Buffer Overrun Detection using Linear Programming and Static Analysis

Vinod Ganapathy, Somesh Jha
{vg|jha}@cs.wisc.edu
Computer Sciences Department, University of Wisconsin-Madison

David Chandler, David Melski, David Vitek
{chandler|melski|dvitek}@grammatech.com
Grammatech Inc., 317, N. Aurora St., Ithaca NY 14850

UW-MADISON COMPUTER SCIENCES TECHNICAL REPORT 1488

Abstract

This paper addresses the issue of identifying buffer overrun vulnerabilities by statically analyzing C source code. We demonstrate a scalable analysis based on modeling C string manipulations as a linear program. We also present fast, scalable solvers based on linear programming, and demonstrate how to make the analysis context sensitive. Based on these techniques, we built a prototype and used it to identify several vulnerabilities in popular security critical applications.

1 Introduction

Buffer overruns are one of the most exploited class of security vulnerabilities. In a study by the SANS institute [3], buffer overruns in RPC services ranked as the top vulnerability to UNIX systems. A simple mistake on the part of a careless programmer can cause a serious security problem. Consequences can be as serious as a remote user acquiring `root` privileges on the vulnerable machine. To add to the problem, these vulnerabilities are easy to exploit, and several "cookbooks" [4, 31] are available to construct such exploits. As observed by several researchers [23, 34], C is highly vulnerable because there are several library functions that manipulate buffers in an unsafe way. Millions of lines of legacy code have been written in C, and systems running these applications continue to be vulnerable.

Several approaches have been proposed to mitigate the problem – these range from dynamic techniques [8, 10, 12, 14, 24, 27] that *prevent* attacks based on buffer overruns, to static techniques [17, 23, 29, 33, 34] that examine source code to *eliminate* these bugs before the code is deployed. Combinations of static and dynamic techniques have also been proposed where the results of static analysis are used to remove run-time checks. Unlike static techniques, dynamic techniques do not eliminate bugs, and often have the undesirable effect of causing the application to crash when an attack is discovered. Static techniques have the added advantage that they impose no run-time overhead on the applications.

In this paper, we describe the design and implementation of a tool that statically analyzes C source code to detect buffer overrun vulnerabilities. In particular, this paper demonstrates:

- The use of static analysis to model C string manipulations as a linear program.
- The design and implementation of fast, scalable solvers based on novel use of techniques from the linear programming literature. The solution to the linear program determines buffer bounds.
- Techniques to make the program analysis context sensitive.
- The efficacy of other program analysis techniques, such as static slicing to understand and eliminate bugs from source code.

One of our principle design goals was to make the tool scale to large real world applications. We used the tool to audit several popular and commercially used packages. The tool identified 14 previously unknown buffer overruns in `wu-ftpd-2.6.2` (Section 6.1.1) in addition to several known vulnerabilities in other applications.

The rest of the paper is laid out as follows: Section 2 describes the overall architecture of our tool. Section 3 and Section 4 describe the design of two solvers used by our tool. Section 5 describes a technique to make the program analysis context-sensitive. We report our experience with the prototype implementation in Section 6. Section 7 discusses related work, and Section 8 concludes.

2 Overall Tool Architecture

The tool has five components (Figure 1) that are described in the remainder of this section. Section 2.1

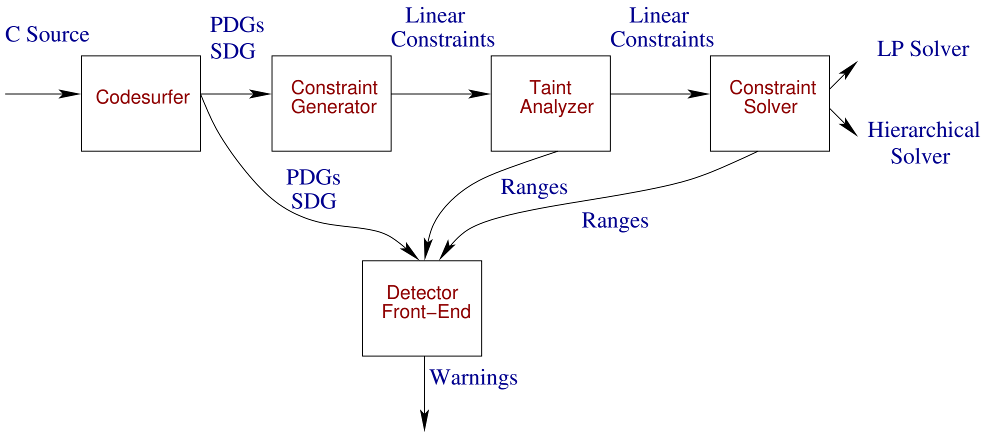

Figure 1: Overall Architecture of the Buffer Overrun Tool

```
main(int argc, char* argv[]){
   char header[2048], buf[1024],
        *cc1, *cc2, *ptr;
   int counter;
   FILE *fp;
   ...
   ptr = fgets(header, 2048, fp);
   cc1 = copy_buffer(header);
   for (counter = 0; counter < 10;
                            counter++){
       ptr = fgets (buf, 1024, fp);
       cc2 = copy_buffer(buf);
   }
}

char *copy_buffer(char *buffer){
   char *copy;
   copy = (char *) malloc(strlen(buffer));
   strcpy(copy, buffer);
   return copy;
}
```

Figure 2: Running Example

describes the code-understanding tool CodeSurfer. CodeSurfer is used by the *constraint generator*, the *detector front-end*, and to help the user examine potential overruns. Section 2.2 describes constraint generation. Section 2.3 presents *taint analysis*, which identifies and removes unconstrained constraint variables. Section 2.4 describes how to solve the constraint system, and Section 2.5 explains how to use the solution to the constraint system in order to detect potential buffer overruns. The program in Figure 2 will serve as a running example.

2.1 Codesurfer

The constraint generator and the detector front-end are both developed as plug-ins to CodeSurfer. CodeSurfer is a code-understanding tool that was originally designed to compute precise interprocedural slices [20, 21]. CodeSurfer builds a whole program representation that includes a system dependence graph (that is composed of program dependence graphs for each procedure), an interprocedural control-flow graph, abstract syntax trees (ASTs) for program expressions, side-effect information, and points-to information. CodeSurfer presents the user with a GUI for exploring its internal program representations. The queries that CodeSurfer supports include forward and backward slicing from a program point, precise interprocedural chopping between two program points (for details, see [28]), finding data and control dependence predecessors and successors from a program point, and examining the points-to set of a program variable. CodeSurfer presents the user with a listing of their source code that is "hot", i.e., the user can click on a program point in their code and ask any of the queries listed above.

CodeSurfer has two primary uses in the buffer overrun tool: (1) the constraint generator is a CodeSurfer plug-in that makes use of CodeSurfer's ASTs and pointer analysis (an implementation of Andersen's analysis [6]). (2) the detector front-end is a CodeSurfer plug-in that uses CodeSurfer's GUI in order to display potential overruns. Information about potential overruns is linked to CodeSurfer's internal program representation, so that the user can make use of CodeSurfer's features, such as slicing, in order to examine potential overruns.

2.2 Constraint Generation

Constraint generation is similar to the approaches proposed in [17, 23, 33]. We also use points-to information returned by Codesurfer, thus allowing for more precise constraints. Each pointer `buf`, to a character buffer, is modeled by four constraint variables – `buf!used!max` and `buf!used!min`, which denote the maximum and minimum number of bytes used in the buffer, and `buf!alloc!max` and `buf!alloc!min`, which denote the maximum and minimum number of bytes allocated for the buffer.

Each integer variable `i` is modeled by the constraint variables `i!max` and `i!min` which represent the maximum and minimum value of `i`, respectively. Program statements that operate on character buffers or integer variables are modeled using linear constraints over constraint variables.

Our constraints model the program in a *flow-* and *context-insensitive* manner, with the exception of library functions that manipulate character buffers. A flow-insensitive analysis ignores the order of statements, and a context-insensitive analysis does not differentiate between multiple call-sites to the same function. For a function call to a library function that manipulates strings (e.g., `strcpy` or `strlen`), we generate constraints that model the effect of the call; for these functions, the constraint model is context-sensitive. In Section 5, we will show how we extended the model to make the constraints context-sensitive for user defined functions as well.

Constraints are generated using a single pass over the program's statements. There are four program statements that result in constraint generation: buffer declarations, assignments, function calls, and return statements. A buffer declaration such as `char buf[1024]` results in constraints that indicate that `buf` is of size 1024. A statement that assigns into a character buffer (e.g., `buf[i]='c'`) results in constraints that reflect the effect of the assignment on `buf!used!max` and `buf!used!min`. An assignment to an integer `i` results in constraints on `i!max` and `i!min`.

As mentioned above, a function call to a library function that manipulates string buffers is modeled by constraints that summarize the effect of the call. For example, the `strcpy` statement at line `(18)` in Figure 2 results in the following constraints:

copy!used!max $\geq$ buffer!used!max
copy!used!min $\leq$ buffer!used!min

For each user-defined function `foo`, there are constraint variables for `foo`'s formal parameters that are integers or strings. If `foo` returns an integer or a string, then there are constraint variables (e.g., `copy_buffer$return!used!max`) for the function's return value. A call to a user-defined function is modeled with constraints for the passing of actual parameters and the assignment of the function's return value.

