A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*

David Wagner Jeffrey S. Foster Eric A. Brewer Alexander Aiken
University of California, Berkeley

Abstract

We describe a new technique for finding potential buffer overrun vulnerabilities in security-critical C code. The key to success is to use static analysis: we formulate detection of buffer overruns as an integer range analysis problem. One major advantage of static analysis is that security bugs can be eliminated before code is deployed. We have implemented our design and used our prototype to find new remotely-exploitable vulnerabilities in a large, widely deployed software package. An earlier hand audit missed these bugs.

1. Introduction

Buffer overrun vulnerabilities have plagued security architects for at least a decade. In November 1988, the infamous Internet worm infected thousands or tens of thousands of network-connected hosts and fragmented much of the known net [17]. One of the primary replication mechanisms was exploitation of a buffer overrun vulnerability in the `fingerd` daemon.

Since then, buffer overruns have been a serious, continuing menace to system security. If anything, the incidence of buffer overrun attacks has been increasing. See Figure 1 for data extracted from CERT advisories over the last decade. Figure 1 shows that buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time. A partial examination of other sources suggests that this estimate is probably not too far off: buffer overruns account for 27% (55 of 207) of the entries in one vulnerability database [29] and for 23% (43 of 189) in another database [33]. Finally, a detailed examination of three months of the `bugtraq` archives (January to March, 1998) shows that 29% (34 of 117) of the vulnerabilities reported are due to buffer overrun bugs [7].

Buffer overruns are so common because C is inherently unsafe. Array and pointer references are not automatically bounds-checked, so it is up to the programmer to do the checks herself. More importantly, many of the string operations supported by the standard C library—`strcpy()`, `strcat()`, `sprintf()`, `gets()`, and so on—are unsafe. The programmer is responsible for checking that these operations cannot overflow buffers, and programmers often get those checks wrong or omit them altogether.

As a result, we are left with many legacy applications that use the unsafe string primitives unsafely. Programs written today still use unsafe operations such as `strcpy()` because they are familiar. Even sophisticated programmers sometimes combine the unsafe primitives with some ad-hoc checks, or use unsafe primitives when they somehow "know" that the operation is safe or that the source string cannot come under adversarial control.

Unfortunately, programs that use just the "safe" subset of the C string API are not necessarily safe, because the "safe" string primitives have their own pitfalls [43]:

- The `strn*()` calls behave dissimilarly. For instance, `strncpy(dst, src, sizeof dst)` is correct but `strncat(dst, src, sizeof dst)` is wrong. Inconsistency makes it harder for the programmer to remember how to use the "safe" primitives safely.
- `strncpy()` may leave the target buffer unterminated. In comparison, `strncat()` and `snprintf()` always append a terminating `'\0'` byte, which is another example of dissimilarity.
- Using `strncpy()` has performance implications, because it zero-fills *all* the available space in the target buffer after the `'\0'` terminator. For example, a `strncpy()` of a 13-byte buffer into a 2048-byte buffer overwrites the entire 2048-byte buffer.
- `strncpy()` and `strncat()` encourage off-by-one bugs. For example, `strncat(dst, src, sizeof dst - strlen(dst) - 1)` is correct, but don't forget the `-1`!
- `snprintf()` is perhaps the best of the "safe" primitives: it has intuitive rules, and it is very general. However, until recently it was not available on many systems, so portable programs could not rely on it.

*This research was supported in part by the National Science Foundation Young Investigator Award No. CCR-9457812, NASA Contract No. NAG2-1210, and an NDSEG fellowship.

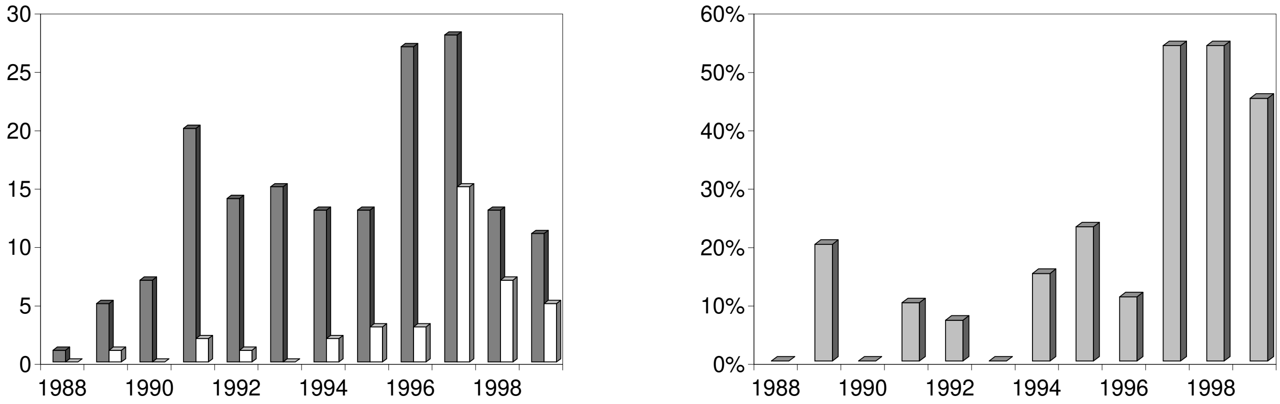

Figure 1. Frequency of buffer overrun vulnerabilities, derived from a classification of CERT advisories. The left-hand chart shows, for each year, the total number of CERT-reported vulnerabilities and the number that can be blamed primarily on buffer overruns. The right-hand chart graphs the percentage of CERT-reported vulnerabilities that were due to buffer overruns for each year.

In all cases, the programmer must still keep track of buffer lengths accurately, which introduces another opportunity for mistakes.

In short, today's C environments make it easy to do the wrong thing, and, worse still, hard to do the right thing with buffers. This suggests that an automated tool to help detect this class of security-relevant coding errors may be of considerable benefit.

1.1. Overview

This paper describes a tool we developed to detect buffer overruns in C source code. Though this is only a first prototype, early results look promising. For example, the tool found several serious new vulnerabilities in one large security-critical application, even though it had been hand-audited previously.

This work involves a synthesis of ideas from several fields, including program analysis, theory, and systems security. The main idea is to apply standard static analysis techniques from the programming languages literature to the task of detecting potential security holes. We focus specifically on static analysis so that vulnerabilities can be proactively identified and fixed before they are exploited. We formulate the buffer overrun detection problem as an *integer constraint* problem, and we use some simple graph-theoretic techniques to construct an efficient algorithm for solving the integer constraints. Finally, security knowledge is used to formulate heuristics that capture the class of security-relevant bugs that tend to occur in real programs.

Others have applied runtime code-testing techniques to the problem, using, e.g., black-box testing [41, 42] or software fault injection [21] to find buffer overruns in real-world applications. However, runtime testing seems likely to miss many vulnerabilities. Consider the following example:

```
if (strlen(src) > sizeof dst)
    break;
strcpy(dst, src);
```

Note that off-by-one errors in buffer management, such as the one shown above, have been exploited in the past [36, 48]. The fundamental problem with dynamic testing is that the code paths of greatest interest to a security auditor—the ones which are never followed in ordinary operation—are also the ones that are the hardest to analyze with dynamic techniques. Therefore, in this work we focus on static analysis.

A theme in this work is the trade-off between precision and scalability. If scalability is not addressed from the start, program analyses often have trouble handling large applications. Since we wish to analyze large programs, such as sendmail (tens of thousands of lines of code), we explicitly aim for scalability even if it comes at some cost in precision. This motivates our use of several heuristics that trade off precision for scalability.

As a result of imprecision, our analysis may miss some vulnerabilities (*false negatives*) and produce many false alarms (*false positives*), but it is still a useful tool. In our experience, even though our relatively imprecise analysis generates many false alarms, it still reduces the number of unsafe string operations to be checked by hand by an order of magnitude or more; see Section 5.5.

We introduce two fundamental, new insights in this paper:

1. **We treat C strings as an abstract data type.** In C, pointers are the bane of program analysis, and any code fragments that manipulate buffers using pointer operations are very difficult to analyze. However,

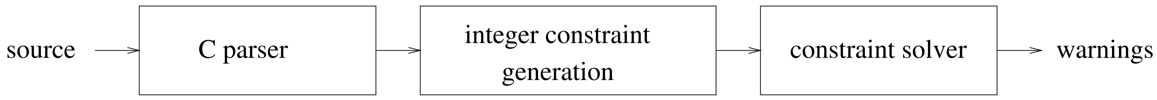

Figure 2. The architecture of the buffer overflow detection prototype.

most C buffer overruns are in string buffers, and most string operations use the standard C library functions. This suggests modelling C strings as an abstract data type with operations like `strcpy()` (copy strings), `strcat()` (concatenate strings), and so on. Any buffer overruns caused by manipulating strings using primitive pointer operations cannot be detected, but such code won't otherwise interfere with the analysis.

