables.

## 2.2 Representation of Control Flow

In both SSA and gated SSA, definitions to the same variable are given different names so that there is only one static assignment to each variable in the program. A particular definition of variable $v$ is denoted by adding a subscript to $v$ as in $v_i$. A use of $v$ is replaced with the definition of $v$ that reaches that particular use. In the rest of the paper, we refer to the set of all definitions as $D$.

In the presence of control flow, multiple statements in the program may be responsible for producing the values used at a program point. Gated SSA introduces the concept of $\gamma$-functions to capture the multiplicity of reaching definitions and the conditions under which they apply.

**Definition 2.1** *A $\gamma$-function has the form*

$$d = \gamma(\langle p_1, d_1\rangle, \ldots, \langle p_n, d_n\rangle),$$

*where $d, d_1, \ldots, d_n \in D$ and $p_1, \ldots, p_n$ are predicates. Placed at the confluence of n control flow paths, it says that the new definition $d$ is assigned $d_i$ if predicate $p_i$ holds.*

One and only one of the predicates in a $\gamma$-function holds dynamically. The definition associated with the held predicate is the one that defines the value assigned.

By construction, each use of a variable in the gated SSA representation can be reached by only one definition, which we refer to as the *active definition*. Note that the definition operands to the $\gamma$-functions may themselves be definitions of $\gamma$-functions. We can find all the possible values of $v$ defined by a $\gamma$-function by computing the transitive closure of the operands of the function.

## 2.3 Handling Pointer Dereferences

We use a hybrid approach to handling indirect memory accesses. We use a scalable global pointer alias analysis on the whole program to determine which accesses may be aliased with each other. The alias information is represented using a flat context-insensitive name space called *abstract memory locations*. To gain more precision, we also introduce an optimistic and unsound analysis that reasons about the locations according to their *access paths*.

### 2.3.1 Abstract Memory Locations

We use a pointer alias analysis, based on Steensgaard's flow and context-insensitive analysis, to partition the memory space of a program into a set of *abstract memory locations* $M$. The abstract memory function, mem : $S \rightarrow M$, maps a given load or store statement to the abstract memory location it accesses. References mapped to different abstract memory locations are guaranteed to be unaliased; on the other hand, accesses mapped to the same abstract memory location may or may not be aliased.

By mapping all accesses to their abstract memory locations, indirect accesses can be handled like direct accesses with one exception. Since an abstract memory location may represent multiple physical locations, a store does *not* overwrite all the previously held values. Definitions to abstract memory locations are thus treated as *weak updates*—updates that add new values to the location without destroying the old ones. Destructive updates are known as *strong updates*.

We use $\phi$-functions to represent weak updates. A $\phi$-function has two definitions for operands, the first represents the new value being assigned, the second represents the old existing value being held. Since the second operand may itself be assigned a $\phi$-function, tracing all the operands to the $\phi$-function transitively will find all the values assigned to a definition of an abstract memory location. Note that unlike the $\gamma$-function, the condition under which each definition takes effect is not specified.

**Definition 2.2** *A $\phi$-function has the form*

$$d = \phi(d_1, d_2),$$

*where $d, d_1, d_2 \in D$. Introduced to model weak updates, it says that the new definition $d$ is either $d_1$ or $d_2$.*

Example 1 illustrates how abstract locations are used in IPSSA to represent array accesses. (Assume that location $m$ is used by Steensgaard's analysis to represent `D`'s elements.) As shown in this example, the value of `c` is either that of `a` or `b` depending on whether `j = i`. While the condition is relatively simple in this example, it can be quite complex in general. The $\phi$-function captures the choice between `a` and `b` without recording the condition.

**Example 1:** *Abstract memory locations in IPSSA*

| Line | Source code | IPSSA representation |
|---|---|---|
| 1 | `int a = 1;` | $a_0 = 1$ |
| 2 | `int b = 2;` | $b_0 = 2$ |
| 3 | `int c = 3;` | $c_0 = 3$ |
| 4 | `int D[10];` | $m_0 =$ UNINIT |
| 5 | `D[i] = a;` | $m_1 = \phi(a_0, m_0)$ |
| 6 | `D[j] = b;` | $m_2 = \phi(b_0, m_1)$ |
| 7 | `c = D[i];` | $c_1 = m_2$ |

### 2.3.2 Access Paths

Let us first illustrate the need for a stronger pointer analysis with the help of Example 2. Suppose we are interested in developing a tool that detects when a program dereferences a NULL pointer. In this example, a flow-sensitive, but not path-sensitive, pointer analysis would infer that `u` may point to either `a` or NULL. Thus, `*u` on line 10 yields either 1 or an error as a result of dereferencing NULL. Analyzing the paths in the program, however, reveals that `u` is dereferenced only when `P` is true on line 10, which implies that `u` can only point to `a`. Understanding paths can therefore help suppress false warnings.

The second, more precise, component in our hybrid pointer alias analysis uses a name space, separate from abstract memory locations, based on access paths. The name space of function $f$ includes:

- Local variables.
- Formal parameters.
- New locations returned by invocations of `malloc()`.
- The return location of $f$ denoted as $\text{RET}^f$.
- Global variables.