As in [33], constraints are associated with pointers to character buffers rather than the character buffers themselves. This means that some aliasing among character buffers is not modeled in the constraints and false negatives may result. We chose to follow [33] in this regard because we are interested in improving precision by using a context sensitive program analysis (Section 5). Currently, context-sensitive pointer analysis does not scale well, and using a context-insensitive pointer analysis would undermine our objective of performing context-sensitive buffer overrun analysis.

However, we discovered that we could make use of pointer analysis to eliminate some false negatives. For instance, consider the statement "`strcpy(p->f, buf)`," where `p` could point to a structure `s`. The constraints generated for this statement would relate the constraint variables for `s.f` and `buf`. Moreover, we use the results of pointer analysis to handle arbitrary levels of dereferencing. Constraint generation also makes use of pointer information for integers.

Figure 3 shows a few constraints for the program in Figure 2, along with the program statement that generated them. Most of the constraints are self-explanatory, however a few comments are in order:

- Since we do not model control flow, we ignore predicates in our constraint generation. Hence we do not model the effect of the predicate in an `if` or `for` statement; the predicate `counter < 10` in line `(8)` was ignored in our example.
- The statement `counter++` is particularly interesting when generating linear constraints. A linear constraint such as counter!max $\geq$ counter!max + 1 cannot be interpreted by a linear program solver. Hence, we model this statement by treating it as a pair of statements: counter$'$ = counter + 1; counter = counter$'$. These two constraints capture the fact that counter has been incremented by 1, and can be translated into constraints that are acceptable to a linear program solver, although the resulting linear program will be *infeasible*. Section 3 discusses these and related issues in detail.
- A program variable that acquires its value from the environment or from user input in an unguarded manner is considered unsafe – for instance, the statement `getenv("PATH")`, which returns the search path, could return an arbitrarily long string. To reflect the fact that the string can be arbitrarily long, we generate constraints getenv$return!used!max $\geq \infty$ and getenv$return!used!min ≤ 0. Similarly, an integer variable `i` accepted as user input gives rise to constraints i!max $\geq \infty$ and i!min $\leq -\infty$

Constraint	Program Statement
header!used!max ≥ 2048	6
header!used!min ≤ 1	6
buffer!used!max ≥ buf!used!max	10 (function call)
buffer!used!min ≤ buf!used!min	10 (function call)
buffer!alloc!max ≥ buf!alloc!max	10 (function call)
buffer!alloc!min ≤ buf!alloc!min	10 (function call)
copy_buffer$return!alloc!max ≥ copy!alloc!max	19
copy_buffer$return!alloc!min ≤ copy!alloc!min	19
copy_buffer$return!used!max ≥ copy!used!max	19
copy_buffer$return!used!min ≥ copy!used!min	19
cc2!used!max ≥ copy_buffer$return!used!max	10 (assignment)
cc2!used!min ≤ copy_buffer$return!used!min	10 (assignment)
cc2!alloc!max ≥ copy_buffer$return!alloc!max	10 (assignment)
cc2!alloc!min ≥ copy_buffer$return!alloc!min	10 (assignment)
counter′!max ≥ counter!max + 1	8 (counter++)
counter!max ≥ counter′!max	8 (counter++)
counter′!min ≤ counter!min + 1	8 (counter++)
counter!min ≤ counter′!min	8 (counter++)

Figure 3: Some constraints for the running example

```
Input: Set of Constraints C
Output: Subset of C with no uninitialized, or infinite variables
    InfSet = {var | var ≤ −∞∨ var ≥ ∞} ∪ {var | var is un-initialized}
    while InfSet ≠ φ
        Select and remove var from InfSet
        foreach Constraint c ∈ C of the form MaxVar ≥ RHS
            if MaxVar is var
                Drop c from C
            else if var appears in RHS
                Set MaxVar to +∞ and add MaxVar to InfSet
                Drop c from C
            endif
        foreach Constraint c ∈ C of the form MinVar ≤ RHS
            if MinVar is var
                Drop c from C
            else if var appears in RHS
                Set MinVar to -∞ and add MinVar to InfSet
                Drop c from C
            endif
    Return C
```

Figure 4: Algorithm for Taint Analysis

2.3 Taint Analysis

The linear constraints then pass through a *taint analysis* module. In Sections 3 and 4 we will demonstrate two techniques to solve the constraints using linear programming. The main goal of the taint analysis module is to make the constraints amenable to these solvers. Linear programming can work only with finite values, hence this requires us to remove variables that can obtain infinite values. Moreover, it is also important that max variables have finite lower bounds and min variables have finite upper bounds. Hence, the objectives of this module are twofold:

• *Identify and remove any variables that get an infinite value*: As mentioned in section 2.2, some constraint variables var are associated with constraints of the form var $\geq \infty$ or var $\leq -\infty$. Taint analysis identifies constraint variables that can directly or indirectly be set to $\pm\infty$ through such constraints and removes them from the set of constraints.

• *Identify and remove any uninitialized constraint variables*: The system of constraints is examined to see if all max constraint variables have a finite lower bound, and all min constraint variables have a finite upper bound; we refer to constraint variables that do not satisfy this requirement as *uninitialized*. Constraint variables may fail to satisfy the above requirement if either the program variables that they correspond to have not been initialized in the source code, or program statements that affect the value of the program variables have not been captured by the constraint generator. The latter case may arise when the constraint generator does not have a model for a library function that affects the value of the program variable. It is important to realize that this analysis is not meant to capture uninitialized *program* variables, but is meant to capture uninitialized *constraint* variables.

Figure 4 presents the taint analysis algorithm. In the constraints obtained by the program in Figure 2, no variables will be removed by the taint analysis module, assuming that we modeled the library functions `strlen`, `fgets` and `strcpy` correctly.

2.4 Constraint Solving

The constraints that remain after taint analysis can be solved using linear programming. We have developed two solvers, both of which use linear programming to obtain values for the constraint variables. The first method uses a linear program solver on the entire set of constraints to obtain values for constraint variables; a detailed description of the algorithm can be found in Section 3. The second method analyzes and breaks up the set of constraints into smaller subsets, and passes each of these subsets to the linear program solver; we explain this algorithm in Section 4.

The goal of both solvers is the same, to obtain the best possible estimate of the number of bytes used and allocated for each buffer in any execution of the program. For a buffer pointed to by `buf`, finding the number of bytes used corresponds to finding the "tightest" possible range [`buf!used!min`..`buf!used!max`]. This can be done by finding the lowest and highest values of the constraint variables `buf!used!max` and `buf!used!min` respectively that satisfy all the constraints. Similarly, we can find the "tightest" possible range for the number of bytes allocated for the buffer by finding the lowest and the highest values of `buf!alloc!max` and `buf!alloc!min` respectively.

For the program in Figure 2, the constraint variables take on the values shown in Figure 5; We explain in detail in Sections 3 and 4 how these values were obtained.

Variable	`min` Value	`max` Value
`header!used`	1	2048
`header!alloc`	2048	2048
`buf!used`	1	1024
`buf!alloc`	1024	1024
`cc1!used`	0	2048
`cc1!alloc`	0	2047
`ptr!used`	1	2048
`ptr!alloc`	1024	2048
`cc2!used`	0	2048
`cc2!alloc`	0	2047
`buffer!used`	1	2048
`buffer!alloc`	1024	2048
`copy!used`	0	2048
`copy!alloc`	0	2047
`counter`	0	∞

Figure 5: Values of some constraint variables

2.5 Detecting Overruns

Based on the values inferred by the solver, as well as the values inferred by the taint analysis module, the detector decides whether there was an overrun on each buffer. We use several heuristics to give the best possible judgment. We shall explain some of these in the context of the values from Figure 5.

• The solver found that the buffer pointed to by `header` has 2048 bytes allocated for it, but that its length could have been between 1 and 2048 bytes. This is a scenario where a buffer overrun can never occur – and hence the buffer pointed to by `header` is flagged as "safe". The same is true of the buffer pointed to by `buf`.

• The buffer pointed to by `ptr` was found to have between 1024 and 2048 bytes allocated, while between 1 and 2048 bytes could have been used. Note that `ptr` is part of two assignment statements. The assignment statement `(6)` could make `ptr` point to a buffer as long as 2048 bytes, while the statement `(9)` could make `ptr` point to a buffer as long as 1024 bytes. The flow insensitivity of the analysis means that we do not differentiate between these program points, and hence can only infer that `ptr` was up to 2048 bytes long. In such a scenario, where the value of `ptr!used!max` is bigger than `ptr!alloc!min` but smaller than (or equal to) the value of `ptr!alloc!max`, we conservatively conclude that there might have been an overrun. This can result in a *false positive* due to the flow insensitivity of the analysis.

• In cases such as for program variable `copy` where we observe that `copy!alloc!max` is less than `copy!used!max`, we know that there is a run of the program in which more bytes were written into the buffer than it could possible hold, and we conclude that there was an overrun on the buffer.

Notice that the constraint variables corresponding to `cc1` and `cc2` get the same value; this is a result of the context-insensitivity of our analysis. We will show in Section 5 how to enhance the precision of the analysis using context sensitivity.

We have developed a GUI front end (Figure 11) that enables the end-user to "surf" the warnings – every warning is linked back to the source code line that it refers to. Moreover, the user can exploit the program slicing capabilities of Codesurfer to verify real overruns.

