Buffer Overrun Detection using Linear Programming and Static Analysis [*]

Vinod Ganapathy, Somesh Jha
University of Wisconsin-Madison
[vg,jha]@cs.wisc.edu

David Chandler, David Melski, David Vitek
Grammatech Inc., Ithaca, NY 14850
[chandler,melski,dvitek]@grammatech.com

ABSTRACT

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

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Constraints; G.1.6 [**Optimization**]: Linear Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Languages, Reliability, Security

Keywords

Buffer overruns, static analysis, linear programming

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 with consequences 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 "cookbooks" [4] are available to construct such exploits. As observed by several researchers [22, 30], C is highly vulnerable because there are several library functions that manipulate buffers in an unsafe way.

Several approaches have been proposed to mitigate the problem: these range from dynamic techniques [8, 10, 11, 13, 18, 24] that *prevent* attacks based on buffer overruns, to static techniques [17, 22, 26, 29, 30] that examine source code to *eliminate* these bugs before the code is deployed. Unlike static techniques, dynamic techniques do not eliminate bugs, and typically have the undesirable effect of causing the application to crash when an attack is discovered.

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: We discuss related research in Section 2, followed by an overall description of our tool in Section 3. Section 4 describes constraint resolution techniques used by our tool, and Section 5 describes techniques to make the program analysis context-sensitive. Section 6 contains experimental results, and Section 7 concludes.

2. RELATED WORK

Several techniques have been proposed to mitigate the problem of buffer overruns. Dynamic techniques such as Stackguard [13], RAD [10] help to detect and prevent stack smashing attacks by protecting the return address on the stack. ProPolice [18] generalizes these techniques by protecting more entities such as frame pointers, local variables and function arguments. Pointguard [14] protects all pointer accesses by encrypting the pointers when they are stored in memory, and decrypting them when they are loaded into registers. Safe languages like Java introduce runtime array bounds checks to preserve type-safety. However, redundant runtime checks can impose performance overhead, and tools such as ABCD [8] aim to eliminate redundant checks. CCured [11, 24] is a tool that uses static analysis to judiciously insert runtime checks for correctness of pointer manipulations to create a type-safe version of a C program. These techniques prevent attacks based on unsafe memory accesses, but fail to eliminate the bugs from source code.

This paper focuses on static analysis techniques that examine source code for the presence of buffer overruns, and thus help the developer in eliminating the overrun before source code is deployed.

[*] This work was supported in part by NSF grant CCR-9619219 and ONR contracts N00014-01-1-0796 and N00014-01-1-0708.

CCS'03, October 27–31, 2003, Washington, DC, USA
Copyright 2003 ACM 1-58113-738-9/03/0010 ...$5.00.

Several static analysis tools have been proposed. These tools can be broadly classified as (a) Annotation driven tools (b) Tools that use symbolic analysis and (c) Tools that extract a model from the source code, and use it to detect the presence of bugs.

CSSV [17] and Splint [22] are annotation driven tools. In these tools, user-supplied annotations, such as pre- and post-conditions of a function, are used to aid static analysis. CSSV 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, with correctness assertions included, and use a conservative static analysis algorithm to detect faulty integer manipulations, which directly translate to bugs in the C source code. The analysis is performed on a per-procedure basis, and annotations (called *contracts*) are used to make the analysis inter-procedural. The number of false alarms generated by the tool depends on the accuracy of the contracts. The analysis used by CSSV to check the correctness of integer manipulations was heavyweight, and may scale poorly to large programs. For instance, CSSV took $>$ 200 seconds to analyze a string manipulation program with a total of about 400 lines of code. Splint on the other hand, sacrifices soundness and completeness, and uses a light-weight static analysis to detect bugs in source code. Splint uses a flow-sensitive intra-procedural program analysis, and user supplied pre- and post-conditions are used to make the analysis inter-procedural.