**Example 2:** *Path sensitivity in pointer analysis*

| Line | Source code | IPSSA representation |
|---|---|---|
| 1 | `int *u;` | |
| 2 | `int a = 0;` | $a_0 = 0$ |
| 3 | `if (P) {` | |
| 4 | `u = &a;` | $u_0 = \&a$ |
| 5 | `}else{` | |
| 6 | `u = NULL;` | $u_1 =$ NULL |
| 7 | `}` | $u_2 = \gamma(\langle P, u_0\rangle, \langle \neg P, u_1\rangle)$ |
| 8 | `a = 1;` | $a_1 = 1$ |
| 9 | `if (P) {` | |
| 10 | `a = *u;` | $t_1 = a_1$ |
| | | $t_2 =$ ERROR |
| | | $a_2 = \delta(\langle P, t_1\rangle, \langle \neg P, t_2\rangle)$ |
| 11 | `}` | $a_3 = \gamma(\langle P, a_2\rangle, \langle \neg P, a_1\rangle)$ |

In addition, the values in formal parameters and global variables at the entry of a function may be used in a dereference operation or a field access to *derive* further locations. We denote the location pointed to by $v$ *at the entry* of function $f$ as $v^{\wedge}$ and the location stored in field `fld` of $v$ as $v$.`fld`. We use the term *access path* to refer to a combination of dereference and field access operations. An access path is *simple* if it does not include any iterated dereference or field access operations.

We refer to local variables of a function $f$, its parameters, global variables, variables created within the function, and those locations accessible by applying simple access paths to the values of parameters and global variables at procedure entry, as the *hot locations* of $f$. We make the unsound assumption that these hot locations are distinct from each other and from any other locations.

Note that using such an assumption may lead to wrong results. In Example 3, the location pointed to by variable `u` on line 10 is given by an iterated access path `p(->next)*`, which according to our assumption, is unaliased to the location pointed to by `q` or `p`. The former is clearly false, and the latter is also false if the loop is not executed at all. The alias with `q` can potentially be discovered by a path-sensitive analysis; our current pointer analysis will miss the alias, however, as predicates are considered only after pointer analysis.

**Example 3:** *An iterated access path*

```
1    struct node {
2        struct node *next;
3    };
4    void f(struct node *p, struct node *q){
5        struct node *u, *v;
6        u = p;
7        while (u != q) {
8            u = u->next;
9        }
10       u->next = NULL;
11   }
```

Contents of hot locations are tracked path-sensitively and context-sensitively. Indirect accesses to hot locations are replaced with direct accesses. The representation is path-sensitive; that is, the condition under which each use or definition takes place is encoded with $\gamma$-functions. All other accesses are analyzed by mapping them to their abstract memory locations, as discussed in Section 2.3.1.

Returning to Example 2, the $\gamma$-function at the end of the `if` statement on line 7 states that variable `u` holds the value assigned to definition $u_0$ if $P$ holds and $u_1$ otherwise. Consider the indirect access `*u` on line 10. The active definition of $u$ on line 10 is $u_2$. When $P$ is true, $u_2$ is the same as $u_0$, which has the value $\&a$, thus `*u` is the active definition of $a$, $a_1$. When $P$ is false, $u_2$ is given by $u_1$, which has the value NULL. Dereferencing a NULL would generate an ERROR. These two possible values of `*u` are encoded by a $\gamma$-function, with the help of two fresh definitions $t_1$ and $t_2$. This example illustrates how IPSSA represents definition-use relationships of indirect accesses simply and directly.

## 2.4 Data Flow Across Procedure Boundaries

The interface to a function consists of a set of *incoming parameters* and a set of *outgoing parameters*. The incoming parameters of $f$ include not just the declared formal parameters, but all locations whose value upon function entry may be accessed by $f$ and the functions it invokes. Such locations may include global variables, abstract memory locations, as well as locations derived from parameters and global variables. Similarly, the outgoing parameters of $f$ include not just $\text{RET}^f$ but all the locations written to in $f$ that are visible outside $f$. Such locations may include global variables, abstract memory locations, results of `malloc` calls, as well as locations derived from parameters, global variables, and the return location.

We define a new pair of functions, $\iota$ and $\rho$, to handle definition-use chains that span procedural boundaries. An $\iota$-function is placed at a function entry to specify the incoming definitions for each incoming parameter. It has one operand for each call site identifying the call site and the active definition of the actual parameter. A $\rho$-function is placed after each call for every variable that is an outgoing parameter of at least one of the called functions. It has one operand for each callee identifying the function and the active definition of the return value.