2. **We model buffers as pairs of integer ranges.** Rather than tracking the contents of each string variable directly, we summarize each string with two integer quantities: the number of bytes allocated for the string buffer (its *allocated size*), and the number of bytes currently in use (its *length*). The standard C library functions can be modelled by what they do to the allocated size and length of strings without regard to the strings' contents.

   We formulate the problem of detecting buffer overflows in terms of integer range tracking. Any algorithm for integer range analysis will work: we just check, for each string buffer, whether its inferred allocated size is at least as large as its inferred maximum length.

These two ideas provide a conceptual framework for buffer overrun analysis.

Our formulation of the problem suggests a natural division of labor for the implementation: a front end models string operations as integer range constraints, while a back end solves the resulting constraint system. See Figure 2 for a diagram of the system organization.

A secondary contribution of this research is a scalable and very fast integer range analysis. One novel feature of this analysis is the ability to handle cyclic data dependencies without loss of precision by invoking a fixpoint theorem. However, we could easily replace this algorithm with some other technique for integer range analysis.

The organization of the first half of this paper parallels the structure of our implementation. First, we need to define a constraint language (see Section 2). Given this mathematical foundation, we generate constraints from the source code (see Section 3), solve the resulting constraint system (see Section 4), and check all of the string variables for overflow.

The second half of this paper focuses on analysis of our approach, including our initial experience with the prototype (Section 5), a review of related work (Section 6), and a few concluding remarks (Section 7). Appendix A presents the proofs for all of our theoretical results, and Appendix B expands on more details of the constraint solver.

2. The constraint language

In this section we describe the language of constraints we use to model string operations.

Let $\mathbb{Z}$ denote the set of integers and write $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, +\infty\}$ for the extended integers. The subsets of $\mathbb{Z}^\infty$ form a complete lattice with $\subseteq$ as the partial order.

We restrict our attention to integer *ranges*. However, many of the comments here also apply more generally to arbitrary subsets of $\mathbb{Z}^\infty$. A *range* is a set $R \subseteq \mathbb{Z}^\infty$ of the form $[m, n] = \{i \in \mathbb{Z}^\infty : m \leq i \leq n\}$. When S is a subset of $\mathbb{Z}^\infty$, we write $\inf S$ and $\sup S$ for the minimum and maximum element (with respect to $\leq$) of S; in particular, for ranges we have $\inf[m, n] = m$ and $\sup[m, n] = n$. The *range closure* of any set $S \subseteq \mathbb{Z}^\infty$ is the minimal range R (with respect to $\subseteq$) containing S, i.e., $R = [\inf S, \sup S]$. For example, the set $S = \{-1, 0, 4\}$ has range closure $[-1, 4]$, since $\inf S = -1$ and $\sup S = 4$; note that the range notation $[-1, 4]$ is shorthand for the set $\{-1, 0, 1, 2, 3, 4\} \subseteq \mathbb{Z}^\infty$.

We extend the usual arithmetic operators to act on sets $S, T \subseteq \mathbb{Z}^\infty$ in the natural way:

$$\begin{aligned} S + T &= \{s + t : s \in S, t \in T\} \\ S - T &= \{s - t : s \in S, t \in T\} \\ S \times T &= \{s \times t : s \in S, t \in T\} \end{aligned}$$

For notational convenience we often write n as shorthand for the singleton set $\{n\}$, when $n \in \mathbb{Z}^\infty$. Thus, the expression $2T$ acquires its natural interpretation $2T = \{2\} \times T = \{2t : t \in T\}$.

When the result of an operation is not a range, we take its range closure. When this rule is followed, the extended arithmetical operators obey most of the usual algebraic laws. For instance, $S + T = T + S$, $S + 0 = S$, $S + S = 2S$, $S \times T = T \times S$, $0 \times S = 0$, and so on. However, the distributive rule does not hold (in general we only have $S \times (T + U) \subseteq S \times T + S \times U$; see [32]) and the rule for subtraction introduces a slightly ugly feature since in general $S - S \neq 0$.

In practice, it is useful to extend the constraint language to include min and max operators:

$$\begin{aligned} \min(S, T) &= \{\min(s, t) : s \in S, t \in T\} \\ \max(S, T) &= \{\max(s, t) : s \in S, t \in T\} \end{aligned}$$

C code	Interpretation
`char s[n];`	$n \subseteq \text{alloc}(\texttt{s})$
`strlen(s)`	$\text{len}(\texttt{s}) - 1$
`strcpy(dst,src);`	$\text{len}(\texttt{src}) \subseteq \text{len}(\texttt{dst})$
`strncpy(dst,src,n);`	$\min(\text{len}(\texttt{src}), n) \subseteq \text{len}(\texttt{dst})$
`s = "foo";`	$4 \subseteq \text{len}(\texttt{s}), \quad 4 \subseteq \text{alloc}(\texttt{s})$
`p = malloc(n);`	$n \subseteq \text{alloc}(\texttt{p})$
`p = strdup(s);`	$\text{len}(\texttt{s}) \subseteq \text{len}(\texttt{p}), \quad \text{alloc}(\texttt{s}) \subseteq \text{alloc}(\texttt{p})$
`strcat(s,suffix);`	$\text{len}(\texttt{s}) + \text{len}(\texttt{suffix}) - 1 \subseteq \text{len}(\texttt{s})$
`strncat(s,suffix,n);`	$\text{len}(\texttt{s}) + \min(\text{len}(\texttt{suffix}) - 1, n) \subseteq \text{len}(\texttt{s})$
`p = getenv(...);`	$[1, \infty] \subseteq \text{len}(\texttt{p}), \quad [1, \infty] \subseteq \text{alloc}(\texttt{p})$
`gets(s);`	$[1, \infty] \subseteq \text{len}(\texttt{s})$
`fgets(s,n,...);`	$[1, n] \subseteq \text{len}(\texttt{s})$
`sprintf(dst,"%s",src);`	$\text{len}(\texttt{src}) \subseteq \text{len}(\texttt{dst})$
`sprintf(dst,"%d",n);`	$[1, 20] \subseteq \text{len}(\texttt{dst})$
`snprintf(dst,n,"%s",src);`	$\min(\text{len}(\texttt{src}), n) \subseteq \text{len}(\texttt{dst})$
`p[n] = '\0';`	$\min(\text{len}(\texttt{p}), n+1) \subseteq \text{len}(\texttt{p})$
`p = strchr(s,c);`	`p = s+n;` $\quad [0, \text{len}(\texttt{s})] \subseteq n$
`h = gethostbyname(...);`	$[1, \infty] \subseteq \text{len}(\texttt{h->h\_name})$, $[-\infty, \infty] \subseteq \texttt{h->h\_length}$

Table 1. Modelling the effects of string operations: some examples.

For example, when $S = \{1, 2, 3, 4\} = [1, 4]$ and $T = \{3, 4, 5, 6\} = [3, 6]$, then $\inf T = 3$, $\min(S, T) = [1, 4]$, and $S - T = [-5, 1]$.

We define an *integer range expression* e as

$$e ::= v \mid n \mid n \times v \mid e + e \mid e - e \mid \max(e, \ldots, e) \mid \min(e, \ldots, e)$$

where $n \in \mathbb{Z}$ and $v \in$ Vars, a set of range variables. An *integer range constraint* has the form $e \subseteq v$. Notice we require the right-hand side to be a variable.

Note that equality constraints of the form $v + n = w$ fit within this framework, since they can be equivalently expressed as the pair of simultaneous constraints $v + n \subseteq w$, $w - n \subseteq v$. Equality constraints are useful for unifying variables that are discovered (during constraint generation) to refer to the same memory location.

An assignment $\alpha : v \mapsto \alpha(v) \subseteq \mathbb{Z}^\infty$ *satisfies* a system of constraints if all of the constraint assertions are true when the formal variable names v are replaced by the corresponding values $\alpha(v)$. For assignments α and β, we say that $\alpha \subseteq \beta$ if $\alpha(v) \subseteq \beta(v)$ holds for all variables v. The *least solution* to a constraint system is the smallest assignment α that satisfies the system, i.e., a satisfying assignment α such that any other satisfying assignment β obeys $\alpha \subseteq \beta$.

Theorem 1. *Every constraint system has a unique least solution.*

Proof. See the Appendix A for the proof. □

In fact, as we shall see later, these constraint systems usually can be solved efficiently.