3 Constraint Resolution using Linear Programming

A Linear Program is an optimization problem that is expressed as follows:

$$\begin{aligned}\text{Minimize} &: \texttt{cx}\\ \text{Subject To} &: \texttt{Ax} \geq \texttt{b}\end{aligned}$$

where A is an $m \times n$ matrix of constants, b and c are vectors of constants, and x is a vector of variables. This is equivalent to saying that we have a system of m inequalities in n variables, and are required to find values for the variables such that all the constraints in the system are satisfied and the *objective function* `cx` takes its lowest possible value. It is important to note that the above form is just one of the numerous ways in which a linear program can be expressed. For a more comprehensive view of linear programming, see [11, 30]. Linear programming works on finite real numbers; that is, the variables in the vector x are only allowed to take finite real values. Hence the optimum value of the objective function, if it exists, is always guaranteed to be finite.

Linear programming is well studied in the literature, and there are well-known techniques to solve linear programs, Simplex [16] being the most popular of them. Other known techniques, such interior point methods [35] work provably in polynomial time. Commercially available solvers for solving linear programs, such as SoPlex [36, 37] and CPLEX [26] implement these and related methods.

The set of constraints that we obtained after program analysis are linear constraints, hence we can formulate our problem as a linear program. Our goal is to obtain the values for `buf!alloc!min`, `buf!alloc!max`, `buf!used!min` and `buf!used!max` that yield the tightest possible ranges for the number of bytes allocated and used by the buffer pointed to by `buf` in such a way that all the constraints are satisfied. More precisely, we are interested in finding the lowest possible values of `buf!alloc!max` and `buf!used!max`, and the highest possible values of `buf!alloc!min` and `buf!used!min` subject to the set of constraints. We can obtain the desired bounds for each buffer `buf` by solving four linear programs, each with the same constraints but with different objective functions:

Minimize: `buf!alloc!max`
Maximize: `buf!alloc!min`
Minimize: `buf!used!max`
Maximize: `buf!used!min`

However, it can be shown (the proof is beyond the scope of this paper) that for the kind of constraints generated by the tool, if all `max` variables have finite lower bounds, and all `min` variables have finite upper bounds, then the values obtained by solving the four linear programs as above are also the values that optimize the linear program with the same set of constraints subject to the objective function:

Minimize: $\sum_{\texttt{buf}}$ (`buf!alloc!max` - `buf!alloc!min` + `buf!used!max` - `buf!used!min`)

Note that this objective function combines the constraint variables across *all* buffers. Since taint analysis ensures that all `max` variables have finite lower bounds and all `min` variables have finite upper bounds, we can solve just *one* linear program, and obtain the bounds for *all* buffers.

It must be noted that we are actually interested in obtaining integer values that represent buffer bounds `buf!alloc!max`, `buf!used!max`, `buf!alloc!min` and `buf!used!min`. The problem of finding integer solutions to a linear program is called Integer Linear Programming and is a well known NP-complete problem [18]. Our approach is thus an approximation to the real problem of finding *integer* solutions that satisfy the constraints. In some cases, however, it is possible to solve the problem using standard linear programming algorithms and yet obtain integer solutions to the variables in the linear program. This is possible when the constraints can be expressed as $\texttt{A}\cdot\texttt{x} \geq \texttt{b}$, and A is a *unimodular* matrix [5, 19, 30, 32]. Here A is an $m \times n$ matrix of integer constants, x is an $n \times 1$ vector of variables, and b is an $m \times 1$ vector of integer constants. In our experience, the constraints produced by the tool have always produced integer solutions.

3.1 Handling Infeasible Linear Programs

While at first glance the method seems to give the desired buffer bounds, it does not work for all cases. In particular, an optimal solution to a linear program need not even exist. We describe briefly the problems faced when using a linear programming based approach for determining the buffer bounds.

A linear program is said to be *feasible* if one can find finite values for all the variables such that all the constraints are satisfied. For a linear program in n variables, such an assignment is a vector in $\mathbb{R}^n$ and is called a solution to the linear program. A solution is said to be *optimal* if it also maximizes (or minimizes) the value of the objective function. A linear program is said to be *unbounded* if a solution exists, but no solution optimizes the objective function. For instance, consider:

$$\begin{aligned} \text{Maximize} &: \ \mathtt{x} \\ \text{Subject To} &: \ \mathtt{x} \geq 5 \end{aligned}$$

Any value of $\mathtt{x} \geq 5$ is a solution to the above linear program, but no finite value $\mathtt{x} \in \mathbb{R}$ optimizes the objective function. Finally, a linear program is said to be *infeasible* if it has no solutions. An example of an infeasible linear program is shown in Figure 6.

$$\begin{aligned} \text{Minimize} &: \ \mathtt{counter!max} \\ \text{Subject To} &: \ \mathtt{counter'!max} \geq \mathtt{counter!max} + \mathtt{1} \\ & \ \mathtt{counter!max} \geq \mathtt{counter'!max} \end{aligned}$$

Figure 6: An Infeasible Linear Program

In our formulation, if a linear program has an optimal solution, we can use that value as the buffer bound. None of the linear programs in our case can be unbounded, since the constraints have been examined by the taint analyzer to ensure that all `max` variables have finite lower bounds. We minimize for the `max` variables in the objective function, and since all the `max` variables have finite lower bounds, the lowest value that each `max` variable can obtain is also finite. Similarly, all `min` variables have finite upper bounds, and so when we maximize the `min` variables, the highest values that they could obtain are also finite. *Hence taint analysis is an essential step to ensure that our approach works correctly.*

However, when the linear program is infeasible, we cannot assign any finite values to the variables to get a feasible solution. As a result, we cannot obtain the values for the buffer bounds. In such a case, a safe option would be to set all `max` variables to ∞ and `min` variables to -∞, but that information would be virtually useless to the user of the tool because there would be too many false alarms. The linear program may be infeasible due to a small subset of constraints; in such a scenario, setting all variables to infinite values will be overly conservative. For instance, the constraints in Figure 2 are infeasible because of the constraints generated for the statement `counter++`. Constraints generated by most real world programs have such statements, as well as statements involving pointer arithmetic, and we can expect the constraints for such programs to be infeasible. Thus, the conservative approach of setting all constraint variables to infinite values is unacceptable.

We have developed an approach in which we try to remove a "small" subset of the original set of constraints so that the resultant constraint system is feasible. In fact, the problem of "correcting" infeasible linear programs to make them feasible is a well studied problem in the operations research community. The approach is to identify *Irreducibly Inconsistent Sets* (called *IIS*) [9]. An IIS is a minimal set of inconsistent constraints, i.e., the constraints in the IIS together are infeasible, but any subset of constraints in the IIS form a feasible set. For instance, both the constraints in the linear program in Figure 6 constitute an IIS because the removal of any one of the two constraints makes the linear program feasible. There are several efficient algorithms available to detect IISs in a set of constraints. We used the *Elastic Filtering algorithm* described in [9]. The Elastic Filtering Algorithm takes as input a set of linear constraints and identifies an IIS in these constraints (if one exists). An infeasible linear program may have more than one IISs in it, and the elastic filtering algorithm is guaranteed to find at least one of these IISs. To produce a feasible linear program from an infeasible linear program, we may be required to run the elastic filtering algorithm several times; each run identifies and removes an IIS and produces a smaller linear program which can further be examined for presence of IISs.

Figure 7 pictorially shows our approach to obtain a set of feasible linear constraints from a set of infeasible linear constraints. We first examine the input set, depicted as C, to find out if it is feasible; if so, it does not contain IISs, and C can be used as the set of constraints in our linear program formulation. If the C turns out to be infeasible, then it means that there is a subset of C that forms one or more IISs. This subset is depicted as C' in the figure. The elastic filtering algorithm, over several runs, identifies and removes the subset C' from the set of constraints. The resultant set $C - C'$ is feasible. We then set the values of the `max` and `min` variables appearing in C' to ∞ and -∞ respectively. We do so because we cannot infer the values of these variables using linear programming, and hence setting these variables to infinite values is a conservative approach. These variables whose values are infinite may appear in the set of constraints $C - C'$. The scenario is now similar to taint analysis, where we had some constraint variables whose values were infinite, and we had to identify and remove the constraint variables that were "tainted" by the infinite variables. Therefore, we run steps `(2)-(13)` of the taint analysis algorithm (Figure 4) with `Infset` as the constraint variables that appear in C'. This step results in further removal of constraints, which are depicted in the Figure 7 by a subset C'' of $C - C'$. The set of constraints after removal of C'', denoted as D in Figure 7,

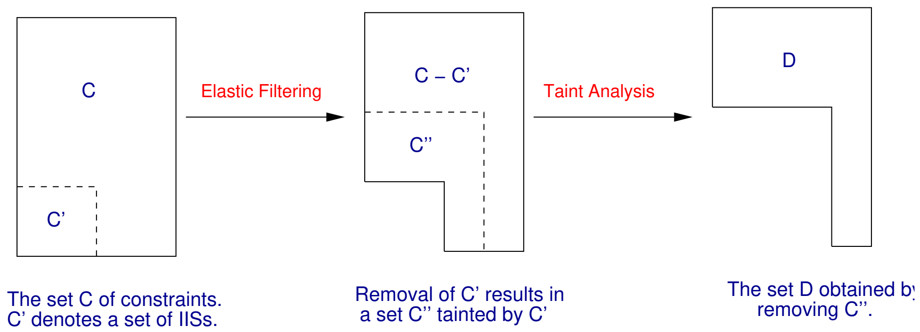

Figure 7: Making an Infeasible set of constraints amenable to Linear Programming

Input: Set of Constraints C
Output: For each buffer `buf`, values for `buf!used!max`, `buf!used!min`, `buf!alloc!max`, `buf!alloc!min`
(1) `Removed` = ϕ
(2) **while** C is infeasible
(3) `IIS_set` = ELASTIC_FILTER_ALGORITHM(C)
(4) C = C - `IIS_Set`
(5) `Removed` = `Removed` $\cup$ `IIS_Set`
(6) **foreach** constraint variable `v` appearing in `Removed`
(7) **if** `v` is a `max` variable
(8) `v` $\leftarrow \infty$
(9) **else**
(10) `v` $\leftarrow -\infty$
(11) C = output of steps `(2)-(12)` of *Taint Analysis* (Figure 4) by setting `Infset` = `Removed`
(12) `MaxSet` = $\{v \mid v$ is a `max` constraint variable appearing in $C\}$
(13) `MinSet` = $\{u \mid u$ is a `min` constraint variable appearing in $C\}$
(14) Minimize: $(\sum_{v \in \texttt{MaxSet}} v)$ - $(\sum_{u \in \texttt{MinSet}} u)$ Subject To: C
(15) Set each variable to the value returned by the Linear Program Solver.