**Definition 2.3** *An $\iota$-function has the form*

$$d = \iota(\langle c_1, d_1\rangle, \ldots, \langle c_n, d_n\rangle),$$

*where $c_1, \ldots, c_n$ are call sites and $d, d_1, \ldots, d_n \in D$. Placed at the entry of a function $f$, it says that $d$, representing a definition of a formal parameter, is assigned the actual definition $d_i$ if called at call site $c_i$.*

**Definition 2.4** *A $\rho$-function has the form*

$$d = \rho(\langle f_1, d_1\rangle, \ldots, \langle f_n, d_n\rangle),$$

*where $f_1, \ldots, f_n \in F$ and $d, d_1, \ldots, d_n \in D$. Placed after a call site, it says that $d$, representing a definition of a variable modified by at least one of the callees, is assigned the formal definition $d_i$ if $f_i$ is invoked.*

Example 4 illustrates how parameter passing is represented in IPSSA. Suppose the function pointer `fp` has been found to point to either `f1` or `f2`. An $\iota$-function with two operands, representing the two potential call sites `c1` and `c2`, is used to define the formal parameter in each of the functions, `f1` and `f2`. Special locations $\text{RET}^{f1}$ and $\text{RET}^{f2}$ are introduced to represent the return values for functions `f1` and `f2`, respectively. Finally, a $\rho$-function is introduced at each call site to combine the results of the two possible functions invoked.

**Example 4:** *Interprocedural data flow in the presence of function pointers*

| Line | Source code | IPSSA representation |
|---|---|---|
| 1 | `int f1(int a){` | $n_0 = \iota(\langle \mathsf{c1}, d_0\rangle, \langle \mathsf{c2}, e_0\rangle)$ |
| 2 | `  return a+1;` | $\text{RET}^{f1}{}_0 = a_0 + 1$ |
| 3 | `}` | |
| 4 | `int f2(int b){` | $b_0 = \iota(\langle \mathsf{c1}, d_0\rangle, \langle \mathsf{c2}, e_0\rangle)$ |
| 5 | `  return b-1;` | $\text{RET}^{f2}{}_0 = b_0 - 1$ |
| 6 | `}` | |
| 7 | `void foo(){` | |
| 8 | `  int d = ...;` | $d_0 = \ldots$ |
| 9 | `c1:d = (*fp)(d);` | $d_1 = \rho(\langle \mathsf{f1}, \text{RET}^{f1}{}_0\rangle, \langle \mathsf{f2}, \text{RET}^{f2}{}_0\rangle)$ |
| 10 | `}` | |
| 11 | `void bar(){` | |
| 12 | `  int e = ...;` | $e_0 = \ldots$ |
| 13 | `c2:e = (*fp)(e);` | $e_1 = \rho(\langle \mathsf{f1}, \text{RET}^{f1}{}_0\rangle, \langle \mathsf{f2}, \text{RET}^{f2}{}_0\rangle)$ |
| 14 | `}` | |

# 3. CONSTRUCTION OF IPSSA

Our IPSSA construction algorithm is based on the algorithm proposed for SSA[7], as outlined below:

1. Give all the assigned variables new names.
2. Introduce $\phi$-functions in the case of SSA, or $\gamma$-functions in the case of gated SSA, at the iterated dominance frontier of assignments[7].
3. Rename each use in the program by the definition that defines the value of that use. The active definition of a use can be found by walking up the dominator tree of the program's control flow graph starting with the node representing the use and locating the first definition of that variable.

Our algorithm uses similar steps to handle every statement in the program except for loads, stores and procedure calls. We make the following major extensions to represent interprocedural relationships and indirect accesses:

1. Since there are many interprocedural paths in a program, gates of the $\gamma$-functions are generated on demand as the paths of interest are identified.
2. The IPSSA representation captures the definition-use relationships within each function, fully taking into account the effects of the functions invoked therein. This feature makes it relatively easy to track context-sensitive paths originating from and terminating in different functions based on demand.
3. All indirect accesses are replaced with direct accesses to hot or abstract memory locations. For each indirect access, the analysis looks up the definition-use relationships found so far to determine the addresses accessed and create further definition-use relationships. Thus, not all definitions are known a priori and a fixpoint computation is necessary to determine the final representation.

Due to space limitations, we will concentrate only on the third and most interesting extension below.

## 3.1 Computing the Fixpoint

We use Steensgaard's flow and context-insensitive pointer analysis[20] to identify call targets. This approach seems to suffice for programs that do not use indirect function calls aggressively.

In the absence of recursion, the IPSSA representation can be built in a single pass over the call graph, in a bottom-up manner starting with the leaf nodes. Each function is analyzed iteratively to accomplish the following:

1. Replace indirect accesses and definitions with direct accesses and definitions to abstract memory locations or hot locations.
2. At each call site, enter the actual incoming parameters as operands in the $\iota$-functions of each callee.
3. At each call site, build $\rho$-functions to capture all the outgoing parameters in the callees.
4. Identify the function's incoming and outgoing parameters, and create $\iota$-functions for the incoming ones.

A fixpoint computation is needed to handle recursion. We first find the strongly connected components (SCCs) of the call graph. Interprocedurally, we compute the fixpoint solution for one SCC at a time in a bottom-up manner, starting with the leaf SCCs. The iteration stabilizes when no more changes are made to the representation of an SCC.