3. Constraint generation

The first step is to parse the source code; we use the BANE toolkit [2]. Our analysis proceeds by traversing the parse tree for the input C source code and generating a system of integer range constraints. With each integer program variable `v` we associate a range variable v. As discussed before, with each string variable `s` we associate two variables, its allocated size (the number of bytes allocated for `s`), denoted alloc(`s`), and its length (the number of bytes currently in use), denoted len(`s`). We model each string operation in terms of its effect on these two quantities.

For convenience, the length of a string is defined to include the terminator `'\0'`. Thus, the safety property to be verified is

$$\text{len}(\texttt{s}) \leq \text{alloc}(\texttt{s}) \qquad \text{for all string variables } \texttt{s}.$$

For each statement in the input program, we generate an integer range constraint. Integer expressions and integer variables are modelled by corresponding range operations. For an assignment `v = e`, we generate the constraint $e \subseteq v$. For example, for the assignment `i = i+j`, we generate the constraint $i + j \subseteq i$. We ignore assignments with dereferenced pointers on the left; see below for a discussion.

For string operations, we pattern-match to determine what kind of constraint to generate. Some sample constraints are summarized in Table 1. The left column shows the C code for a string operation of interest, and the right column shows the generated constraints. For example, the second line says that the return value of the `strlen()` library call is the length of the string passed as its first argument, minus one (for the string terminator). The third line in the table says that the effect of the `strcpy()` operation is to overwrite the first argument with the second argument, and thus after the `strcpy()` the length of the first argument is equal to the length of the second argument. Note that although `strncpy()` may leave its target unterminated, we do not model this behavior.

For scalability and simplicity of implementation, we use a *flow-insensitive* analysis, i.e., we ignore all control flow and disregard the order of statements. Flow-insensitive analyses sacrifice some precision for significant improvements in scalability, efficiency, and ease of implementation. We do not claim that flow-insensitive analysis is necessarily the best approach for a production-quality buffer overrun tool; instead, we merely observe that its advantages (ease of implementation, scalability) mapped well to our initial goals (construction of a proof-of-concept prototype that can analyze large, real-world applications).

Note that the `strcat()` operation is difficult to model accurately in a flow-insensitive model, because we must assume that it can execute arbitrarily often (for instance, inside a loop). As a result, in a flow-insensitive analysis *every* non-trivial `strcat()` operation is flagged as a potential buffer overrun. This is a price we have to pay for the better performance of flow-insensitive analyses. Fortunately, most of the C library string operations are *idempotent*, which means that they do not present any intrinsic problems for a flow-insensitive analysis.

Finally, we model function calls *monomorphically*, i.e., we merge information for all call sites of the same function. Let `f()` be a function defined with the formal integer parameter `formal`. We add a variable `f_return` to denote the return value of `f()`. A `return` statement in `f()` is treated as an assignment to `f_return`. Each function call `b = f(a)` is treated as an assignment of the actuals to the formals (i.e., `formal = a`) followed by an assignment that carries the return value of `f()` (i.e., `b = f\_return`). Note that the body of each function is processed only once, so this strategy is simple and efficient, but not necessarily precise.

After the possible ranges of all variables are inferred, we may check the safety property for each string `s`. Suppose the analysis discovers that len(`s`) and alloc(`s`) take on values only in $[a, b]$ and $[c, d]$, respectively. There are three possibilities:

1. If $b \leq c$, we may conclude that the string `s` never overflows its buffer.
2. If $a > d$, then a buffer overrun always occurs in any execution that uses `s`.
3. If the two ranges overlap, then we cannot rule out the possibility of a violation of the safety property, and we conservatively conclude that there is the potential for a buffer overrun vulnerability in `s`.

```
char s[20], *p, t[10];
strcpy(s, "Hello");
p = s + 5;
strcpy(p, " world!");
strcpy(t, s);
```

Figure 3. A buffer overrun that the analysis would not find due to the pointer aliasing. In this example, a 13-byte string is copied into the 10-byte buffer **t**.

3.1. Handling pointers

Ideally, we would like the constraint generation algorithm to be *sound*: if α is a satisfying assignment for the constraint system generated by this algorithm on some program, then $\alpha(v)$ should contain every possible value that the integer program variable `v` can take on during the execution of the program. Our algorithm is, however, unsafe in the presence of pointers or aliasing.

Table 1 is deliberately vague about pointer operations. This is because, in the simplistic model used in the prototype, pointer aliasing effects are largely ignored, and the rules for dealing with pointer expressions are highly ad-hoc. For example, the statement `q = p+j;` is interpreted as alloc(`p`) $- j \subseteq$ alloc(`q`), len(`p`) $- j \subseteq$ len(`q`). This interpretation is correct in the absence of writes to `*p` and `*q`, but due to the implicit aliasing of `p` and `q` a write to one string is not reflected when the other string is read in some subsequent program statement. Figure 3 gives an example of a code segment with a static buffer overrun that is undetected by the analysis. Thus, ignoring pointer aliasing can cause the analysis to miss some vulnerabilities and, as we shall see later, can occasionally cause false alarms.

Doubly-indirected pointers (e.g., `char **`) are hard to handle correctly with our heuristics and thus are ignored in our tool. Arrays of pointers present the same problems and are treated similarly. As an unfortunate result, command-line arguments (`char *argv[]`) are not treated in any systematic way.

Function pointers are currently ignored. We also ignore union types. These simplifications are all unsound in general, but still useful for a large number of real programs.

It seems that one can retain some benefits of static analysis despite (largely) ignoring pointers and aliasing. Nonetheless, in practice there is one related C idiom that cannot be ignored: use of C `struct`'s. Structures form essentially the only mechanism for abstraction or construction of data structures, so it is not surprising that they are widely used. Experience suggests that modelling structures properly is crucial to obtaining good results: an earlier analysis tool that ignored structures was mostly useless for understanding real programs of any non-trivial complexity. One aspect that complicates analysis of structures is that they are commonly used in conjunction with pointers (for example, we might want to know whether `p->left->right` refers to the same object as `q->right->right`), yet one of the goals of the prototype was to avoid the implementation complexity associated with a full treatment of pointers, if possible.

This seeming paradox is resolved with a simple trick for modelling structures: all structure objects with the same (or compatible) C type are assumed to be potentially aliased and are modelled with a single variable in the constraint system (see also [16]). In addition, structure field references are further disambiguated using lexical field names, so that `hp->h_length` is not considered the same memory location as `hp->h_addr`. This technique can introduce false alarms (but doesn't miss real vulnerabilities, unless the program uses casts in unusual ways), yet it seems to work well enough in practice, in lieu of a full pointer analysis.

4. Solving integer range constraints

The design of the constraint language is motivated by the following intuition. Suppose we are analyzing a program with k variables. Consider the statespace $\mathbb{Z}^k$ whose i-th component records the value of the i-th program variable. We may consider an execution of the program as a path through the statespace. With this perspective, our goal is to find a minimal bounding box that encloses all of the dynamically possible paths through the k-dimensional state space.

In this section, we give an efficient algorithm for finding a bounding box solution to a system of constraints. In practice, our algorithm scales linearly on our benchmarks. Notice that the solution to the constraint system gives us bounds on the ranges of each program variable standing alone, but cannot give us any information on relationships that hold between multiple program variables. As an alternative, we could imagine computing a minimal convex polyhedron that encloses all the execution paths (using, e.g., the simplex method). This would return more precise results, but it would probably also scale up very poorly to the large problem instances encountered when analyzing real-world programs. For instance, sendmail contains about 32k non-comment, non-blank lines of C code, and it yields a constraint system with about 9k variables and 29k constraints. The simplification to bounding boxes is what allows the constraint solver to run very efficiently.

We develop a bounding box algorithm by beginning with the simplest case: assume that arithmetic and min/max expressions are omitted, so that each constraint has the form $f(v_i) \subseteq v_j$, where $f \in \mathcal{AF} = \{x \mapsto ax + b : a \in \mathbb{Z}, b \in \mathbb{Z}^\infty\}$ is an affine function on $\mathbb{Z}^\infty$ extended to operate on ranges in the natural way, i.e., $f(R) = \{f(r) : r \in R\} \subseteq \mathbb{Z}^\infty$.

We form a directed graph whose vertices are the variables v_i. For each constraint $f(v_i) \subseteq v_j$ we add the labeled directed edge $v_i \xrightarrow{f} v_j$. Each vertex v_i is marked with a range $\alpha(v_i)$ giving the current estimate of the solution. All ranges are initially set to $\alpha(v_i) := \emptyset$. Then constraints of the form $n \subseteq v$ are processed by setting $\alpha(v_i) := \textsc{Range-closure}(\alpha(v_i) \cup \{n\})$ and the solver is called.