Figure 8: Constraint Resolution using Linear Programming

satisfies the property that all `max` variables appearing in it have finite lower bounds, and all `min` variables have finite upper bounds. Moreover, D is feasible, and will only yield optimal solutions when solved as a linear program with the objective functions described earlier. Hence, we solve the linear program using the set of constraints in D.

Figure 8 summarizes our approach to constraint resolution using linear programming. Steps `(1)–(10)` of the algorithm describe the transformation that removes the IISs, while step `(11)` performs the taint analysis to obtain the set of constraints which can be used in the linear program formulation.

3.2 Implementation

We have implemented the above algorithm by extending the commercially available package SoPlex [36, 37]. SoPlex is a linear program solver; we extended it by adding IIS detection and taint analysis. In practice, linear program solvers work much faster when the constraints have been *presolved*. Presolving is a method by which constraints are simplified *before* being passed to the solver. In several cases, we can make simple inferences about the constraints; for instance, if `x` $\geq$ 5 is the only constraint involving `x`, and we wish to minimize `x`, it is clear that `x` is 5. Several such techniques are described in the literature [7]; we have incorporated some of them in our solver.

4 Solving Constraint Systems Hierarchically

In the previous section, we described an approach that used linear programming to determine bounds on the constraint variables. When the linear program was infeasible, we detected and removed IISs and solved a feasible subset of the constraints. In this section, we present an alternate approach for solving a set of constraints that handles infeasible sets of constraints in a different way. This approach was also developed independently by Rugina and Rinard in [29]. The idea behind this approach is to decompose the set of constraints into smaller subsets, and solve each subset separately. We do so by constructing a directed acyclic graph (DAG), each of whose members is a set of constraints, and solve each member in the order that it appears in a topological sort of the DAG.

To construct such a DAG, we first identify sets of con-

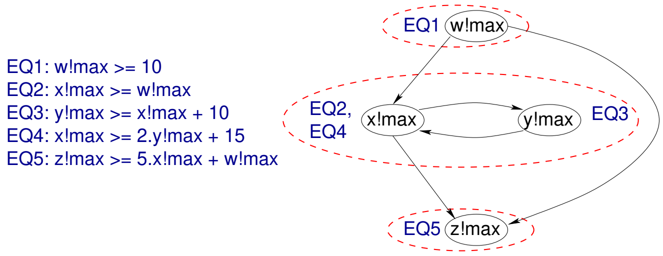

Figure 9: Constraint Dependency Graph – an example

straints such that each member of the set depends on the other members of the set either directly or indirectly. Consider for instance, the constraints shown on the left in Figure 9. Constraint `EQ3` gives a lower bound for the variable `y!max` based on the value of `x!max`. However the value of `x!max` itself has a lower bound determined by `EQ2` and `EQ4`. Thus `EQ3` "depends" on `EQ2` and `EQ4`.

To formalize the notion of dependency, we construct a graph whose vertices are the constraint variables in the set of equations. We associate the vertex corresponding to a variable `x` with all constraints in which `x` appears on the LHS. We draw an edge from a vertex `y` to a vertex `x` if there is a constraint that has `y` on the RHS and `x` on the LHS. We then identify *Strongly Connected Components (SCCs)* in this graph. The set of constraints associated with the vertices in an SCC are defined to be dependent upon each other. Figure 9 shows the constraints associated with each vertex; the SCCs are identified using dotted lines. `EQ2`, `EQ3` and `EQ4` are dependent on each other.

Recall that if we coalesce the SCCs in a graph, then the resulting graph is a DAG. The topological sort of the DAG naturally defines a hierarchy in the DAG. Hence, we consider each SCC in topologically sorted order, and solve the constraints associated with that SCC. Each SCC consists of a set of linear constraints, and we formulate a linear program to minimize (maximize) each `max` (`min`) variable that appears in the set of constraints just as we did in Section 3.

If the set of constraints C in an SCC are found to be infeasible, we can immediately set all `max` and `min` variables appearing on the LHS of each constraint in C to ∞ and -∞ respectively. This approach does not require us to identify and remove IISs in C. This is because an IIS detection algorithm combined with the taint analysis that follows IIS detection, denoted by steps `(1)-(11)` of Figure 8, would remove all the constraints in C and set the variables appearing on the LHS in each constraint in C to infinite values. This can be attributed to the fact that each constraint in C is dependent on at least one more constraint in C; consequently, setting any LHS variable to an infinite value will result in the LHS variables of all constraints getting infinite values. Hence, the approach of solving a set of dependent constraints together obviates the need for IIS detection and elimination.

Once we have the values for the constraint variables that appear in an SCC, we can substitute these values in the constraints that are associated with the children of the SCC. Once all the SCCs have been solved, the values for all the constraint variables in the set of constraints becomes available.

A few points are worth noting with respect to this solver:

• Constraint simplification by substituting available values presents an opportunity to avoid calling an LP solver if the simplification makes the constraints amenable to presolve. For instance, for the set of constraints shown in Figure 9, we can infer that the value of `w!max` is 10 without having to invoke a linear program solver. This value can be substituted in `EQ2` and `EQ5`, thus simplifying these constraints. Similarly, we can infer the value of `z!max` once the value of `x!max` is available.

• The IIS detection based approach for handling infeasibility is an approximation algorithm. It may remove more constraints than are actually required to make the constraints feasible; as a result more constraint variables than necessary may be set to ∞/-∞. It can be shown that the solution obtained by the hierarchical solver is precise, in the sense that it sets the fewest number of constraint variables to ∞/-∞. Furthermore, when the linear program is feasible, this solver produces the same solution as obtained by the algorithm in Figure 8. This gives rise to a trade-off, i.e., the user can choose between the hierarchical solver which solves more (but smaller) linear programs, the solutions to which are mathematically precise, or choose the algorithm from Figure 8, which may be imprecise, but is more efficient. In our experiments, we noted that the approach from Section 3 can be up to 3 times faster than the hierarchical solver, while sacrificing the precision of only 5% of the constraint variables.

• Since we have broken down the problem into one of solving small sets of constraints, we could use a different solver for each set of constraints. Some kinds of constraint systems have fast solvers, for instance, the problem of finding a solution to a set of difference constraints can be formulated as a shortest-path problem [13].

• Lastly, for very large constraint systems, one could envision solving the SCCs at the same depth in parallel. Thus, a DAG with depth D can be solved in D steps.

5 Adding Context Sensitivity

The constraint generation process described in Section 2 was context-insensitive. When we generated the constraints for a function, we considered each call-site as an assignment of the actual-in variables to the formal-in variables, and the return from the function as an assignment of the formal-out variables to the actual-out variables. As a result, we merged information across call-sites, thus making the analysis imprecise. In this section we describe how to incorporate context sensitivity.

Constraint inlining is similar in spirit to inlining function bodies at call-sites. Observe that in the context-insensitive approach, we lost precision because we treated *different* call-sites to a function identically, i.e, by assigning the actual-in variables at each call-site to the *same* formal parameter.

Constraint inlining alleviates this problem by creating a fresh instance of the constraints of the called function at each call-site. In other words, at each call-site to a function, we produce the constraints for the called function with the local variables and formal variables renamed uniquely for that call-site. This is illustrated in the example below, which shows some of the constraints for the function `copy_buffer` from Figure 2 specialized for the call-site at line `(7)`:

```
copy!alloc!max1 ≥ buffer!used!max1 - 1
copy!used!max1 ≥ buffer!used!max1
copy!used!min1 ≤ buffer!used!min1
copy_buffer$return!used!max1 ≥ copy!used!max1
copy_buffer$return!used!min1 ≤ copy!used!min1
```

Context-sensitivity can be obtained by modeling each call-site to the function as a set of assignments to the renamed instances of the formal variables. The actual-in variables are assigned to the *renamed* formal-in variables, and the *renamed* formal-out variables are assigned to the actual-out variables. As a result, there is exactly one assignment to each renamed formal-in parameter of the function, which alleviates the problem of merging information across different calls to the same function.