To handle strong updates of indirect accesses to hot locations, our algorithm does not analyze a statement (other than the entry of the procedure) unless at least one of its predecessors has been visited. The reason is illustrated by the following example:

```
1     x  = 1;
2     *p = 2;
3     y  = x;
```

If `*p` happens to carry the singleton address of `&x`, `x` on line 3 can only carry the value 2 and not 1. Without any knowledge of the value of `*p`, monotonicity requires that line 2 be modeled as overwriting all variables. Thus, processing line 3 would not have yielded any information. We can proceed to analyze line 3 as long as `*p` is known to carry at least one value and the iteration process is guaranteed to find the greatest fixpoint. The same principle requires that whenever a call site is encountered, at least one of the callees has to be analyzed from its entry to one exit. This is always possible unless the call never returns.

## 3.2 Indirect Accesses

We handle indirect accesses by tracking down the definitions of the dereferenced variables to determine the set of hot locations that might be accessed. We use abstract memory locations whenever accesses outside the hot locations are made. If a variable to be dereferenced is defined by a copy statement, a $\gamma$-function, a $\rho$-function, or an $\iota$-function in a callee, we need to follow its definition transitively to find the sources of all the values assigned. We describe the process of resolving an indirect access as an iterative application of a set of conditional rewrite rules. The process terminates when no dereferencing operations are left in the representation, or if the access path is determined to be recursively defined, as discussed in Section 3.2.2.

| Case | | [Load] s: $w_s = *d$ | [Store] s: $*d = d'$ |
|---|---|---|---|
| [**Direct**] | $d = \&v$ | $w_s = \text{ad}(v, s)$ | $v_s = d'$ |
| [**Expression**] | $d$ is assigned an expression, $m = \text{mem}(s)$ | $w_s = \text{ad}(m, s)$ | $m_s = \phi(d', \text{ad}(m, s))$ |
| [**Abstract**] | $d = m'_{s'}$, $m = \text{mem}(s)$ | $w_s = \text{ad}(m, s)$ | $m_s = \phi(d', \text{ad}(m, s))$ |
| [**Copy**] | $d = d_1$ | $w_s = *d_1$ | $*d_1 = d'$ |
| [$\gamma$-**function**] | $d = \gamma(\langle p_1, d_1\rangle, \ldots, \langle p_n, d_n\rangle)$ $t_1, \ldots, t_n$ fresh | $t_1 = *d_1$ $\ldots$ $t_n = *d_n$ $w_s = \gamma(\langle p_1, t_1\rangle, \ldots, \langle p_n, t_n\rangle)$ | $t_1 = *d_1$ $*d_1 = \gamma(\langle p_1, d'\rangle, \langle \neg p_1, t_1\rangle)$ $\ldots$ $t_n = *d_n$ $*d_n = \gamma(\langle p_n, d'\rangle, \langle \neg p_n, t_n\rangle)$ |
| [$\rho$-**function**] | $d = \rho(\langle f_1, d_1\rangle, \ldots, \langle f_n, d_n\rangle)$ $t_1, \ldots, t_n$ fresh, $s$ pushed on call stack when traversing $d_i$ | $t_1 = *d_1$ $\ldots$ $t_n = *d_n$ $w_s = \rho(\langle f_1, t_1\rangle, \ldots, \langle f_n, t_n\rangle)$ | $t_1 = *d_1$ $*d_1 = \rho(\langle f_1, d'\rangle, \ldots, \langle f_n, d_n\rangle)$ $\ldots$ $t_n = *d_n$ $*d_n = \rho(\langle f_1, d_1\rangle, \ldots, \langle f_n, d'\rangle)$ |
| [$\iota$-**function: 1**] | $d = v_0$ $v_0 = \iota(\langle s_1, d_1\rangle, \ldots, \langle s_n, d_n\rangle)$ call stack is empty | $w_s = \text{ad}(v^\wedge, s)$ | $v^\wedge_s = d'$ |
| [$\iota$-**function: 2**] | $d = \iota(\langle s_1, d_1\rangle, \ldots, \langle s_n, d_n\rangle)$ $s_i$ popped from call stack | $w_s = *d_i$ | $*d_i = d'$ |

**Table 1: Conditional rewrite rules for pointer resolution**

### 3.2.1 Conditional Rewrite Rules

The rewrite rules are summarized in Table 1. We assume a set of definitions $d \in D$, abstract memory locations $m \in M$, statements $s \in S$, variables $v, w, t \in V$. For clarity, the definition of variable $v$ at statement $s$ is given the name $v_s$. We denote the active definition for $v$ at statement $s$ as $\text{ad}(v, s)$. All local variables are initialized with UNINIT. If the active definition of variable $v$ is demanded in the application of a rewrite rule and none is available, variable $v$ is identified as an incoming parameter and a definition $v_0 = \iota(...)$ is inserted at the entry of the function.

Load `s:` $w_s = *d$ and store `s:` $*d = d'$ statements are rewritten as follows:

[**Direct**] If $d$ is assigned an address-of operation $\&v$, $*d$ is simply the variable $v$.

[**Expression**] If $d$ is a definition involving arithmetic or array element reference, model it with its abstract memory location $m = \text{mem}(s)$. If $s$ is a load statement, replace $*d$ with the active definition of $m$ at $s$. If $s$ is a store statement, a new definition for $m$ is created. To model the weak update, the right-hand side is a $\phi$-function, whose new value is $d'$ and the old value is given by $\text{ad}(m, s)$.