The solver works by propagating information in this graph. We say that an edge $v_i \xrightarrow{f} v_j$ is *active* if $f(\alpha(v_i)) \not\subseteq \alpha(v_j)$. To propagate information along such an active edge (also known as *relaxation*), we set $\alpha(v_j) := \textsc{Range-closure}(\alpha(v_j) \cup f(\alpha(v_i)))$. An *augmenting path* is one containing only active edges. (This wording is in deliberate analogy to standard algorithms for shortest-paths and network flow problems.) The goal of the algorithm is to find augmenting paths and propagate information along them by relaxing the upper bounds on the solution.

If the resulting directed graph is acyclic, we can trivially solve the constraint system in linear time: we topologically sort the graph and propagate information along each edge in sorted order. Graphs with cycles are harder to handle.

The approach given above can be rephrased in the perhaps more familiar language of fixpoints over lattices. Each constraint $f(v_i) \subseteq v_j$ induces a continuous function F on assignments given by

$$(F(\alpha))(v_k) = \begin{cases} \alpha(v_j) \cup f(\alpha(v_i)) & \text{if } j = k \\ \alpha(v_k) & \text{otherwise,} \end{cases}$$

and in this way the constraint system gives us a set of such functions $\{F\}$. Now note that a satisfying assignment for the constraint system forms a fixpoint for all the F's, and vice versa. Therefore, we are seeking the least fixpoint of the functions $\{F\}$, because it will be the least solution to the constraint system.

We could search for the fixpoint using a standard worklist algorithm that visits all the augmenting paths in breadth-first order and propagates information along them by relaxation. However, the basic worklist algorithm would exhibit serious problems: in the presence of cycles, it might not terminate! For instance, consider the constraint system containing the two constraints $5 \subseteq v$ and $v + 1 \subseteq v$. A naive algorithm would loop forever, revising its initial estimate $\alpha(v) = [5, 5]$ to $[5, 6]$, $[5, 7]$, $[5, 8]$, etc. This "counting to infinity" behavior arises because the lattice of ranges has infinite ascending chains, and thus the monotonicity of $\{F\}$ is not enough to ensure termination.

At this point, we have three options for restoring termination.

1. We could restrict attention to those programs that induce acyclic constraint systems.

2. We could introduce a widening operator that raises variables involved in cycles to the trivial solution $[-\infty, \infty]$, as pioneered in [10] and [11]. This avoids infinite ascending chains.

3. We could directly solve the constraint subsystem associated with each cycle, using domain-specific information about the constraint language.

The first is not very attractive, because real programs often involve cycles, such as those created by loops and recursion. Even worse, cycles are almost unavoidable for a flow-insensitive analysis: for example, the C assignment `i = i1+` will always induce a cycle in the form of a constraint $i + 1 \subseteq i$. One disadvantage of the second option is that it introduces imprecision, i.e., it will only provide an approximate solution (an upper bound on the least satisfying assignment).

This paper follows the third option. We show how to avoid divergent behavior, without introducing any imprecision, by directly solving for the fixpoint of the constraint subsystem associated with each cycle.

A typical cycle looks like

$$f_1(v_1) \subseteq v_2, \ \ldots, \ f_{n-1}(v_{n-1}) \subseteq v_n, \ f_n(v_n) \subseteq v_1.$$

Transitively expanding this cycle, we find that $f(v_1) \subseteq v_1$ where $f = f_n \circ \cdots \circ f_1$. (We may view f loosely as Shostak's *loop residue* [56] for the cycle.) The composition of affine functions is affine, so f is affine. The observation is that we can precisely solve this cyclic constraint system without any divergence whatsoever, by using a simple fact on the fixpoints of affine functions.

Lemma 1. *Let $f(x) = ax + b$ be an affine function in $\mathcal{AF}$ with $a > 0$, let R be a range, and let $S \subseteq \mathbb{Z}^\infty$ be the minimal range satisfying $R \subseteq S$ and $f(S) \subseteq S$. Then (1)* $\sup S = \infty$ *if* $\sup f(R) > \sup R$*; also, (2)* $\inf S = -\infty$ *if* $\inf f(R) < \inf R$*. If neither clause (1) nor clause (2) applies, we have $S = R$. If both clauses apply, we have* $S = [-\infty, \infty]$.

Theorem 2. *We can solve the constraint subsystem associated with a cycle in linear time.*

To restate the theorem intuitively: if we ever find an augmenting path that traverses an entire cycle, the theorem shows us how to immediately apply a widening operator *without any loss of precision whatsoever*. This provides a simple way to avoid the "counting to infinity" behavior that arises from traversing a cycle multiple times. Thus, the real contribution of Theorem 2 is that it shows how to find the fixpoint of the system precisely and efficiently; since we are working in a lattice with infinite ascending chains, standard techniques cannot provide this.

Figure 4 presents an algorithm that uses these ideas to handle cycles efficiently. This time, we use a depth-first search so that we can recover the edges participating in the cycle as soon as we see a back-edge. The HANDLE-CYCLE procedure (left unspecified here, for space reasons) retraces the cycle discovered in the depth-first search using the predecessor pointers and then processes that cycle using the algorithm sketched in the proof of Theorem 2 (see Appendix A).

CONSTRAINT-SOLVER
1. Set $\alpha(v_i) := \emptyset$ for all i, and set *done* := false.
2. For each constraint of the form $n \subseteq w$, do
3. Set $\alpha(w) :=$ RANGE-CLOSURE$(\alpha(w) \cup \{n\})$.
4. While *done* $\neq$ true, call ONE-ITERATION.

ONE-ITERATION
1. Set $C(v_i) :=$ white for all i and set *done* := true.
2. For each variable v, do
3. If $C(v) =$ white, do
4. Set prev(v) := null and call VISIT(v).

VISIT(v)
1. Set $C(v) :=$ gray.
2. For each constraint of the form $f(v) \subseteq w$, do
3. If $f(\alpha(v)) \not\subseteq \alpha(w)$, do
4. Set $\alpha(w) :=$ RANGE-CLOSURE$(\alpha(w) \cup f(\alpha(v)))$.
5. Set *done* := false.
6. If $C(w) =$ gray, call HANDLE-CYCLE(v,w,prev).
7. If $C(w) =$ white, do
8. Set prev$(w) := v$ and call VISIT(w).
9. Set $C(v) :=$ black.

RANGE-CLOSURE(S)
1. Return the range $[\inf S, \sup S]$.

Figure 4. An algorithm that efficiently solves systems of integer range constraints.

In theory, this solution process could take $O((n + m)k)$ time in the worst case, where k counts the number of cycles in the graph. In practice, though, k seems to be small, and the algorithm usually runs in linear time, probably because of sparsity and locality in the constraint systems that arise during the analysis of typical programs.

This concludes our treatment of constraint solving for simple constraints. We have extended the algorithm to handle the full constraint language, including multi-variable expressions and min/max operators. See Appendix B for the details.

5. Early experience with the prototype

This section details some early experience with the current version of the overrun detection tool.

The experimental methodology was simple. The tool was applied to several popular software packages. The tool typically produced a number of warnings about potential buffer overruns, and one of us examined the source by hand to screen out the false alarms. Some sample output is shown in Figure 5.

We applied the tool to about a dozen software packages. Due to lack of space, we omit the cases where the tool found nothing of interest.

5.1. Linux net tools

The best success story so far arose from an analysis of the Linux `nettools` package, which contains source for standard networking utilities such as `netstat`, `ifconfig`, `route`, and so on. The programs themselves total about 3.5k lines of code, with another 3.5k devoted to a support library[1].

This package had already been audited by hand once in 1996 after several buffer overruns were found in the code [31], so it came as somewhat of a surprise when the tool discovered several serious and completely new buffer overrun vulnerabilities. One library routine trusts DNS responses and blindly copies the result of a DNS lookup into a fixed-length buffer, trusting both the `hp->h_name` and `hp->h_length` values returned. In both cases, this trust is misplaced. Another routine blindly copies the result of a `getnetbyname()` lookup into a fixed-size buffer. At first glance, this may appear harmless; however, `getnetbyname()` may issue a NIS network query in some cases, and thus its response should not be trusted. Several other places also perform unchecked `strcpy()`'s into fixed-size buffers on the stack that can apparently be overrun by spoofing DNS or NIS results or by simply registering a host with an unexpectedly long name.

These vulnerabilities seem likely to be remotely exploitable[2]. It is worth stressing that these holes were previously unknown, despite an earlier manual audit of the code.