Some of the constraints for the call-site to `copy_buffer` at line `(7)` in Figure 2 are shown below:

```
buffer!used!max1 ≥ header!used!max
buffer!used!min1 ≤ header!used!min
cc1!used!max ≥ copy_buffer$return!used!max1
cc1!used!min ≤ copy_buffer$return!used!min1
```

With this approach to constraint generation, we obtain the values 2047 and 2048 for `cc1!alloc!max` and `cc1!used!max` respectively, while `cc2!alloc!max` and `cc2!used!max` get the values 1023 and 1024 respectively, which is an improvement over the values reported in Figure 5.

We have implemented this approach because it requires only minimal changes to the constraint generation process that we have already described. However, it also has some shortcomings:

- It does not handle recursive function calls; this is attributed to the fact that inlining cannot work in the presence of recursion.
- The number of constraint variables in the constraints with context sensitivity may be exponentially larger than the number of constraints in their non-context sensitive counterpart. As a result, we do not expect this technique to scale well to large programs.

These drawbacks can be overcome through the use of *summary constraints*. Summary constraints summarize the effect of a function call in terms of the constraint variables representing global variables and formal parameters of the called function. Once the summary constraints of a function are available, we can obtain context sensitivity by substituting actual parameters in place of the formal parameters in the summary constraints. This approach is described in detail in Appendix A.1.

6 Experience with the tool

We tested our prototype implementation on several popular commercially used programs. In each case, the tool produced several warnings; we used these warnings, combined with Codesurfer features such as slicing, to check for real overruns. We tested to see if the tool discovered known overruns documented in public databases such as `bugtraq` [1] and CERT [2], and also checked to see if any overruns that were previously unreported were discovered. We report our experience with `wu-ftpd` and `sendmail`. Results on a few more packages are in Appendix A.2.

All our experiments were performed on a machine with a 3GHz P4 Xeon processor machine with 4GB RAM running Debian GNU/Linux 3.0. We used Codesurfer version 1.8 for our experiments, the `gcc-3.2.1` compiler for building the programs, and `glibc` version 2.2.4 for macro-expansion. Codesurfer implements several pointer analysis algorithms; in each case we performed the experiments with a field-sensitive version of Andersen's analysis [6] that uses the common-initial-prefix technique of Yong and Horwitz [39] to deal with structure casts. We configured the tool to use the hierarchical solver described in Section 4 for constraint resolution (so the values obtained will be precise), and produce constraints in a context-insensitive fashion.

6.1 WU-FTP Daemon

We tested two versions of the wu-ftp daemon, a popular file transfer server. Version 2.5.0 is an older version with several known vulnerabilities (see CERT advisories CA-1999-13, CA-2001-07 and CA-2001-33), while version 2.6.2 is the current version with several security patches that address the known vulnerabilities.

6.1.1 wu-ftpd-2.6.2

wu-ftpd-2.6.2 has about 18K lines of code, and produced 178 warnings when examined by our tool. Upon examining the warnings, we found 14 previously unreported overruns; we will describe one of these in detail.

The tool reported a potential overrun on a buffer pointed to by accesspath in the procedure read_servers_line in rdservers.c, where as many as 8192 bytes could be copied into the buffer for which up to 4095 bytes were allocated. Figure 10 shows the code snippet from read_servers_line which is responsible for the overrun.

```
int read_servers_line (FILE *svrfp,
                       char *hostaddress,
                       char *accesspath){
    static char buffer[BUFSIZ];
    ...
    while (fgets(buffer, BUFSIZ, svrfp)){
      ...
      if ((hp = gethostbyname(hcp))){
        struct in_addr in;
        memmove(&in, hp->h_addr, sizeof(in));
        strcpy(hostaddress, inet_ntoa(in));
      }
      else
        strcpy(hostaddress, hcp);

      strcpy(accesspath, acp);
    }
}
```

Figure 10: Code snippet from wu-ftpd-2.6.2

The fgets statement may copy as many as 8192 (BUFSIZ) bytes into buffer, which is processed further in this function. As a result of this processing, acp and hcp point to locations inside buffer. By an appropriate choice of the contents of buffer, one could make acp or hcp point to a string buffer as long as 8190 bytes, which could result in an overflow on the buffer pointed to either by accesspath or hostname respectively.

The procedure read_servers_line is called at several places in the code. For instance, it is called in the main procedure in ftprestart.c where read_servers_line is called with two local buffers, hostaddress and configdir, which have been allocated 32 bytes and 4095 bytes respectively. This call reads the contents of the file _PATH_FTPSERVERS, which typically has privileged access. However, in non-standard and unusual configurations of the system, _PATH_FTPSERVERS could be written to by a local user. As a result, the buffers hostaddress and configdir can overflow based on a carefully chosen input string, possibly leading to a local exploit. The use of a strncpy or strlcpy statement instead of the unsafe strcpy in read_servers_line rectifies the problem.

Some other new overruns which were detected by the tool were:

- An unchecked sprintf in main in the file ftprestart.c could result in 16383 bytes being written into a local buffer that was allocated 4095 bytes.
- Another unchecked sprintf in main in the file ftprestart.c could result in 8447 bytes being written into a local buffer that was allocated 4095 bytes.
- An unchecked strcpy in main in the file ftprestart.c could result in 8192 bytes being written into a local buffer that was allocated 4095 bytes.

In each of the above cases, a carefully chosen string in the file _PATH_FTPACCESS can be used to cause the overrun. As before, _PATH_FTPACCESS typically has privileged access, but could be written to by a local user in non-standard configurations. We contacted the wu-ftpd developers [22], and they have acknowledged the presence of these bugs in their code, and are in the process of fixing the bugs (at the time of writing this paper).

6.1.2 wu-ftpd-2.5.0

wu-ftpd-2.5.0 has about 16K lines of code; when analyzed by our tool, it produced 139 warnings. Figure 11 shows a screenshot of the GUI that our tool provides for the user to surf the warnings. Each of the warnings shown is a "hot" link, and is linked back to the line of source code that is responsible for the warning. Consider the first warning shown in the figure; it depicts that the tool found a potential overrun on the buffer buf in the procedure vreply in the file ftpd.c. It also shows two possible locations where it thinks the overrun could have occurred – an snprintf, and an sprintf statement.

The first location where a potential overrun was found, an snprintf, was:

```
snprintf(buf + (n ? 4 : 0),
         n ? (sizeof(buf)-4) : sizeof(buf),
         "%s", fmt);
```

Clearly, no more than sizeof(buf) bytes are written into buf, and hence this statement is safe. However since the tool ignores control flow, this statement is mod-

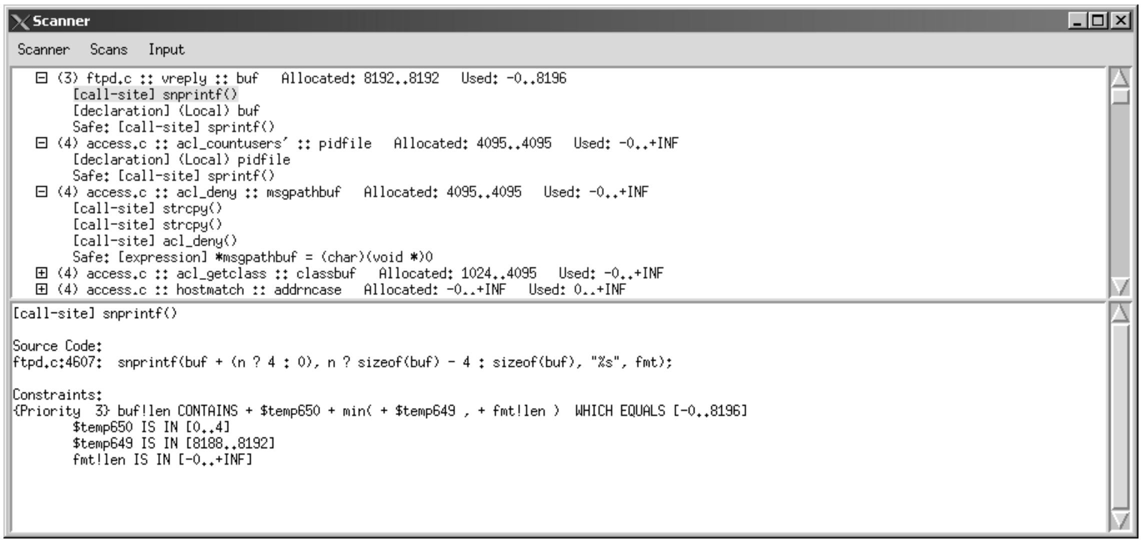

Figure 11: A screenshot from the `wu-ftpd-2.5.0` analysis

eled as though `sizeof(buf)` bytes could be written at the location `buf + 4`, which causes the tool to report that as many as 8196 bytes could be written into `buf` for which 8192 bytes where allocated. As a result, this warning is a false alarm. The second location associated with this warning, an `sprintf` statement, turns out to be safe since it copies only 16 bytes into the 8192 byte array `buf`. The tool inferred this from the constraints, and hence this statement was marked "safe" as is shown in the figure.