[**Abstract**] If $d$ is a definition to an abstract memory location $m'$, simply model the address accessed by its abstract memory location. We do not look up the definitions of $m'$ because $m'$ can only be updated weakly and thus may point to many different locations.

[**Copy**] If $d$ is assigned another definition $d_1$, $d$ is substituted with $d_1$.

[$\gamma$-**function**] If $d$ is assigned the result of a $\gamma$-function, we create fresh variables $t_1, \ldots, t_n$, to represent the dereferenced result for each of the possible definitions. For a load statement, a $\gamma$-function, which preserves predicates properly, is defined to collect the dereferenced results. For a store statement, a definition is generated for each potential destination. A dereferenced location retains its old value if the predicate under which it is chosen is false.

[$\rho$-**function**] If $d$ is assigned the result of a $\rho$-function, the statement is rewritten like $\gamma$-functions, with the exception that the temporary variables are assigned definitions in the callees. When interpreting definitions in a callee, to support context sensitivity we need to record that the search was initiated at call site $s$, in case the pointer resolution takes the search to the entry of the callee and back to the caller. This information is used when rewriting $\iota$-functions described below. To model the function call semantics, call sites encountered are pushed onto a call stack.

[$\iota$-**function**] If $d = v_0$ is assigned the result of an $\iota$-function, the action depends on whether the call stack is empty. If the call stack is empty, the search has reached the entry of the function that initiated the pointer resolution. The location accessed is simply represented symbolically as $v^\wedge$. If the call stack is not empty, the search continues back to the caller by popping the call site and selecting the corresponding actual parameter.

### 3.2.2 Terminating Pointer Resolution

If either of the conditions below is detected during the pointer resolution process, the process is aborted and the access is simply modeled with its abstract memory location.

1. The access path is recursively defined, as illustrated in Example 3. By collecting the definitions visited so far, we can detect when the same definition is revisited and identify recursive paths.

2. New objects are ambiguously named. Newly allocated variables, returned by `malloc()`, are treated as unique objects, named by the call site. Such a scheme is not powerful enough to handle the case when a program uses several hot locations to hold onto several instances created by the same call site, as illustrated in Example 5. We do not allow pointer resolution to loop around `malloc` calls.

**Example 5:** *Calls to* `malloc` *in a loop*

```
1     p = malloc(...)
2     while (...) {
3         q  = malloc(...);
4         *q = 1;
5         x  = *p;
6         *q = 2;
7         p  = q;
8     }
```

On line 5, `q` and `p` point to two different objects created in two consecutive iterations of the loop. They have values 1 and 2, respectively. The single `malloc()` call site on line 5 cannot be used to refer to both objects.

### 3.2.3 Inserting New Definitions

Inserting a new definition in the IPSSA representation at statement $s$ may introduce new definitions at the iterated dominance frontier of $s$ and require updating some of the uses of the variable being defined. To facilitate this operation, we keep track of not just the use-definition relations in SSA but also the definition-use relations. We denote the set of uses of a definition $d$ by $\text{USES}(d)$ and the set of definitions used in $u$ by $\text{DEFS}(u)$. The steps to create a definition $d$ of location $l$ at statement $s$ are:

1. Collect $D_l$, the set of all reaching definitions[7] of $l$ at $s$. Usually this set contains just the active definition of $l$ at $s$, but when $s$ is a join node, $D_l$ may contain more than one element, one for each predecessor of $s$ in the control flow graph.

2. For each $d' \in D_l$ and each use $u \in \text{USES}(d')$, if $d$ is located after $d'$ on each path from $d'$ to $u$, update $\text{ad}(l, s')$ to be $d$ and update $\text{USES}(d)$ and $\text{DEFS}(u)$ accordingly.

3. For each statement on the dominance frontier of $s$, insert a new $\gamma$-function definition for $l$ if none is present. Otherwise, substitute the definition active at $s$ with $d$ in the existing $\gamma$-function.

Clearly, further definitions can be introduced in step 3, which would require the process to be repeated. Termination is guaranteed, however, because the iterated dominance frontier of $s$ is bounded.

## 3.3 Handling call sites

At each call site, we connect the actual incoming parameters to the formal parameters of each callee and connect the formal outgoing parameters to the actual.

Recall that an incoming parameter may be a declared parameter, a global variable or an abstract memory location. For the former, the correspondence between the formal and actual parameters is given by the operands supplied in the call itself. For all the other cases, the names of the formal and actual are the same. ^ and field operators in the access path of a formal parameter are translated into dereference and field operations of the actual parameter. At call site $c$, for each actual incoming parameter $p$, and for each callee $f$, we insert $\langle c, \text{ad}(p, c)\rangle$ as an operand of the $\iota$-function for the corresponding formal parameter in $f$. If the same actual parameter is passed in as different formal parameters, we detect a violation to our unsound assumption and report a warning.