5.2. Sendmail 8.9.3

The latest version of sendmail (about 32k lines of C code) was one of the first programs analyzed. Some sample output is shown in Figure 5, which shows (for example) that solving the constraint system took less than two seconds; also, Section 5.5 presents a more detailed study of the warnings from the tool. Sendmail makes an especially interesting test, because it has been extensively audited by hand for buffer overruns and other vulnerabilities. Also, we feel that it makes for a very thorough test of the applicability of the tool to large, complex applications.

The testing session did not uncover any security vulnerabilities in sendmail-8.9.3. A few small bugs were identified that could in theory lead to buffer overruns, but they do not seem exploitable in practice because the relevant inputs are not under adversarial control. Nonetheless, the new bugs identified do demonstrate the potential to find subtle coding errors in real code using automated analysis techniques.

The most important bug identified by the tool was a complex off-by-one error in the management of string buffers. This bug is hinted at by the warning about `'dfname@collect()'`: the tool discovered that 20 bytes were allocated for a buffer called `dfname` (defined in the `collect()` procedure), and that a string containing possibly as many as 257 bytes might be copied into the 20-byte buffer. This is a potential violation of the safety property. In this case, the tool suggests that the lengthy string came from the return value of `queuename()`, but was not able to identify any further dependencies of interest.

Upon further investigation, using other diagnostics from the tool, we found that a complex sequence of invocations can cause `queuename()` to return a 21-byte string (including the terminating `'\0'`). (The 257-byte figure is a result of imprecision in the analysis.) The troublesome sequence is: `orderq()` reads a file from the queue directory, and copies its filename (possibly as many as 21 bytes long, including the `'\0'`) into `d->d_name` and then into `w->w_name`; then `runqueue()` calls `dowork(w->w_name+2,...)`, and `dowork()` stores its first argument (which can be as long as 19 bytes) into `e->e_id`; next `queuename()` concatenates `"qf"` and `e->e_id`, returning the result, which is copied into `dfname`; but `queuename()`'s return value might be as long as 19+2=21 bytes long (including the `'\0'`), which will overflow the 20-byte `dfname` buffer.

This minor bug is the result of a common off-by-one error: the programmer apparently forgot to include the string terminator `'\0'` when counting the number of bytes needed to store the return value from `queuename()`. The very complex calling pattern needed to trigger this pattern illustrates why this type of bug is so difficult for humans to find on their own and why automated tools are so well suited for this task.

We note that this coding error survived at least one manual audit (the bug predates version 8.7.5, and survived an extensive sweep of the code apparently inspired by CERT advisory CA-96.20).

For completeness, we explain some of the other warning messages in Figure 5. The warning about `'from@savemail()'` is caused by imprecision in the analysis. The relevant code looks something like this:

```
if (sizeof from
    < strlen(e->e_from.q_paddr) + 1)
     break;
strcpy(from, e->e_from.q_paddr);
```

A human would realize that the `strcpy()` is not reached

[1] Throughout this paper, we exclude comments and blank lines from our counts of code sizes.

[2] We haven't written exploit code to confirm this, but examination of the source suggests that standard attacks are likely to work.

```
Warning: function pointers; analysis is unsafe...
1.74user 0.07system 0:01.99elapsed 90%CPU
Probable buffer overflow in `dfname@collect()':
  20..20 bytes allocated, -Infinity..257 bytes used.
  <- siz(dfname@collect())
  <- len(dfname@collect()) <- len(@queuename_return)
Probable buffer overflow in `from@savemail()':
  512..512 bytes allocated, -Infinity..+Infinity bytes used.
  <- siz(from@savemail())
  <- len(from@savemail()) <- len((unnamed field q_paddr))
Slight chance of a buffer overflow in `action@errbody()':
  7..36 bytes allocated, 7..36 bytes used.
  <- siz(action@errbody())
  <- len(action@errbody())
...
```

Figure 5. Some example output from the analysis tool. This example is a small sample of some of the more interesting output from an analysis run of sendmail 8.9.3.

unless it is safe to execute. The tool does not find this proof of safety because the range analysis is flow-insensitive and thus blind to the `if` statement.

The warning about `` `action@errbody()' `` (another false alarm) is also instructive. The relevant section of code has the following form:

```
char *action;
if (bitset(QBADADDR, q->q_flags))
    action = "failed";
else if (bitset(QDELAYED, q->q_flags))
    action = "delayed";
```

We can readily see that $\mathrm{alloc}(\texttt{action}) = \mathrm{len}(\texttt{action})$ always holds for this code segment, so there is no safety problem. However, the "bounding box" range analysis is fundamentally unable to detect invariants describing the possible relationships between values of program variables—another form of imprecision—so it is unable to detect and exploit this invariant to prove the code safe.

In this case, the analysis can only assume that the string `action` may have as few as 7 bytes allocated for it but as many as 8 bytes copied into it. This happens fairly often: when a pointer can refer to multiple strings of different lengths, the analysis usually reports that its size and length both have the same range $[d, e]$, and when $e > d$ there is no way to rule out the possibility of a problem. We use several heuristics to try to detect this class of false alarms and prioritize all warnings: this class of violations of the safety property is labelled "Slight chance of a buffer overrun."

One aspect of this trial that is not apparent from Figure 5 is the large number of false alarms encountered (see Section 5.5). Weeding through the false alarms took a full day of staring at warning messages and source code. A developer already experienced in sendmail internals might have completed the task more quickly, but it would still undoubtedly be a time-consuming process.

5.3. Sendmail 8.7.5

Finding new security vulnerabilities is a compelling way to validate the effectiveness of the tool, but it requires considerable time with no guarantee of positive results. As a time-saving alternative, we applied the tool to old software known to contain serious vulnerabilities to see if the bugs could have been detected. Sendmail is one of the classic examples of an application that has been vulnerable to buffer overruns in the past. Since CERT reported several overruns in sendmail 8.7.5 (see CA-96.20), and since the next version was audited by hand to try to eliminate such bugs, we decided to use this as a test platform.

The tool found many potential security exposures in sendmail 8.7.5:

- An unchecked `sprintf()` from the results of a DNS lookup to a 200-byte stack-resident buffer; exploitable from remote hosts with long DNS records. (Fixed in sendmail 8.7.6.)
- An unchecked `sprintf()` to a 5-byte buffer from a command-line argument (indirectly, via several other variables); exploitable by local users with "`sendmail -h65534 ...`". (Fixed in 8.7.6.)
- An unchecked `strcpy()` to a 64-byte buffer when parsing stdin; locally exploitable by "`echo /canon aaaaa... | sendmail -bt`". (Fixed in 8.7.6)
- An unchecked copy into a 512-byte buffer from stdin; try "`echo /parse aaaaa... | sendmail -bt`". (Fixed in 8.8.6.)

Improved analysis	False alarms that could be eliminated
flow-sensitive	$19/40 \approx 48\%$
flow-sens. with pointer analysis	$25/40 \approx 63\%$
flow- and context-sens., with linear invariants	$28/40 \approx 70\%$
flow- and context-sens., with pointer analysis and inv.	$38/40 \approx 95\%$

Table 2. Expected reduction in false alarms from several potential improvements to the analysis.

- An unchecked `sprintf()` to a 257-byte buffer from a filename; probably not easily exploitable. (Fixed in 8.7.6.)
- A call to `bcopy()` could create an unterminated string, because the programmer forgot to explicitly add a `'\0'`; probably not exploitable. (Fixed by 8.8.6.)
- An unchecked `strcpy()` in a very frequently used utility function. (Fixed in 8.7.6.)
- An unchecked `strcpy()` to a (static) 514-byte buffer from a DNS lookup; possibly remotely exploitable with long DNS records, but the buffer doesn't live on the stack, so the simplest attacks probably wouldn't work. Also, there is at least one other place where the result of a DNS lookup is blindly copied into a static fixed-size buffer. (Fixed in 8.7.6.)
- Several places where the results of a NIS network query is blindly copied into a fixed-size buffer on the stack; probably remotely exploitable with long NIS records. (Fixed in 8.7.6 and 8.8.6.)

Most of these coding errors became a threat only because of subtle interactions between many pieces of the program, so the bugs would not be apparent from localized spot-checks of the source. This seems to be a good demonstration of the potential for finding real vulnerabilities in real software.

To our knowledge, none of the vulnerabilities found in sendmail 8.7.5 by our tool have been described publicly before.

5.4. Performance

In our experience, the performance of the current implementation is sub-optimal but is usable. For example, the analysis of sendmail (about 32k lines of C code) took about 15 minutes of computation on a fast Pentium III workstation: a few minutes to parse the source, the rest for constraint generation, and a few seconds to solve the resulting constraint system.