ARCHER [33] is a tool that functions by symbolically executing the code, while maintaining information about variables in a database as execution proceeds. The current state of the program is given by the values in the database. The execution of program statements potentially causes a change in the state of the program. At statements that access buffers, ARCHER checks, using the database, whether the access is within the bounds of the array, and flags an error if not. Rugina and Rinard [26] describe a tool geared specifically to detect out-of-bounds errors and race conditions on small divide and conquer programs where they determine symbolic bounds on array indices and use this information to detect illegal accesses. Larson and Austin propose a testing tool [23] to detect input related faults. This tool uses actual program execution using a test input, but enhances bug coverage by maintaining more information about the possible values of variables along the path followed by the test input. These techniques have the advantage that they can be used to detect more than just array out of bounds accesses, as is demonstrated in [23]. Moreover, the analysis is path sensitive since an actual program path is followed, and hence false alarm rates are low. However, the disadvantage is that the coverage of these tools is limited to the set of program paths examined.

BOON [29, 30], like our tool, extracts a model from the source code – namely, these tools model strings as abstract data types and transform the buffer overrun detection problem into a range analysis problem. However, BOON does not employ precise pointer analysis algorithms. Moreover, the analysis was flow- and context-insensitive. Our tool builds on the seminal ideas introduced in BOON by using more precise pointer analysis algorithms, and enhances the program analysis to make it context-sensitive. Additionally, our tool employs algorithms based on linear programming for constraint resolution as opposed to the custom built range solver employed by BOON. Our tool also equips the user with several other static analysis algorithms such as static slicing, which enable the user to understand the reason behind the bug.