The matching of actual and formal outgoing parameters is similar to that of incoming parameters, except for the return value. The actual parameter matching the formal return value of $f$, $\text{RET}^f$, is the one assigned the result of the function call. Different callees may have different outgoing parameters. The set of actual outgoing parameters of the call site is thus the union of those for each callee. For each actual outgoing parameter $p$, we create

$$p_s = \rho(\langle f_1, d_1\rangle, \ldots, \langle f_n, d_n\rangle)$$

such that $d_i$ has the value $\text{ad}(p', \text{EXIT}(f_i))$, if $f_i$ has corresponding formal parameter $p'$, and $\text{ad}(p, s)$ otherwise.

# 4. EXPERIMENTAL RESULTS

We have implemented our IPSSA construction algorithm in the SUIF 2 compiler infrastructure. To demonstrate the practicality of our approach, we have also built a checker that utilizes the IPSSA representation to find two of the most common sources of security vulnerability: buffer overruns and format string violations.

## 4.1 A Security Vulnerability Detector

We say that a program has a security vulnerability if there exists a definition-use chain such that:

1. The source of the definition-use chain is a user-supplied string. Examples include the return results of library functions such as `gets`, `fgets`, `read`, `msgrcv`, `getpw`, `readdir`, as well as the `argv` argument of `main`. We refer to such strings as *tainted* because the user gets to control their lengths and contents.

2. The sink of the chain is one of the following:

   (a) A write into a buffer with a statically declared size. The user-supplied string may be longer than the buffer and may thus overrun it.

   (b) A format string argument to a system function of the `printf` family. The user-supplied string may contain format specifiers that can cause program data be overwritten.

3. Definition-use chains leading to static buffers must not contain calls to functions like `snprintf`. Guaranteed not to store anything beyond a given buffer length, `snprintf` effectively *untaints* the user-supplied string and prevents buffer overrun. This does not apply to format string violations.

Note that our definition of vulnerabilities does not include buffer overruns where arrays strings are manipulated as arrays of characters.

Our checker starts by finding the sources of taintedness in the program, and follows the definition-use chains in the IPSSA representation until a sink satisfying one of the above conditions is reached. The gates of the $\gamma$-functions along the paths are created and the predicates are analyzed, using the Yacas computer algebra system[16], to determine if the path is feasible. If a potential violation is discovered, our checker prints the complete definition-use chain found including the predicates that must hold for that particular path to be executed.

Space limitations preclude a full description of our algorithm, so only a brief overview is presented here. To handle

library functions such as `strcpy`, `strncpy`, `strcat`, `sprintf`, we create models for them written in a small special-purpose language designed for the purpose. The analysis is fully context-sensitive; when following the definition-use chains, call sites are pushed on a call stack at function entries and popped off at function exits to avoid unrealizable interprocedural paths. We detect cycles in our search and cache intermediate results to avoid exploring the same path more than once.

## 4.2 Applications in the Experiment

We applied our security vulnerability checker to 10 applications, many of which are server programs. As vulnerability in these programs poses a significant security risk, many of them have received a great deal of scrutiny. The programs are listed in Table 2, in order of their lengths. About 50,000 lines of code were analyzed in our experiment in total, with the largest program being in excess of 13,000 lines. `lhttpd` is a small web server, `polymorph` cleans up Unix filesystems after downloading binaries from Usenet, `bftpd` and `trollftpd` are FTP servers, `man` displays Unix manual pages, `pgp4pine` adds PGP encryption to the Pine mail reader, `cfingerd` is a `finger` daemon, `muh` is a network game, `gzip` is a compression tool, and `pcre` is a regular expression library.

| Program | Version | Lines of code | # proc's | IPSSA constr. |
|---|---|---|---|---|
| `lhttpd` | 0.1 | 888 | 21 | 5.2 s |
| `polymorph` | 0.4.0 | 1,015 | 19 | 1.0 s |
| `bftpd` | 1.0.11 | 2,946 | 47 | 3.2 s |
| `trollftpd` | 1.26 | 3,584 | 48 | 11.3 s |
| `man` | 1.5h1 | 4,139 | 83 | 29.3 s |
| `pgp4pine` | 1.76 | 4,804 | 69 | 17.5 s |
| `cfingerd` | 1.4.3 | 5,094 | 66 | 15.5 s |
| `muh` | 2.05d | 5,695 | 95 | 20.4 s |
| `gzip` | 1.2.4 | 8,162 | 93 | 17.0 s |
| `pcre` | 3.9 | 13,037 | 47 | 22.4 s |

**Table 2: Applications and their analysis times**

Table 2 reports the versions of the programs analyzed, the number of lines of code, the number of procedures, and the time taken to create the IPSSA representation, assuming that the control flow graph, dominators, and Steensgaard's pointer alias analysis have already been computed. The measurements were taken on a 2.2 GHz Pentium 4 machine with 2GB of memory running Linux. IPSSA is constructed in under 30 seconds for each of the applications.

The results of running our security vulnerability checker are summarized in Table 3. For each application, we show the number of warnings reported and a break-down of these warnings into three categories: buffer overrun vulnerabilities, format string vulnerabilities, and false alarms. Some of the paths reported share the same sources and sinks; we also report the total number of sources and sinks identified in the warnings. To provide insight into the nature of these paths, we report the number of definitions found along each path and the number of procedures it spans. Finally, we report the time our checker took for each program. With the exception of `lhttpd`, described below, the analysis ran in under 30 seconds for each application.