The prototype generates extensive debugging output and has not been optimized, so we expect that the analysis time could be reduced with additional effort. On the other hand, the time required to examine all the warnings by hand currently dwarfs the CPU time needed by the tool, so better performance is not an immediate priority. For now, the most important property of the system is that it scales up readily to fairly large applications[3].

5.5. Limitations

The main limitation of the prototype is the large number of false alarms it produces, due to imprecision in the range analysis. As a consequence, a human must still devote significant time to checking each potential buffer overrun.

Our tool generates 44 warnings marked `Probable` for sendmail 8.9.3. Four of these were real off-by-one bugs, which leaves 40 false alarms. Despite the high success rate (1 in 10 warnings indicated real bugs), eliminating the false alarms by hand still requires a non-negligible level of human effort.

One way to reduce the number of false alarms requiring human attention is to trade off time for precision in the integer analysis. For example, we could envision moving to a flow-sensitive or context-sensitive analysis. This obviously raises the question of which improvements are worth the effort and at what cost. To estimate the potential benefits of various possible improvements to the analysis, we classified—by hand—the causes of each false alarm in sendmail 8.9.3. See Table 2 for the results. (A *linear invariant* is a simple, linear relationship between program variables—such as $\mathtt{x} + \mathtt{y} < 5$ or $\text{alloc}(\mathtt{buf}) \geq \mathtt{buflen}$—that holds in all program executions.)

These figures suggest that, in retrospect, it might have been better to use a more precise but slower analysis. We expect that standard analysis techniques (such as SSA form [13], Pratt's method [49] or Shostak's loop residues [56], and a points-to analysis) could be used to improve on our current prototype by an order of magnitude or more. However, significant engineering effort is probably required to get there.

Despite the unwieldy number of false alarms produced by our tool, our approach is still a substantial improvement over the alternative: in a typical code review, one would identify all the potentially unsafe string operations (perhaps using `grep`), trace back all execution paths leading to those unsafe operations, and manually verify that none of them lead to exploitable overruns. For comparison, there are about 695 call sites to potentially unsafe string operations

[3]We have no experience with very large applications, e.g., programs with hundreds of thousands of lines of code, so it remains unknown how our techniques scale up to such program sizes.

in the sendmail 8.9.3 source which would need to be manually checked in a typical code audit—15× more than the number that must be examined with our tool—so we conclude that our tool is a significant step forward.

One important gap in our understanding of the prototype's limitations is that it is difficult to rigorously measure the false negative rate. As a first approximation, we may examine all the buffer overruns in sendmail that have been fixed in the three years since the release of version 8.7.5; any such bug not reported by the tool is a false negative. To our knowledge, the only publicly-reported overrun in sendmail 8.7.5 is the `chfn` vulnerability [44], where a local user can overflow a 257-byte buffer by changing their gecos field in `/etc/passwd`. Due to pointer aliasing and primitive pointer operations, our tool does not find the `chfn` bug, although a better pointer analysis would have revealed the problem. A detailed manual examination of the source code revision history shows that a number of other buffer overruns in sendmail 8.7.5 have been quietly fixed without any public announcements[4]. As far as we know, our tool finds all of those vulnerabilities (see Section 5.3 for examples). This evidence suggests that our tool's false negative rate is non-negligible but still acceptable.

A final problem with the tool is that it does not provide as much information about each potential buffer overrun as we might like. As can be seen from Figure 5, the output shows only which buffer overflowed, not which statement was at fault. This ambiguity is arguably an unfortunate consequence of the constraint-based formulation. To improve the situation somewhat, we extended the constraint solver to report which variable(s) contributed to each violation of the safety property. This heuristic is not always reliable, but it does help.

6. Related work

LINEAR PROGRAMMING. Many papers have suggested using linear programming techniques to discover program invariants, including the simplex method, Fourier-Motzkin variable elimination [53], the Omega method [50], the SUP-INF method [5, 55], Shostak's loop residues [56], and algorithms for special classes of linear systems [30, 9, 38]. Typically, one combines linear programming with *abstract interpretation* over some simple domain (convex polyhedra, octagons, etc.) [10, 11, 23, 25, 26, 24, 52]. In this context, linear programming algorithms provide a tool for manipulating subsets of $\mathbb{Z}^k$, with operations such as $\cup$, $\cap$, projection, widening, and testing for feasibility. See especially [11] for an early example of a tool that infers linear invariants of small programs using abstract interpretation and the simplex method. Although the simplex-based techniques offer more precision than our range analysis, it is not clear how well they scale.

PARALLELIZING COMPILERS. One important application for array reference analysis is in discovering implicit parallelism in sequential Fortran programs [40, 4, 50]; however, those techniques do not seem to help with the buffer overrun problem because they focus too narrowly on the special case of loop optimization.

ARRAY BOUNDS CHECKING. One way to avoid buffer overruns is to use runtime array bounds checks. There are several implementations of array bounds checking for C, including SCC [3], gcc extensions [35], Purify [51], and BoundsChecker [46]. However, many of these tools impose a large performance overhead (instrumented programs are typically 2–3× slower than the original versions [3, 35, 8, 22]). As a result, the tools are usually used only for debugging, not for production systems.

To reduce the high cost of runtime bounds checking, several researchers have studied optimization techniques for eliminating redundant checks [22, 39, 57]. However, they typically focus on moving bounds checks to less frequently executed locations, rather than on eliminating all bounds checks. For example, hoisting bounds checks out of loops using loop invariants greatly reduces the performance impact of the bounds checks but cannot reduce the number of checks in the program's source code. Therefore, these optimization techniques are not well suited for proactively finding buffer overruns.

Other works have concentrated on eliminating all bounds checks for some type-safe languages. For example, Necula and Lee develop a certifying compiler for a type-safe subset of C that eliminates most bounds checks using Shostak's loop residues [45]. Also, Xi and Pfenning propose a method to eliminate runtime array bounds checking for ML with the help of some assertions added by the programmer to capture certain program invariants [60, 61]. Of course, none of these tools can eliminate buffer overruns in large legacy applications written in C.

RANGE ANALYSIS. Our approach to range analysis builds on much prior work in the literature, including early work on abstract interpretation [10] and range propagation [27] as well as more mature work on systems for static debugging [6], generalized constant propagation [59], and branch prediction [47]; however, our emphasis on analysis of large programs spurred us to develop new techniques with better scaling behavior.

CONSTRAINT-BASED ANALYSES. Philosophically, our analysis may be viewed as a constraint-based analysis [1]; however, it is unusual to incorporate arithmetic expressions in the set constraint language and solver (but see [28] for an important partial exception).

[4] We do not know whether these bugs were known to the sendmail developers, or whether they were fortuitously eliminated by the more-defensive programming style initiated in versions 8.7.6 and 8.8.0.

Note also that techniques for solving integer constraint systems may be found in the artificial intelligence literature [14, 32, 37, 58]; however, their algorithms typically stress generality for small problems ("hundreds of nodes and constraints" [14]) over scalability and thus are not directly applicable here.

LINT-LIKE TOOLS. Several commonly used tools [34, 18, 19] use static analysis and some heuristics to detect common programming errors (such as type errors, abstraction violations, and memory management bugs), but these tools don't detect buffer overruns.

Many practitioners have noted that `grep` can be a useful if crude test for finding buffer overruns by searching for all uses of unsafe string operations; however, a substantial time investment is often required to deal with the very large number of false alarms. Our results demonstrate an $15\times$ improvement over `grep` for the case of sendmail 8.9.3 (see Section 5.5).

PROGRAM VERIFICATION. ESC is an automated program checker for Modula-3 and Java that catches many programming errors at compile-time, using program verification techniques [15]. One disadvantage of ESC is that it requires coders to annotate module interfaces with information about expected pre- and post-conditions, but it can use this information to find a very large class of potential bugs.

STACKGUARD. Stackguard is a runtime tool which detects buffer overruns on the stack before they cause harm [12]. Stackguard imposes very little performance overhead and has been applied to large suites of applications, including an entire Linux distribution. Stackguard is a powerful tool that can serve as a strong deterrent against many existing buffer overrun attacks; however, it does not stop all overrun attacks, and thus should not be relied upon as the only line of defense.

7. Conclusion

This paper introduces a simple technique for the automated detection of buffer overrun vulnerabilities. Of particular significance is its ability to analyze large, complex programs. Because we trade off precision for scalability, our tool generates a relatively large number of false alarms, but it seems likely that a more sophisticated analysis could reduce the frequency of false alarms. We also demonstrated that our prototype implementation can find even very subtle bugs that elude human auditors. Although the tool is certainly no substitute for defensive programming or a careful code review, our experience suggests that it can complement and reduce the burden of these approaches.