We analyzed the warnings to check for a widely exploited overrun reported in CERT advisory CA-1999-13. The buffer overflow was on a globally declared buffer `mapped_path` in the procedure `do_elem` in the file `ftpd.c`. It was noted in [23] that the overrun was due to a statement `strcat(mapped_path, dir)`, where the variable `dir` could be derived (indirectly) from user input. As a result it was possible to overflow `mapped_path` for which 4095 bytes were allocated. Our tool reported the range for `mapped_path!used` as [0..+∞], while `mapped_path!alloc` was [4095..4095]. We note that `strcat(dst, src)` would be modeled as four linear constraints by our tool:

$$\boxed{\begin{array}{l} \texttt{dst}'\texttt{!used!max} \geq \texttt{dst!used!max} + \texttt{src!used!max} \\ \texttt{dst!used!max} \geq \texttt{dst}'\texttt{!used!max} \\ \texttt{dst}'\texttt{!used!min} \leq \texttt{dst!used!min} + \texttt{src!used!min} \\ \texttt{dst!used!min} \leq \texttt{dst}'\texttt{!used!min} \end{array}}$$

The first two constraints make the linear program infeasible, as explained in Section 3, and result in `dst!used!max` being set to +∞. Hence, in `wu-ftpd-2.5.0`, `mapped_path!used!max` will be set to +∞, and the tool would have reported the same range even in the absence of an overrun. We used Codesurfer's program slicing feature to confirm that `dir` could be derived from user input. We found that the procedure `do_elem`, one of whose parameters is `dir`, was called from the procedure `mapping_chdir`. This function was in turn called from the procedure `cmd`, whose input arguments could be controlled by the user. This shows the importance of providing the end user with several program analysis features. These features, such as program slicing and control and data dependence predecessors, which are part of Codesurfer, aid the user of the tool to understand the source code better and hence locate the source of the vulnerability.

6.2 Sendmail

Sendmail is a very popular mail transfer agent. We analyzed `sendmail-8.7.6`, an old version that was released after a thorough code audit of version `8.7.5`. However, this version has several known vulnerabilities. We also analyzed `sendmail-8.11.6`; in March 2003, two new buffer overrun vulnerabilities were reported in the then current `sendmail` version. We note that `sendmail-8.7.6` and `sendmail-8.11.6` are vulnerable to these overruns as well.

6.2.1 `sendmail-8.7.6`

`sendmail-8.7.6` has about 38K lines of code; when analyzed by our tool, it produced 295 warnings. Due to the large number of warnings, we focused on scanning the warnings to detect some known overruns.

Wagner *et al.* use BOON [34] to report an off-by-one bug in `sendmail-8.9.3` where as many as 21 bytes, returned by a function queuename, could be written into a 20 byte array dfname. Our tool identified four possible program points in sendmail-8.7.6 where the return value from queuename is copied using strcpy statements into buffers which are allocated 20 bytes. As in [34], our tool reported that the return value from queuename could be up to 257 bytes long, and further manual analysis was required to decipher that this was in fact an off-by-one bug. Another minor off-by-one bug was reported by the tool where the programmer mistakenly allocated only 3 bytes for the buffer delimbuf which stored "\n\t ", which is 4 bytes long including the end of string character.

6.2.2 sendmail-8.11.6

sendmail-8.11.6 is significantly larger than version 8.7.6 and has 68K lines of code; when we ran our tool, it produced 453 warnings. We examined the warnings to check if the tool discovered the new vulnerabilities reported in March 2003.

One of these vulnerabilities is on a function crackaddr in the file headers.c, which parses an incoming e-mail address string. This function stores the address string in a local static buffer called buf that is declared to be MAXNAME + 1 bytes long, and performs several boundary condition checks to see that buf does not overflow. However, the condition that handles the angle brackets (<>) in the From address string is imprecise, thus leading to the overflow [25].

Our tool reported that bp, a pointer to the buffer buf in the function had bp!alloc!max = $+\infty$ and bp!used!max = $+\infty$, thus raising an warning. However, the reason bp!alloc!max and bp!used!max were set to $+\infty$ was because of several pointer arithmetic statements in the body of the function. In particular, the statement bp-- resulted in bp!alloc!max = $+\infty$ and bp!used!max = $+\infty$. Hence, this warning would have existed even if the boundary condition checks were correct.

We have discovered that the use of control dependence information, which associates each statement with the predicate that decides whether the statement will be executed, is crucial to detecting such overruns reliably.

6.3 Performance

Figure 12 contains representative numbers from our experiments with wu-ftpd-2.6.2 and sendmail-8.7.6. All timings are wall-clock times, and are an average over 4 runs; CODESURFER denotes the time taken by Codesurfer to analyze the program, GENERATOR denotes the time taken for constraint generation, while TAINT denotes the time taken for taint analysis. The constraints produced can be resolved in one of two ways; the rows LPSOLVE and HIERSOLVE report the time taken by the solvers from Section 3 and Section 4 respectively. The number of constraints output by the constraint generator is reported in the row PRE-TAINT, while POST-TAINT denotes the number of constraints after taint-analysis.

	wu-ftpd-2.6.2	sendmail-8.7.6
CODESURFER	12.54 sec	30.09 sec
GENERATOR	74.88 sec	266.39 sec
TAINT	9.32 sec	28.66 sec
LPSOLVE	3.81 sec	13.10 sec
HIERSOLVE	10.08 sec	25.82 sec
Number of Constraints Generated		
PRE-TAINT	22008	104162
POST-TAINT	14972	24343

Figure 12: Performance of the tool

These results serve to demonstrate the trade-off between performance and precision of the Hierarchical solver versus the IIS detection based solver from Section 3. While the IIS detection based approach is much faster, it is not mathematically precise. However, we found that it is a good approximation to the solution obtained by the hierarchical solver. In case of wu-ftpd-2.6.2 fewer than 5% of the constraint variables, and in the case of sendmail-8.7.6 fewer than 2.25% of the constraint variables obtained imprecise values when we used the IIS detection based approach.

6.4 Adding Context Sensitivity

We report here our experience with using context-sensitive analysis on wuftpd-2.6.2 using both the constraint inlining approach and the summary constraints approach. To measure the effectiveness of each approach, we will count the number of range variables that were refined in comparison to the corresponding ranges obtained in a context-insensitive analysis. Recall that the value of a range variable var is given by the corresponding constraint variables var!min and var!max as [var!min..var!max]. We chose this metric since, as explained in Section 2.5, the detector uses the values of the ranges to produce diagnostic information, and more precise ranges will more precise diagnostic information.

The context-insensitive analysis on wuftpd-2.6.2 yields values for 7310 range variables. Using the summary constraints approach, we found that 72 of these range variables obtained more precise values. Note that in this approach the number of constraint variables (and hence the number of range variables) is the same as in the context-insensitive analysis. However, the number of constraints may change, and we observed a 1% increase in the number of constraints. This change can be attributed to the fact that summarization introduces a some constraints (the summaries), and removes some constraints (the old call-site assignment constraints).

The constraint inlining approach, on the other hand, leads to a $5.8\times$ increase in the number of constraints, and a $8.7\times$ increase in the number of constraint variables (and hence the number of range variables). This can be attributed to the fact that the inlining based approach specializes the set of constraints at each callsite. In particular, we observed that the 7310 range variables from the context-insensitive analysis were specialized to 63704 range variables based on calling context. We can count the number of range variables that obtained more precise values in two possible ways:

- Out of the 63704 specialized range variables, 7497 range variables had obtained more precise values than the corresponding unspecialized range variables.
- Out of the 7310 unspecialized range variables, 406 range variables had obtained more precise values in at least one calling context.

As noted earlier, the constraint inlining approach returns more precise information than the summary constraints based approach. To take a concrete example, we consider the program variable `msgcode` (an integer), which is the formal parameter of a function `pr_mesg` in the file `access.c` in `wu-ftpd-2.6.2`. The function `pr_mesg` is called from several places in the code with different values for the parameter `msgcode`. The summary constraints approach results in the value [530..550] for the range variable corresponding to `msgcode`. However, the constraint inlining approach refines these ranges – for instance, it is able to infer that `pr_mesg` is always called with the value 530 from the function `pass` in the file `ftpd.c`.

6.5 Effects of Pointer Analysis

As observed in Section 2, we were able to reduce false negatives through the use of pointer analysis. The tool is capable of handling arbitrary levels of dereferencing. For instance, if `p` points to a pointer to a structure `s`, the pointer analysis algorithms correctly infer this fact. Similarly, if `p` and `q` are of type `char**` (i.e., they point-to pointers to buffers), the constraints for a statement such as `strcpy(*p, *q)` would be correctly modeled in terms of the points-to sets of `p` and `q` (recall that we generated constraints in terms of pointers to buffers rather than buffers themselves).

To observe the benefits of pointer analysis we generated constraints with the pointer analysis algorithms turned off. Since fewer constraints will be generated, we can expect to see fewer warnings; in the absence of these warnings, false negatives may result. We observed a concrete case of this in the case of `sendmail-8.7.6`. When we generated constraints without including the results of the pointer analysis algorithms, the tool output only 251 warnings (as opposed to 295 warnings). However, this method resulted in the warning on the array `dfname` being suppressed, so the tool missed the off-by-one bug that we described earlier. A closer look at the code in the procedure `queuename` revealed that in the absence of points-to facts, the tool failed to generate constraints for a statement:

```
snprintf(buf, sizeof buf, "%cf%s",
                                  type, e->e_id)
```

in the body of `queuename` since points to facts for the variable `e`, which is a pointer to a structure, were not generated.