## 4.3 Detected Vulnerabilities

Our checker found a total of 11 buffer overruns and 3 format string errors in 10 applications. All the previously reported overruns and format string vulnerabilities in these programs were detected by our checker, except for two in `trollftpd` and `pgp4pine` because they manipulate strings directly as character arrays. The checker also found six vulnerabilities, which to the best of our knowledge, were not previously known. Four of these, found in `pgp4pine`, can be easily exploited by supplying a specially crafted command line argument to the program. The other two, one in `trollftpd` and one in `polymorph`, however, do not appear to be exploitable.

All the violations detected span more than one procedure. This is not unexpected: all of our applications have been relatively well tested and most remaining bugs are not easy to find. Errors that span different functions from different files are harder to find. For example, the format string violation in `muh` passes through two files and one library function (`strcpy`). Some of these definition-use chains are long; for example, there are 23 and 24 definitions in each of the reported paths in `lhttpd` and `trollftpd`, respectively. It is imperative that bug detectors be able to track long interprocedural definition-use chains. The sparse IPSSA representation is designed expressly for this purpose.

## 4.4 False Positives

What perhaps is even more significant is that our checker reports only one false alarm after analyzing about 50,000 lines of code. The single false positive, found in `pcre`, is due to the following statement:

```
sprintf(buffer, "%.512s%c%.128s",
                filename, sep, nextfile);
```

`filename` is a tainted variable and `buffer` is declared statically. The reported overrun warning is a false alarm because the width specifier `%.512s` limits the length of the copied string. Without detailed knowledge of format strings, our checker is unable to filter out this false positive.

Our unsound assumption that hot locations are unaliased reduces spurious aliases that would otherwise cause many more false warnings. We came across only one definite violation of our assumption, in `trollftpd`. The offending statement is:

```
if (snprintf(wd,PATH_MAX,"%s/%s",wd,dir) < 0) {
   quit421("Path too long",__LINE__);
}
```

The same variable `wd` is passed into `snprintf` twice. This code essentially appends `dir` to `wd`, while ensuring that doing so does not overflow the buffer allocated for `wd`. The aliasing of the first and third arguments is potentially dangerous because an implementation of `snprintf` that clears the output string first would not provide the intended functionality. Our experience suggests that our assumption matches how programs are usually written.

We found that path sensitivity is useful in reducing false warnings in the `lhttpd` program. `lhttpd` uses a standard library function `strtok` for parsing lines of a file. If the first string argument to `strtok` is not NULL, the argument is saved as part of `strtok`'s internal state. Subsequent calls to `strtok` return portions of that string. We capture the conditional data flow faithfully using a function model. This allows the checker to analyze the predicates and conclude that data flow between two invocations of `strtok` with a

| Program | Total warnings | Buffer overruns | Format strings | False alarms | Sources | Sinks | Definitions on path | Proc. spanned | Analysis time |
|---|---|---|---|---|---|---|---|---|---|
| `lhttpd` | 1 | 1 | 0 | 0 | 4 | 1 | 24 | 14 | 99.0 s |
| `polymorph` | 2 | 2 | 0 | 0 | 2 | 2 | 7, 8 | 3, 3 | 2.4 s |
| `bftpd` | 2 | 1 | 1 | 0 | 5 | 2 | 5, 7 | 1, 3 | 2.3 s |
| `trollftpd` | 1 | 1 | 0 | 0 | 4 | 1 | 23 | 5 | 8.5 s |
| `man` | 1 | 1 | 0 | 0 | 3 | 1 | 6 | 4 | 9.6 s |
| `pgp4pine` | 4 | 4 | 0 | 0 | 3 | 4 | 5, 5, 5, 5 | 3, 3, 3, 3 | 27.1 s |
| `cfingerd` | 1 | 0 | 1 | 0 | 5 | 1 | 10 | 4 | 7.4 s |
| `muh` | 1 | 0 | 1 | 0 | 3 | 1 | 7 | 3 | 7.5 s |
| `gzip` | 1 | 1 | 0 | 0 | 3 | 1 | 7 | 5 | 2.0 s |
| `pcre` | 1 | 0 | 0 | 1 | 3 | 1 | 6 | 4 | 9.2 s |
| **Total** | **15** | **11** | **3** | **1** | — | — | — | — | — |

**Table 3: Detected security violations and the checker's running time**

non-NULL first argument does not exist. This information subsequently leads to a suppression of 20 false warnings. We note that the analysis for `lhttpd` took about 100 seconds, which is short compared to the time a programmer would have to spend had these false warnings been reported.

Most other tools report a significantly larger number of false warnings. For example, Shankar et al.'s checker reported 12 false warnings for the one bug in `muh`. Our results suggest that our checker is more effective than existing tools in suppressing false warnings.