Our implementation hinges on two key design considerations. First, treating strings as an abstract data type allows us to recognize natural abstraction boundaries that are obscured by the C string library. Second, formulating the problem in terms of integer range tracking allows us to build on techniques from program analysis.

We conclude that this provides a powerful and successful new approach to finding buffer overrun vulnerabilities. We attribute its success to the new methodology introduced, where we apply static analysis to security problems. One major advantage of static analysis is that it allows us to proactively eliminate security bugs before code is deployed.

Ideally, we would like a tool that could catch *every* buffer overrun. Although our tool does not detect all exploitable overruns, it still finds more than humans do, which shows that we have made real progress toward this greater goal.

8. Acknowledgements

We are grateful to a number of readers whose comments have substantially improved the paper, including Crispin Cowan, George Necula, Adrian Perrig, John Viega, and the anonymous reviewers. Thanks especially to Steven Bellovin (for bringing our attention to some of the limitations of dynamic testing in security applications) and to Manuel Fähndrich (for early discussions on the basic approach to modelling string buffers).

References

[1] A. Aiken, "Set constraints: results, applications, and future directions," *PPCP'94: Principles and Practice of Constraint Programming*, Springer-Verlag, pp.326–335.

[2] A. Aiken, M. Fähndrich, J.S. Foster, Z. Su, "A toolkit for constructing type- and constraint-based program analyses," *TIC'98: Types in Compilation*, Springer-Verlag, 1998, pp.78–96.

[3] T.M. Austin, S.E. Breach, G.S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," *PLDI'94*, ACM.

[4] U. Banerjee, *Dependence analysis for supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[5] W.W. Bledsoe, "The SUP-INF method in Presburger arithmetic,", Memo ATP-18, Math Dept., U. Texas Austin, Dec. 1974.

[6] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," *PLDI'93*, ACM.

[7] The `bugtraq` mailing list, `http://www.securityfocus.com/`.

[8] F. Chow, "A portable machine-independent global optimizer—Design and measurements," Tech. report 83-254, PhD thesis, Computer Systems Lab, Stanford Univ., 1983.

[9] E. Cohen, N. Megiddo, "Improved algorithms for linear inequalities with two variables per inequality," *SIAM J. Computing*, vol.23 no.6, pp.1313–1347, Dec. 1994.

[10] P. Cousot, R. Cousot, "Static determination of dynamic properties of programs," *Proc. 2nd Intl. Symp. on Programming*, Paris, Apr. 1976.

[11] P. Cousot, N. Halbwachs, "Automatic Discovery of Linear Restraints among Variables of a Program," *5th ACM POPL*, 1978, pp.84–97.

[12] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. 7th USENIX Security Conf.*, Jan. 1998.

[13] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, "An Efficient Method of Computing Static Single Assignment Form," *POPL'89*.

[14] E. Davis, "Constraint propagation with interval labels," *Artificial Intelligence*, vol.32 no.3, July 1987, pp.281–331.

[15] D.L. Detlefs, K.R.M. Leino, G. Nelson, J.B. Saxe, "Extended Static Checking," Compaq SRC Research Report 159, 1998.

[16] A. Diwan, K. McKinley, E. Moss, "Type-Based Alias Analysis," *PLDI'98*.

[17] M.W. Eichin, J.A. Rochlis, "With microscope and tweezers: an analysis of the Internet virus of Nov. 1988," *1989 IEEE Symp. Security and Privacy*.

[18] D. Evans, J. Guttag, J. Horning, Y.M. Tan, "LCLint: a tool for using specifications to check code," *SIGSOFT Symp. on Foundations of Software Engineering*, Dec. 1994.

[19] D. Evans, "Static detection of dynamic memory errors," *PLDI'96*.

[20] R. Ghiya, L.J. Hendren, "Putting pointer analysis to work," *POPL'98*.

[21] A.K. Ghosh, T. O'Connor, G. McGraw, "An automated approach for identifying potential vulnerabilities in software," *Proc. IEEE Symp. on Security and Privacy*, May 1998, pp.104–114.

[22] R. Gupta, "Optimizing array bound checks using flow analysis," *ACM Letters on Programming Languages and Systems*, vol.2 no.1–4, Mar.–Dec. 1993, pp.135–150.

[23] N. Halbwachs, Y.-E. Proy, P. Raymond, "Verification of linear hybrid systems by means of convex approximations," *SAS'94: Static Analysis Symp.*, Springer-Verlag, 1994, pp.223–237.

[24] N. Halbwachs, Y.-E. Proy, P. Roumanoff, "Verification of real-time systems using linear relation analysis," *CAV'93: Computer Aided Verification*, Published in *Formal Methods in System Design*, vol.11 no.2, Aug. 1997, Kluwer Academic Publishers, pp.157–185.

[25] M. Handjieva, "Abstract interpretation of constraint logic programs using convex polyhedra," Tech. report LIX/RR/96/06, LIX, Ecole Polytechnique, May 1996.

[26] M. Handjieva, "STAN: A static analyzer for CLP(R) based on abstract interpretation," *SAS'96: Static Analysis Symp.*

[27] W.H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Trans. Software Engineering*, vol.SE-3 no.3, May 1977, pp.243–250.

[28] N. Heintze, "Set based analysis and arithmetic," Tech. report CS-93-221, Carnegie Mellon Univ.

[29] G. Helmer, "Incomplete list of Unix vulnerabilities," `http://www.cs.iastate.edu/~ghelmer/unixsecurity/unix_vuln.html`.

[30] D.S. Hochbaum, J.S. Naor, "Simple and fast algorithms for linear and integer programs with two variables per inequality," *SIAM J. Computing*, vol.23 no.6, Dec. 1994, pp.1179–1192.

[31] D. Holland, `http://www.hcs.harvard.edu/~dholland/computers/netkit.html`.

[32] E. Hyvönen, "Constraint reasoning based on interval arithmetic: the tolerance propagation approach," *Artificial Intelligence*, vol.58, 1992, pp.71–112.

[33] `http://www.infilsec.com/vulnerabilities/`.

[34] S.C. Johnson, "Lint, a C program checker," Computer Science Tech. report 65, Bell Laboratories, 1978.

[35] R. Jones, P. Kelly, "Bounds Checking for C," `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`.

[36] O. Kirch, "The poisoned NUL byte," post to the `bugtraq` mailing list, Oct. 1998.

[37] O. Lhomme, "Consistency techniques for numeric CSPs," *IJCN'93: 13th Intl. Joint Conf. on Artificial Intelligence*, vol.1, 1993.

[38] G. Lueker, N. Megiddo, V. Ramachandran, "Linear programming with two variables per inequality in poly-log time," *SIAM J. Computing*, vol.19 no.6, Dec. 1990, pp.1000–1010.

[39] V. Markstein, J. Cocke, P. Markstein, "Optimization of range checking," *SIGPLAN Notices*, vol.17 no.6, Proc. Symp. on Compiler Construction, June 1982, p.114–119.

[40] D.E. Maydan, J.L. Hennessy, M.S. Lam, "Efficient and Exact Data Dependence Analysis," *PLDI'91*.

[41] B.P. Miller, L. Fredricksen, B. So, "An empirical study of the reliability of Unix utilities," *CACM*, vol.33 no.12, Dec. 1990, pp.32–44.

[42] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murphy, A. Natarajan, J. Steidl, "Fuzz revisited: a re-examination of the reliability of Unix utilities and services," Tech. report CS-TR-95-1268, U. Wisconsin, Apr. 1995.

[43] T.C. Miller, T. de Raadt, "strlcpy and strlcat—Consistent, Safe, String Copy and Concatenation," *FREENIX'99*, USENIX Assoc., Berkeley, CA.

[44] Mudge, "Sendmail 8.7.5 vulnerability," post to the `bugtraq` mailing list, Sep. 1996.

[45] G.C. Necula, P. Lee, "The Design and Implementation of a Certifying Compiler," *PLDI'98*.

[46] NuMega BoundsChecker, `http://www.numega.com/products/aed/vc_more.shtml`.

[47] J. Patterson. "Accurate Static Branch Prediction by Value Range Propagation". *PLDI'95*, pp.67–78.

[48] Phrack Magazine, "The Frame Pointer Overwrite," Sep. 1999, vol.9 no.55.

[49] V.R. Pratt, "Two easy theories whose combination is hard," unpublished manuscript, Sep. 1977.

[50] W. Pugh, D. Wonnacott, "Eliminating false data dependences using the Omega test," *PLDI'92*, pp.140–151.

[51] Pure Atria Purify, `http://www.rational.com/products/purify_unix/index.jtmpl`.