We note that BOON [34] identified this off-by-one bug because of a simple assumption made to model the effect of pointers, i.e., BOON assumes that any pointer to a structure of type `T` can point to all structures of type `T`. While this technique can be effective at discovering bugs, the lack of precise points-to information will lead to a larger number of false alarms.

6.6 Shortcomings

While we found the prototype implementation a useful tool to audit several real world applications, we also noted several shortcomings and are working towards overcoming these limitations.

First, the flow insensitivity of the analysis meant that we would have several false alarms. Through the use of slicing we were able to weed out the false alarms, nevertheless it was a manual and often painstaking procedure. By transitioning to a Static Single Assignment (SSA) representation [15] of the program, we can add a limited form of flow sensitivity to the program. This will result in a large number of constraint variables. Fortunately, we have observed that the solvers readily scale to large linear programs with several thousand variables.

Second, by modeling constraints in terms of pointers to buffers rather than buffers, we can miss overruns, thus leading to false negatives [34]. However, the reason we did so was because the pointer analysis algorithms themselves were flow- and context-insensitive, and generating constraints in terms of buffers would have resulted in a large number of constraints and consequently a large number of false alarms. By transitioning to "better" pointer analysis algorithms we can model constraints in terms of buffers themselves, thus eliminating the false negatives.

7 Related Work

Several static techniques have been proposed to mitigate the problem of buffer overruns. Wagner *et al.* [33, 34] have proposed a tool, BOON, similar in spirit to ours to detect buffer overruns in C source code. However, unlike our tool, BOON does not employ pointer analysis algorithms and does not provide a way to enhance the context-sensitivity of the analysis. Larochelle and Evans [23] propose an extension to LCLint that uses semantic information from annotations in the program to make inferences on buffer bounds. The tool works like a compiler and produces warnings by making inferences based on the annotations. Xi and Pfenning [38] propose an extension to ML that supports type annotations. These annotations are then used to determine the type safety of the programs. However, in both these techniques, the onus is on the user to provide correct annotations. As a result, analyzing large legacy applications without these annotations becomes almost impossible. Dor *et al.* [17] propose a tool (CSSV) that aims to find all buffer overflows with just a few false alarms. The basic idea is to convert the C program into an integer program, and use a conservative static analysis algorithm that can check the correctness of integer manipulations. The analysis is performed on a per-procedure basis; to extend the analysis interprocedurally, they use the concept of *contracts*, which are a set of pre-conditions and post-conditions of a procedure, along with side-effect information. The number of false alarms generated depends on the accuracy of the contracts, which are typically provided by the user. They also discuss techniques whereby conservative user supplied contracts can be automatically refined. Rugina and Rinard [29] propose a technique based on linear programming that infers symbolic upper and lower bounds on arrays. They deal with infeasible linear programs by using a solver similar to the hierarchical solver approach presented in Section 4. They use a flow and context sensitive program analysis to detect several programming errors such as array out-of-bounds errors and race conditions. However, the techniques in [17, 29] have not been tested on large programs, and may scale poorly. For instance, CSSV took > 200 seconds to analyze a string manipulation program with a total of about 400 lines of code.

There are a suite of dynamic techniques that help prevent stack-smashing attacks. Stackguard [14] detects changes to the return address by placing a "canary" word on the stack. RAD [10] defends the return address by storing it in a repository and checking the return address against the repository before the function returns. Both these techniques enhance the compiler to insert function prologues and epilogues that perform the checking. Prasad and Chiueh [27] propose a binary rewriting technique that enhances binaries by incorporating the RAD mechanism; however their technique suffers from imprecision while disassembling the binary. While these methods help in *detecting* and *preventing* attacks based on buffer overruns, they fail to *eliminate* the buffer overflows from the source code, which is the goal around which our tool is built.

Static techniques have also been used to reduce the overhead of run-time checks. CCured [12, 24] is a program transformation system that adds memory safety guarantees to C programs by statically analyzing the source code and classifying pointers as safe or unsafe. Appropriate run-time checks are then inserted depending on the kind of the pointer (lightweight checks for safe pointers). CCured has been applied to several commercial applications with reasonable run-time overhead [12]. However, in some cases, such as systems software, the overhead of CCured could be as high as 87%. Bodik *et al.* present ABCD [8], which provides a way to eliminate frequently executed but redundant array bounds checks for Java programs. This technique assumes the presence of the run-time checks in the code, and suggests a way to improve performance by removing the unwanted checks.

8 Conclusions

We have demonstrated a scalable technique to analyze C source code to detect buffer overrun vulnerabilities. We have shown the efficacy of the technique by applying it to real world examples and identifying new vulnerabilities in a popular security critical package. Our techniques use novel ideas from the linear programming literature, and provide a way to enhance the context sensitivity of the program analysis. The output of our tool, coupled with other program understanding features of Codesurfer, such as static slicing, aid the user to comprehend and eliminate bugs from source code.

9 Acknowledgements

This work was supported in part by the National Science Foundation under grant CCR-9619219, by the Office of

Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

References

[1] bugtraq. www.securityfocus.com.

[2] CERT/CC advisories. www.cert.org/advisories.

[3] The twenty most critical internet security vulnerabilities. www.sans.org/top20.

[4] Aleph-one. smashing the stack for fun and profit. Nov 1996. Phrack Magazine.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, 1994. (DIKU report 94/19).

[7] E. D. Anderson and K. D. Anderson. Presolving in linear programming. *Mathematical Programming*, 71(2):221–245, 1995.

[8] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array-bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[9] J. W. Chinnek and E. W. Dravinieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[10] T-C. Chiueh and F-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, April 2001.

[11] V. Chvátal. *Linear Programming*. W. H. Freeman and Co., New York, 2000.

[12] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the Real World. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[13] T. H. Cormen, C. E. Lieserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2001.

[14] C. Cowan, S. Beaattie, R-F Day., C. Pu, P. Wagle, and E. Walthinsen. Automatic detection and prevention of buffer overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[15] 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 (TOPLAS)*, 13(4):452–490, October 1991.

[16] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J., 1963.

[17] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[18] M. Garey and D. Johnson. *Computers and Intractability*. W. H. Freeman and Co., San Francisco, CA, 1979.

[19] A. J. Hoffman and J. B. Kruskal. Integral boundary points of complex polyhedra. *Linear Inequalities and Related Systems*, pages 233–246, 1956.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[21] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 11–20, New York, December 1994.

[22] K. Landfield. WU-FTPD resource center; personal communication. May 2003.

[23] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[24] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM SIGPLAN-SIGACT Conference on the Principles of Programming Languages (POPL)*, January 2002.

[25] Posting on bugtraq mailing list. Technical analysis of the remote sendmail vulnerability. 4th March 2003. www.securityfocus.com/archive/1/313757.

[26] CPLEX Optimizer. *ILOG CPLEX Division*. 889 Alder Avenue, Incline Village, Nevada. www.cplex.com/.

[27] M. Prasad and T-C. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX'03 Annual Technical Conference*, June 2003.

[28] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, volume 20, pages 41–52, October 1995.

[29] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[30] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, N.Y., 1986.

[31] N. P. Smith. Stack smashing vulnerabilities in the UNIX operating system. 1997.

[32] A. F. Veinott and G. B. Dantzig. Integer extreme points. *SIAM Review*, 10:371–372, 1968.

[33] D. Wagner. *Static Analysis and Computer Security: New techniques for software assurance*. PhD thesis, University of California, Berkeley, December 2000.

[34] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security (NDSS)*, February 2000.

[35] S. J. Wright. *Primal-Dual Interior-Point Methods*. SIAM, Philadelphia, 1997.

[36] R. Wunderling. *Paralleler und Objektorientierter Simplex-Algorithmus*. PhD thesis, Konrad-Zuse-Zentrum fur Informationstechnik Berlin, TR 1996-09. `www.zib.de/PaperWeb/abstracts/TR-96-09/`.

[37] R. Wunderling, A. Bley, T. Pfender, and T. Koch. Sequential object-oriented simplex class library (SoPlex). `www.zib.de/Optimization/Software/Soplex/`.

[38] H. Xi and F. Pfenning. Eliminating array bounds checks through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1998.

[39] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

A Appendix

A.1 Summary Constraints

The approach described in this section addresses the shortcomings of constraint inlining – namely, the method described here handles recursion and does not result in a large number of variables. The basic idea is to summarize the effect of a function call using a set of constraints expressed only in terms of the constraint variables denoting the global program variables and the formal parameters of the called function. We refer to such constraints as *summary* constraints. Consider for instance, the function `copy_buffer` shown in Figure 2. Figure 13 shows a subset of constraints (only those involving `max` variables) generated by `copy_buffer`, and the corresponding summary constraints. Notice that the summary constraints are produced in terms of the constraint variables denoting the formal parameters of `copy_buffer`. The program statements responsible for generating each constraint are also shown.

To summarize the effect of a function call, we must eliminate the constraint variables corresponding to the local variables of the called function. This will result in a new set of constraints for the called function in terms of the constraint variables denoting the formal parameters and the global program variables alone. There are several variable elimination techniques available for linear constraint systems, the most common one being the *Fourier-Motzkin elimination* method. The Fourier-Motzkin method takes as input a set of constraints C and a set of variables V which must be retained in the summary constraints. It then iteratively eliminates the variables not in V. For example, for the constraints shown in Figure 13 the Fourier-Motzkin method would eliminate `copy!alloc!max` by combining constraints `(1)` and `(3)` to produce constraint `(A)`. Similarly, it would eliminate `copy!used!max` by combining constraints `(2)` and `(4)` to produce constraint `(B)`.