# 5. RELATED WORK

## 5.1 SSA and Gated SSA Construction

The commonly-used efficient algorithm for SSA construction was proposed by Cytron et al[7]. Ballance et al. show how to construct gates on top of an existing SSA representation[1]. Tu and Padua speed up the algorithm by introducing the gates during the construction of the SSA representation[21, 22]. We extended their approach to create gates on demand.

Chow et al. address the problem of using SSA in languages with aliasing for the purpose of program optimization[5]. New functions $\chi$ and $\mu$ are introduced to model may-define and may-use relations, respectively. These functions are similar to the $\phi$ function we use in IPSSA. Cytron et al. describe how may-aliasing information can be incorporated into SSA[8]. The representation is refined until a given optimization problem can be solved, thus the size of the representation is kept relatively small. Lapkowski et al. introduce the concept of *primary* and *secondary* SSA numbers[14]. The primary number refers to the version of the variable itself; the secondary one refers to the version of the dereference and is defined for pointer variables. As in regular SSA, variables with the same number refer to the same value. Our representation for interprocedural definition-use relationships share some similarity with the SSA-like representation proposed by Liao et al. for slicing Fortran programs[15].

Our work is different from the above in that we propose using a custom, unsound pointer analysis, rather than just incorporating the results of a known pointer analysis in the representation. Chase et al. deal with the issue of performing pointer analysis while incrementally constructing the representation[3]. Wilson and Lam also use a sparse representation in pointer alias analysis[26].

## 5.2 Error Detection Tools

A number of multi-purpose error-detection tools have been proposed in recent years. Some systems, such as Vault[10], are sound; Intrinsa's PREfix[2] and the xgcc checker[4, 11] are not. The representation proposed here is designed to facilitate the development of practical, unsound checkers.

Since the emergence of costly buffer overflow vulnerabilities in the last few years, several lexical tools have been created to help with source-level security auditing. PScan is a fast and simple tool that searches for non-constant strings passed into varargs functions[9]. The RATS scanning tool provides a security analyst with a list of potential trouble spots on which to focus, along with suggested remedies[19]. It tries to rule out unexploitable problems and assesses the potential severity of each vulnerability. ITS4 scans the source code, looking for function calls that are potentially dangerous[6]. Neither RATS nor ITS4, however, perform definition-use chain analysis, which leaves the security analyst with a lot of manual work.

Wagner et al. developed a flow-insensitive range analysis that detects potential buffer overruns; it finds 1 real bug for every 10 warnings generated[23]. Shankar et al. use a type-based analysis on a number of benchmarks similar to ours to detect format string violations[18]. Despite various refinements introduced to reduce the number of false positives, the analysis still occasionally registers a high false-positive ratio. Evans et al. present the results of their type-based taintedness analysis for several programs[12]. This tool can be quite time-consuming to use because programmers must supply a large number of annotations to reduce the number of false alarms.

# 6. CONCLUSIONS

This paper proposes a new pointer alias analysis for automatic error detection. The analysis is based on an unsound assumption that pointers passed into a procedure, in parameters, global variables, and locations reached by applying simple access paths to parameters and global variables, are all distinct from each other and from any other locations. This assumption matches the semantics of many functions, reduces spurious aliases and speeds up the analysis. To create an efficient and precise pointer alias analysis, we use a hybrid approach. Pointers accessed through simple access paths are analyzed precisely; an efficient flow and context-insensitive analysis is used for other indirect accesses.

Our proposed IPSSA captures intraprocedural and interprocedural definition-use chains for both directly and indirectly accessed memory locations, making them available to subsequent analysis. We have implemented a checker that finds buffer overruns and format string violations to demonstrate the effectiveness of our approach.

Our preliminary experience with the checker suggests that IPSSA is effective. Our checker found 6 new violations and 8 previously reported ones in 10 programs. The checker generated only one false warning, making it much easier to use than existing tools.

## 7. REFERENCES


[1] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. In *Proceedings of Software Practice and Experience*, pages 775–802, 2000.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[4] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, pages 59–70, 2000.

[5] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the Sixth International Conference on Compiler Construction*, pages 253–267, 1996.

[6] Cigital. ITS4: Software security tool. `http://www.cigital.com/its4/`.

[7] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 253–267, June 1993.

[9] A. DeKok. PScan: A limited problem scanner for C source files. `http://www.striker.ottawa.on.ca/~aland/pscan/`.

[10] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the ACM Conference on Operating Systems Design and Implementation*, pages 1–16, 2000.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[13] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 146–161, 2001.

[14] C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to language with multi-level pointers. In *Proceedings of the Seventh International Conference on Compiler Construction*, pages 128–143, 1998.

[15] S.-W. Liao, A. Diwan, R.P. Bosch, A. Ghuloum, and M. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, pages 37–48, 1999.

[16] A. Pinkus. Yacas manual. `http://www.xs4all.nl/~apinkus/manindex.html`.

[17] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[18] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–220, 2001.

[19] Secure Software. RATS, a scanning tool. `http://www.securesoftware.com/rats`.

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[21] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 47–55, 1995.

[22] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, pages 414–423, 1995.

[23] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, 2000.

[24] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *In Proceedings of Object-oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[25] R. P. Wilson. *Efficient, Context-Sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, 1998.

[26] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.