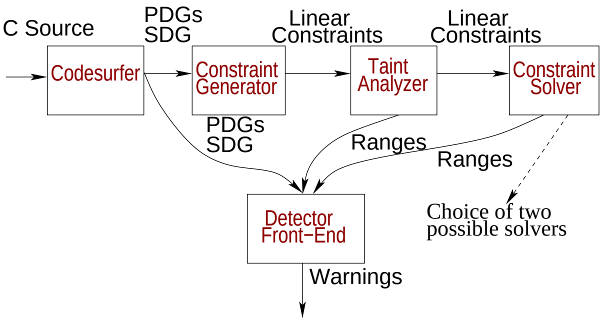

Figure 1: Overall Architecture of the Buffer Overrun Tool

```
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

3. OVERALL TOOL ARCHITECTURE

The tool has five components (Figure 1) that are described in the remainder of this section. Section 3.1 describes the code understanding tool CodeSurfer. CodeSurfer is used by the *constraint generator*, and the *detector front-end* which is a GUI to help the user examine potential overruns. Section 3.2 describes constraint generation. Section 3.3 presents *taint analysis*, which identifies and removes unconstrained constraint variables. Section 3.4 overviews constraint resolution, and Section 3.5 explains the use of the solution to the constraint system to detect potential buffer overruns. The program in Figure 2 will serve as a running example.

3.1 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 inter-procedural chopping between two program points, 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 our tool: (1) the constraint generator is a CodeSurfer plug-in that makes use of CodeSurfer's ASTs and pointer analysis (based on Andersen's analysis [6]). (2)

Constraint	Stmt.
header!used!max ≥ 2048	6
header!used!min ≤ 1	6
buffer!used!max ≥ buf!used!max	10
buffer!used!min ≤ buf!used!min	10
buffer!alloc!max ≥ buf!alloc!max	10
buffer!alloc!min ≤ buf!alloc!min	10
copy_buffer$return!alloc!max ≥ copy!alloc!max	18
copy_buffer$return!alloc!min ≤ copy!alloc!min	18
copy_buffer$return!used!max ≥ copy!used!max	18
copy_buffer$return!used!min ≥ copy!used!min	18
cc2!used!max ≥ copy_buffer$return!used!max	10
cc2!used!min ≤ copy_buffer$return!used!min	10
cc2!alloc!max ≥ copy_buffer$return!alloc!max	10
cc2!alloc!min ≥ copy_buffer$return!alloc!min	10
counter′!max ≥ counter!max + 1	8
counter!max ≥ counter′!max	8
counter′!min ≤ counter!min + 1	8
counter!min ≤ counter′!min	8

Figure 3: Some constraints for the running example

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.

3.2 Constraint Generation

Constraint generation in our tool is similar to the approach proposed in BOON [30]. 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, namely, `buf!alloc!max` and `buf!alloc!min`, which denote the maximum and minimum number of bytes allocated for the buffer, and `buf!used!max` and `buf!used!min`, which denote the maximum and minimum number of bytes used by the buffer. Each integer variable `i` is modeled by 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 ≥ buffer!used!max

copy!used!min ≤ 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 BOON, 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 BOON 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 aim 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, and 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 during constraint generation. Hence, in Figure 2, the predicate `counter` < 10 in line `(8)` was ignored.

• The statement `counter++` is particularly interesting when generating linear constraints. A linear constraint such as `counter!max` ≥ `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 4).

• 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` $\geq \infty$, `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` $\leq -\infty$

3.3 Taint Analysis

The linear constraints then pass through a *taint analysis* module. The main goal of the taint analysis module is to make the constraints amenable to the solvers presented in Section 4. These solvers use linear programming, which can work only with finite values, hence this requires us to remove variables that can obtain infinite values. Section 4 will also demonstrate the importance of `max` variables having finite lower bounds and `min` variables having finite upper bounds. Hence, taint analysis aims to:

• *Identify and remove any variables that get an infinite value*: As mentioned in section 3.2, 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 $\pm\infty$ 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.

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. The taint analysis algorithm is presented in detail in [19].

3.4 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 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 4. We explain in detail in Section 4 how these values were obtained.

3.5 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 4.

- 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.

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

Figure 4: Values of some constraint variables

We have developed a GUI front end 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.

4. CONSTRAINT RESOLUTION USING LINEAR PROGRAMMING

This section describes two solvers based on linear programming that the tool uses to solve the set of generated constraints. We chose to use linear programming for several reasons:

- The use of linear programming allows us to model arbitrary linear constraints. Hence, our solver automatically evolves to handle new kinds of constraints. Other tools [29, 30, 33] use specialized solvers – generation of new kinds of constraints will mean that these solvers have to be specially adapted to deal with them.
- Commercial implementations of linear program solvers are known to scale efficiently to millions of constraints.
- The use of a well developed theory helped us easily reason about the correctness of our solvers.
- Finally, we are currently working on the use of the *dual* of the linear program for diagnostic information. In particular, we are investigating how the dual linear program can be used to produce a program path that leads to the statement that causes the overflow. Such information is valuable since it tells the user of the tool *why* there was an overrun.

4.1 Overview of the solver

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

$$\begin{array}{rl} \text{Minimize} : & \mathtt{c}^{\mathtt{T}}\mathtt{x} \\ \text{Subject To} : & \mathtt{Ax} \geq \mathtt{b} \end{array}$$

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* $\mathtt{c}^T\mathtt{x}$ 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 [27]. 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 [12, 27] being the most popular of them. Other known techniques, such interior point methods [31] work in polynomial time. Commercially available solvers for solving linear programs, such as SoPlex [32] and CPLEX [25] 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. Formally, 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 for `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 [12]. Our approach is thus an approximation to the real problem of finding *integer* solutions that satisfy the constraints.

4.2 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 *feasible* solution to the linear program. A feasible 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 feasible solution exists, but no solution optimizes the objective function. For instance, consider:

$$\begin{array}{rl} \text{Maximize} : & \texttt{x} \\ \text{Subject To} : & \texttt{x} \geq 5 \end{array}$$

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

$$\begin{array}{rl} \text{Minimize} : & \texttt{counter!max} \\ \text{Subject To} : & \texttt{counter}'\texttt{!max} \geq \texttt{counter!max} + \texttt{1} \\ & \texttt{counter!max} \geq \texttt{counter}'\texttt{!max} \end{array}$$

Figure 5: 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++`.

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 operations research. 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 5 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* [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 6 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

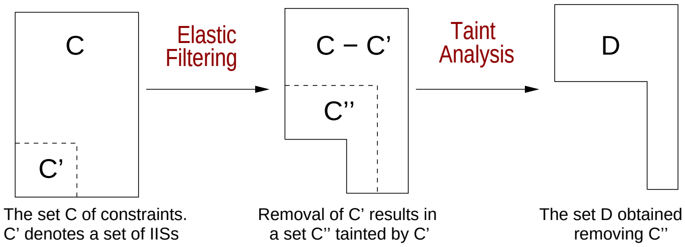

Figure 6: Making an Infeasible set of constraints amenable to Linear Programming

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 apply the taint analysis algorithm to identify the tainted variables, and remove the constraints in which they appear. This step results in further removal of constraints, which are depicted in the Figure 6 by a subset C'' of $C - C'$. The set of constraints after removal of C'', denoted as D in Figure 6, 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. This algorithm is presented in detail in [19].

We have implemented this approach by extending the commercially available package SoPlex [32]. 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. Several such techniques are described in the literature [7]; we have incorporated some of them in our solver.

4.3 Solving Constraints Hierarchically

While the approach presented above is fast, it is an approximation algorithm. In particular, the algorithm may remove more constraints than are actually required to make the constraints feasible. As a result, more constraint variables may be set to the values ∞ or -∞. To address this imprecision, we have designed an implemented a *hierarchical* solver. The idea behind this solver 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 vertices represents a set of constraints. Moreover, each constraint is associated with exactly one vertex of the DAG. The DAG is constructed by defining a notion of "dependency" between a pair of constraints (see [19]). The topological order of the DAG naturally defines a hierarchy of the vertices. The set of constraints corresponding to each vertex is then solved using linear programming. It can be shown that this approach is mathematically precise in that it sets fewest number of constraint variables to ∞ or -∞, and produces precise ranges. We have omitted the details due to space considerations, consult [19] for details.

5. ADDING CONTEXT SENSITIVITY

The constraint generation process described in Section 3 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 two techniques 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. At each call-site to a function, we produce the constraints for the called function with the local variables and formal parameters renamed uniquely for that call-site. This is illustrated in the example below, which shows some of the constraints for `copy_buffer` from Figure 2 specialized for the call-site at line `(7)`:

```
copy!alloc!max_1 ≥ buffer!used!max_1 - 1
copy!used!max_1 ≥ buffer!used!max_1
copy!used!min_1 ≤ buffer!used!min_1
copy_buffer$return!used!max_1 ≥ copy!used!max_1
copy_buffer$return!used!min_1 ≤ copy!used!min_1
```

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.

With this approach to constraint generation, we obtain the range [0..2047] and [1..2048] for `cc1!alloc` and `cc1!used` respectively, while `cc2!alloc` and `cc2!used` obtain [0..1023] and [1..1024] respectively, which is an improvement over the values reported in Figure 4.

Note that using the constraint inlining approach, we can obtain the value of a variable with a particular calling context (the calling context will be encoded implicitly in the renamed variable). However, this comes at a price – since we can have an exponential number of calling contexts, the constraint system will have a large number of variables, and as a result, a large number of constraints. Moreover, this approach cannot work with recursive function calls.

These drawbacks can be overcome through the use of *summary information*. In this approach to inter-procedural dataflow analysis, first suggested by Sharir and Pnueli [28], a "summary" is obtained for each function `foo`, and the summary information is used at each callsite to `foo` to "simulate" the effect of the call.

In our case, a function can be summarized by generating *summary constraints*, which summarize the effect of a function in terms of the constraint variables representing global variables and formal parameters of the function. This is equivalent to finding a projection of the constraints generated by the function on the global variables and the formal parameters of the function. This problem has several well-known solutions. In particular, if the function generates only *difference constraints*, then the problem of finding the summary constraints reduces to an instance of the all-pairs shortest path algorithm [12, 19], for which several efficient algorithms are known. For more general kinds of constraints, the Fourier-Motzkin variable elimination algorithm [16] can be used.

Consider, for instance, constraints generated by `copy_buffer`. This function does not modify or use global variables, and hence we obtain the summary constraints (shown below) by projecting the constraints on the formal parameters of this function.