The Fourier-Motzkin variable elimination method works with affine constraints, and in the worst case, can result in the generation of a large number of constraints. Specifically, if we want to eliminate a variable `v` from a set of constraints, where m constraints use `v` and n constraints define `v`, the output could have as many as $m \cdot n$ constraints. This problem can be partially alleviated by eliminating constraints which are implied by other constraints.

Our observation however is that most of the constraints that are generated by our tool are difference constraints. For instance, about 98.8% of the constraints generated by our tool for `sendmail-8.7.6` were difference constraints. Variable elimination for difference constraints reduces to an all-pairs shortest or longest path problem on a graph formed by the constraints. Hence, we will restrict our exposition to the case when a function generates only difference constraints. A difference constraint has at most two variables, and involves exclusively `max` variables or `min` variables. Hence, when we consider a function that only generates difference constraints, the constraint subsystem involving the `max` variables is completely disjoint from the constraint subsystem involving the `min` variables. This means that we can produce summary constraints for each of these subsystems independently.

First consider a function that does not call other functions, or only calls those functions for which summary functions are available. The function `copy_buffer` from Figure 2 is an example of such a function, since it only makes calls to `strcpy` and `malloc` and we have the summary functions for both of these.

To produce summary constraints for a set of constraints C of such a function in terms of a set of variables V (the set of formal-parameters and globals of the function), we construct a graph to denote the constraints in C. The vertices of this graph are the constraint variables that appear in C. For a constraint of the form $\mathtt{v}_1 \geq \mathtt{v}_2 + \mathtt{w}$, where $\mathtt{v}_1$ and $\mathtt{v}_2$ are `max` variables, we draw an edge with weight `w` from $\mathtt{v}_2$ to $\mathtt{v}_1$. Since there may be several constraints that relate $\mathtt{v}_1$ and $\mathtt{v}_2$, the edge is assigned a weight equal to the greatest difference between these variables. For each constraint of the form $\mathtt{v}_1 \geq \mathtt{w}$, we draw an edge with weight `w` from a dummy "zero" variable $\mathtt{v}_0$ to $\mathtt{v}_1$. For instance, the graph of the constraints involving the `max` variables for the function `copy_buffer` is shown in Figure 14 (the variable `buffer!alloc!max` was not involved in any constraints generated by `copy_buffer` and is hence not shown in the graph). The problem of generating

Subset of Constraints Generated by `copy_buffer`

```
(1) copy!alloc!max ≥ buffer!used!max - 1 (by line 17)
(2) copy!used!max ≥ buffer!used!max (by line 18)
(3) copy_buffer$return!alloc!max ≥ copy!alloc!max (by line 19)
(4) copy_buffer$return!used!max ≥ copy!used!max (by line 19)
```

Equivalent Summary Constraints

```
(A) copy_buffer$return!alloc!max ≥ buffer!used!max - 1
(B) copy_buffer$return!used!max ≥ buffer!used!max
```

Figure 13: Summary constraints for `copy_buffer`

summary functions now reduces to finding the *longest* path between each pair of vertices in V. Intuitively, the longest path length is the maximum difference between the two variables. Hence, if the longest path length from $\mathtt{v}_1$ to $\mathtt{v}_2$ is `a`, we would generate the constraint $\mathtt{v}_2 \geq \mathtt{v}_1 + \mathtt{a}$. The all pairs shortest path problem for vertices in V can be solved using well known techniques (such as the Floyd-Warshall algorithm [13]). An analogous construction for the `min` variables helps produce the summary constraints for the constraints consisting of the `min` variables. In this case, a constraint $\mathtt{v}_1 \geq \mathtt{v}_2 + \mathtt{w}$ would result in an edge with weight `w` from $\mathtt{v}_2$ to $\mathtt{v}_1$, where $\mathtt{v}_1$ and $\mathtt{v}_2$ are `min` variables. However, in this case we would be required to find the *shortest* path between each pair of vertices in V. Thus, for `copy_buffer`, the graph shown in Figure 14 yields the constraints shown in Figure 13.

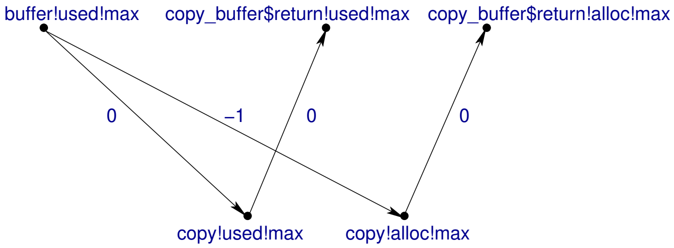

Figure 14: Graph for Summary Constraint Production

We can now use the summary constraints computed for `copy_buffer` in `main` to make the calls to `copy_buffer` context sensitive. This is denoted pictorially in Figure 15. This figure shows the portion of the constraint graph of `main` from Figure 2 pertaining to the constraints generated at line `(10)`. The dotted edge originating from `buf!used!max` denotes the assignment of `buf!used!max` to `buffer!used!max`, while the dotted edges incident on `cc2!used!max` and `cc2!alloc!max` denote the assignment statements from the formal-out constraint variables of `copy_buffer` to the actual-out constraint variables. The dotted edges from `buffer!used!max` to `copy_buffer$return!alloc!max` and `copy_buffer$return!used!max` denote the summary constraints `(A)` and `(B)` from Figure 13 respectively, and are computed by obtaining the pairwise longest paths from Figure 14. The bold edges denote the subsitution of the actual variables in place of the formal parameters in the summary constraints. When we generate constraints for `main`, we only generate the constraints pertaining to the bold lines shown in the figure. Hence, the call to `copy_buffer` at line `(10)` in the program in Figure 2 would result in the constriants:

```
cc2!used!max ≥ buf!used!max
cc2!alloc!max ≥ buf!used!max - 1
cc2!used!min ≤ buf!used!min
cc2!alloc!min ≤ buf!used!min - 1
```

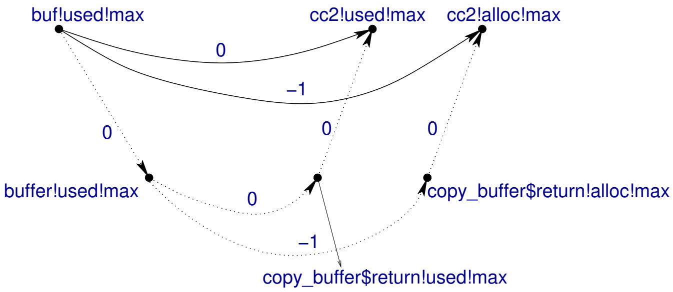

Figure 15: Obtaining Context-Sensitivity

The above technique can be formalized as follows:

- Inspect the call-graph of the program, identify SCCs in it, and coalesce all the nodes belonging to an SCC.
- The resultant graph is a DAG; compute summary constraints in reverse topologically sorted order of the DAG. For each function that calls other functions, summarize the effect of the call by subsituting the actual variables in place of the formal parameters of the called function.

A.2 More Results

We report on the experience of our tool with a few more commercial applications.

A.2.1 Talk Daemon

The talk daemon program, a popular UNIX communication facility, derived from the current FreeBSD release is about 900 lines of code, and produced just 4 warnings on our tool. Upon furthur investigation, we found that all the 4 warnings were false alarms; however one of these warnings was particularly interesting.

The tool reported that as many as 33 bytes could be copied into a buffer pointed to by `tty` which was allocated 16 bytes. The source code responsible for this warning is shown in Figure 16.

On our platform, `UT_LINESIZE` macro-expanded to 32, as a result of which the tool reported the overrun. However, we discovered that when we used the FreeBSD header files for macro-expansion, `UT_LINESIZE` was 8, and hence the warning was suppressed.

This example serves to demonstrate the use of our tool to determine whether an application is vulnerable on a particular platform. For instance, the talk daemon program would have been vulnerable to the aforementioned buffer overflow on our platform.

```
struct utmp
  char  ut_line[UT_LINESIZE];
  ...
};

int find_user(const char *name, char *tty)
  struct utmp ubuf;
  char line[sizeof(ubuf.ut_line) + 1];

  while (fread((char *) &ubuf, sizeof ubuf ..))
    strncpy(line, ubuf.ut_line,
                  sizeof(ubuf.ut_line));
    line[sizeof(ubuf.ut_line)] = '\0';
    ...
        if (...)
          ...
          (void) strcpy(tty, line);
          ...
```

Figure 16: Code Snippet from Talk Daemon

A.2.2 Telnet Daemon from `kth-kerberos-4.0.0`

We tested the Telnet Daemon program from the KTH release of `kerberos-4.0.0` (circa 1995). Telnet daemon has about 9400 lines of code, and produced 40 warnings when analyzed by our tool. The tool identified an interesting bug: it reported that as many as 256 bytes could be copied into `terminaltype`, which points to a buffer only 41 bytes long. We found that the bug was due to a `strncpy` statement in `getterminaltype` in the file `telnetd.c`:

`strncpy(terminaltype, first, sizeof(first))`

Note that `strncpy` was meant to be a "safe" function, but was used in an unsafe way – the programmer mistakenly set the number of bytes to be copied into the destination buffer equal to the size of the source buffer, thus rendering the `strncpy` statement equivalent to its "unsafe" counterpart `strcpy`.