```
copy_buffer$return!alloc!max ≥ buffer!used!max - 1
copy_buffer$return!used!max ≥ buffer!used!max
copy_buffer$return!alloc!min ≤ buffer!used!min - 1
copy_buffer$return!used!min ≤ buffer!used!min
```

To obtain context sensitivity, we use these constraints at each callsite to `copy_buffer` with the formal parameters appearing in the summary constraints replaced with the corresponding actuals. Constraints are generated at line `(7)` by replacing the constraint variables corresponding to `buffer` and `copy_buffer$return` in the summary constraints with the constraint variables corresponding to `header` and `cc1` respectively. Similarly, the relationship between `cc2` and `buf` at line `(10)` can be obtained by substituting them in place of `copy_buffer$return` and `buffer` respectively, in the summary constraints. Note that we must still retain the assignment of the actual variable to the formal-in parameter so that we can obtain the values of the constraint variables corresponding to the local variables of the called function.

This approach is more efficient than the constraint inlining approach since it does not cause an increase in the number of constraint variables. However it is also less precise than constraint inlining because of the same reason. Observe that in constraint inlining the variables were renamed at each callsite, thus enabling us to examine their values due to a particular calling context. On the other hand, in the summary constraints approach the values of the variables are merged across different calling contexts, thus leading to loss of precision. For instance, consider the program in Figure 2. While the values for `cc1!used`, `cc1!alloc`, `cc2!used` and `cc2!alloc` are the same as obtained using constraint inlining, the values of `copy!alloc` and `copy!used` are [0..2047] and [1..2048] respectively. This is because the values that these variables obtained due to calls at line `(7)` and line `(10)` are “merged”. The constraint inlining approach returns the values [0..2047] and [1..2048] for `copy!alloc` and `copy!used` respectively due to the call at line `(7)`, and returns [0..1023] and [1..1024] due to the call at line `(10)`.

This approach is capable of handling recursive function calls, however for simplicity we do not attempt to summarize recursive function calls in our prototype implementation.

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 [19].

Our experiments were performed on a 3GHz Pentium-4 Xeon processor machine with 4GB RAM, running Debian GNU/Linux 3.0. We used CodeSurfer-1.8 for our experiments, the `gcc-3.2.1` compiler for building the programs. 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 [34] to deal with structure casts. We configured the tool to use the hierarchical solver described in Section 4.3 for constraint resolution (so the values obtained will be precise) and produce constraints in a context-insensitive fashion. Section 6.4 discusses the effects of context-sensitive constraint generation.

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 the file `rd-servers.c`, where as many as 8192 bytes could be copied into the buffer for which up to 4095 bytes were allocated. Figure 7 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 7: 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.

A few 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, 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. 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 [22] 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]. The call `strcat(dst, src)` would be modeled as four linear constraints by our tool:

```
dst′!used!max ≥ dst!used!max + src!used!max
dst!used!max ≥ dst′!used!max
dst′!used!min ≤ dst!used!min + src!used!min
dst!used!min ≤ dst′!used!min
```

The first two constraints make the linear program infeasible, as explained in Section 4, 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 latest version of `sendmail`. Both `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 to report an off-by-one bug [30] 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 [30], 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 [5].

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

We note that this bug is hard to track precisely in a flow-insensitive analysis. Moreover, 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. We are working towards enhancing our infrastructure to support these features.

6.3 Performance

Table 1 contains representative numbers from our experiments with `wu-ftpd-2.6.2` and `sendmail-8.7.6`. All timings were obtained using the UNIX `time` command. 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 IIS detection based approach

	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
Total (LP./Hier.)	100.55/106.82 sec	338.24/350.96 sec
Number of Constraints Generated		
Pre-Taint	22008	104162
Post-Taint	14972	24343

Table 1: Performance of the tool

and the hierarchical solves approach respectively (Section 4). 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.

As noted earlier, the IIS detection based approach is more efficient, but is not mathematically precise, whereas the hierarchical solver is mathematically precise. We however found that the solution produced by the IIS detection based approach 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. We also found that this imprecision did not significantly affect the number of warnings – in case of `wu-ftpd-2.6.2` and `sendmail-8.7.6` the IIS based approach resulted in 1 and 2 more warnings respectively (these warnings were false alarms), which shows that in practice we can use the faster IIS detection based approach with little loss of precision.

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. Note that *adding context sensitivity will not find new overruns*. Adding context sensitivity changes the constraints generated so that they precisely reflect the call-return semantics of functions. As a result, we can expect more precise values from the constraint solvers. 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 3.5, 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× increase in the number of constraints, and a 8.7× 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 63704 specialized range variables, 7497 range variables obtained more precise values than the corresponding unspecialized range variables.
- Out of 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`. Constraint inlining 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 3, 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 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 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 [30] 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. Moreover, the benefits observed by adding context-sensitivity were somewhat limited because of the flow insensitivity of the analysis. 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. However, the SSA representation 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 [30]. 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. CONCLUSIONS

We have demonstrated a light-weight 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 context sensitivity. 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.

Acknowledgments. We would like to thank the members of the Wisconsin Safety Analyzer research group, Michael Ferris, Aditya Rao and the anonymous reviewers for their suggestions.

8. 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] Technical analysis of remote sendmail vulnerability. www.securityfocus.com/archive/1/313757.

[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 Prog.*, 71(2):221–245, 1995.

[8] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array-bounds checks on demand. In *ACM Conf. on Prog. Lang. Design and Impl. (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 *21st Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2001.

[11] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the Real World. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2003.

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

[13] C. Cowan, S. Beattie, R-F Day., C. Pu, P. Wagle, and E. Walthinsen. Automatic detection and prevention of buffer overflow attacks. In *7th USENIX Sec. Symp.*, 1998.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Sec. Symp.*, 2003.

[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 Trans. on Prog. Lang. and Systems (TOPLAS)*, 13(4):452–490, 1991.

[16] G. B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[17] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2003.

[18] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. 2000. www.trl.ibm.com/projects/security/ssp/main.html.

[19] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. 2003. UW-Madison Comp. Sci. Tech. Report 1488. ftp://ftp.cs.wisc.edu/pub/tech-reports/reports/2003/tr1488.ps.Z

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

[21] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *2nd ACM Symp. on Foundations of Soft. Engg. (FSE)*, pages 11–20, New York, 1994.

[22] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Sec. Symp.*, 2001.

[23] E. Larson and T. Austin. High coverage detection of input related security faults. In *12th USENIX Sec. Symp.*, 2003.

[24] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Conf. on the Principles of Prog. Lang. (POPL)*, 2002.

[25] CPLEX Optimizer. www.cplex.com/.

[26] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2000.

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

[28] M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Dataflow Analysis*. Prentice Hall Inc., 1981.

[29] D. Wagner. *Static Analysis and Computer Security: New techniques for software assurance*. PhD thesis, UC Berkeley, Dec 2000.

[30] 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)*, 2000.

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

[32] 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/.

[33] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *9th European Soft. Engg. Conf. and 11th ACM Symp. on Foundation of Soft. Engg. (ESEC/FSE)*, 2003.

[34] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1999.