[52] P. Raymond, X. Nicollin, N. Halbwachs, D. Weber, "Automatic testing of reactive systems," *Proc. 19th IEEE Real-Time Systems Symp.*, 1998, pp.200-209.

[53] A. Schrijver, *Theory of linear and integer programming*, Series in Discrete Mathematics, John Wiley & Sons, 1986.

[54] M. Shapiro, S. Horwitz, "The effects of precision of pointer analysis," *SAS'97: Static Analysis Symp.*, Springer-Verlag, pp.16–34.

[55] R. Shostak, "On the SUP-INF method for proving Presburger formulas," *J. ACM*, vol.24 no.4, Oct. 1977, pp.529–543.

[56] R. Shostak, "Deciding linear inequalities by computing loop residues," *J. ACM*, vol.28 no.4, Oct. 1981, pp.769–779.

[57] N. Sosuki, K. Ishihata, "Implementation of array bound checker," *POPL'77*, pp.132–143.

[58] P. Van Hentenryck, H. Simonis, M. Dincbas, "Constraint satisfaction using constraint logic programming," *Artificial Intelligence*, vol.58, 1992, pp.113–159.

[59] C. Verbrugge, P. Co, L.J. Hendren, "Generalized constant propagation: A study in C," *Compiler Construction, 6th Intl. Conf.*, LNCS 1060, Apr. 1996, pp.74–90.

[60] H. Xi, F. Pfenning, "Eliminating array bound checking through dependent types," *PLDI'98*, pp.249–257.

[61] H. Xi, F. Pfenning, "Dependent Types in Practical Programming," *POPL'99*.

A. Proofs of the theorems

Theorem 1. *Every constraint system has a unique least solution.*

Proof. There is a natural intersection operator on assignments, defined by $(\alpha \cap \beta)(v) = \alpha(v) \cap \beta(v)$. The intersection of two satisfying assignments is also a satisfying assignment, since if $\alpha(e) \subseteq \alpha(v)$ and $\beta(e) \subseteq \beta(v)$, then $(\alpha \cap \beta)(e) \subseteq (\alpha \cap \beta)(v)$. This means that if a constraint system has a least solution, it is unique: supposing there are two minimal satisfying assignments $\alpha \neq \beta$, then $\alpha \cap \beta$ is another, smaller satisfying assignment, which contradicts the assumption of non-uniqueness. To show that a least solution exists, let α be the intersection of all satisfying assignments. This intersection is non-empty, since the trivial assignment $v \mapsto [-\infty, \infty]$ satisfies every constraint system. Clearly, if β satisfies the system, then $\alpha \subseteq \beta$. Therefore, α is a satisfying assignment, and it is the least such. □

Lemma 1. *Let $f(x) = ax + b$ be an affine function in $\mathcal{AF}$ with $a > 0$, let R be a range, and let $S \subseteq \mathbb{Z}^\infty$ be the minimal range satisfying $R \subseteq S$ and $f(S) \subseteq S$. Then (1) $\sup S = \infty$ if $\sup f(R) > \sup R$; also, (2) $\inf S = -\infty$ if $\inf f(R) < \inf R$. If neither clause (1) nor clause (2) applies, we have $S = R$. If both clauses apply, we have $S = [-\infty, \infty]$.*

Proof. Let $R = [d, e]$, so that $\sup f(R) = f(e)$ (since f is monotone and $a \geq 1$). If $f(e) > e$, then $f(x) > x$ for all $x \geq e$ (since $a \geq 1$), so that $f(f(e)) > f(e) > e$, etc., and (1) is proved by induction. (2) follows similarly. Finally, if neither clause applies, then $f(R) \subseteq R$, and by the minimality of S we have $S = R$. □

Theorem 2. *We can solve the constraint subsystem associated with a cycle in linear time.*

Proof. Let $f(x) = ax + b$ be the affine function associated with the cycle. It suffices to show that the claim is true for $a > 0$. (If $a = 0$, the theorem is trivial; if $a < 0$, we traverse the cycle twice and consider $f \circ f$.) We show that it suffices to simply compute $f(\alpha(v_1))$ and compare the result with $\alpha(v_1)$. If $f(\alpha(v_1)) \subseteq \alpha(v_1)$, the least solution is $\alpha(v_1)$, and we can stop traversing the cycle. Otherwise, one or both of the clauses of the lemma apply. If both apply, we are done: set $\alpha(v_1) := [-\infty, \infty]$, and let the worklist algorithm trace out the implications for the v_j. If just one applies—say, clause (1)—we simply apply the lemma (a second time) to $R' = [\inf R, \infty]$, and we will be done after this second application. Computing f requires time linear in the length of the cycle, and propagating the result of the analysis around the cycle also requires linear time, so the whole process runs in linear time. □

B. More on constraint solving

In this section, we extend the basic algorithm presented in Section 4 to handle more general constraints. Let us first review how far we have come. We have an efficient algorithm that handles *simple constraints*, i.e., constraints of the form $av_i + b \subseteq v_j$. We have precise techniques for handling cycles. But the algorithms presented so far cannot handle arithmetic or min/max expressions on the left hand side of the constraint. Such constraints are relatively rare: for typical program analysis tasks, only about 2% of the constraints use complex arithmetical expressions, and less than 1% use min/max expressions. Nonetheless, they are still important enough that they cannot be ignored: consider, e.g., the C statement `sprintf(dst, "foo: %s %s", s, t)` to see why we need complex arithmetical expressions; also, modelling the standard library functions `strncpy()`, `snprintf()`, etc., clearly requires support for min/max expressions. We now describe how to extend the algorithm to handle these more general types of constraints.

Let $\mathcal{C}$ be a constraint system consisting of a system of simple constraints $\mathcal{C}'$ along with the complex constraint

$$a_1 v_1 + \cdots + a_n v_n + b \subseteq w. \tag{1}$$

We show how to construct a *reduced constraint system* $R_\alpha(\mathcal{C})$ containing only simple constraints, where the least solution to $R_\alpha(\mathcal{C})$ gives a useful lower bound on the solution to $\mathcal{C}$. The idea is to note that, for each j, (1) implies $a_j v_j + b_j \subseteq w$, where the b_j's are given by

$$b_j = b + \sum_{i=1,\ldots,n;\, i \neq j} a_i \alpha(v_i)$$

and α is any lower bound on the least satisfying assignment to $\mathcal{C}$. Thus we may take $R_\alpha(\mathcal{C}) = \mathcal{C}' \cup \{a_j v_j + b_j \subseteq w : j = 1, \ldots, n\} \cup \{\alpha(v_j) \subseteq v_j : j = 1, \ldots, n\}$, where the constants b_j are defined in terms of α as above. By construction, any satisfying assignment for $\mathcal{C}$ will then satisfy $R_\alpha(\mathcal{C})$.

In principle, this immediately yields an algorithm for solving a constraint system $\mathcal{C}$ containing complex arithmetical expressions: compute the least solution β to $R_\alpha(C)$ (using the algorithm in Figure 4) and set $\alpha := \alpha \cup \beta$, repeating these two steps iteratively until convergence. Termination is guaranteed since a cycle in $\mathcal{C}$ will induce a cycle in $R_\alpha(\mathcal{C})$ and thus will be processed efficiently by the HANDLE-CYCLE procedure.

In practice, our implementation exploits a more efficient approach, where we update the reduced system $R_\alpha(C)$ in place as α is updated. In the optimized algorithm, each change to $\alpha(v_i)$ in the algorithm of Figure 4 immediately forces an update to $R_\alpha(C)$ whenever v_i participates in the left-hand side of some complex constraint. This technique seems to work very well for our purposes, probably because complex constraints are relatively rare.

The approach used to handle to min/max constraints is currently very simplistic: the current implementation propagates information through min/max constraints but does not attempt to handle cycles containing min/max constraints. In principle, this could introduce "counting to infinity", but we have yet to encounter this behavior. This simplification reflects implementation considerations more than any fundamental difficulty with handling this type of constraints. If we ever encounter cycles containing min or max operations, we will implement the following extension of Lemma 1 to min/max constraints:

Lemma 2. *Let* $f(x) = \min\{g_1(x), \ldots, g_n(x), c\}$ *for* $g_1, \ldots, g_n \in \mathcal{AF}$ *and* $c \in \mathbb{Z}^\infty$*, where each* g_j *is of the form* $g_j(x) = a_j x + b_j$ *for* $a_j > 0$*. Let* R *be a range, and let* $S \subseteq \mathbb{Z}^\infty$ *be the minimal range satisfying* $R \subseteq S$ *and* $f(S) \subseteq S$*. Then (1)* $\inf S = -\infty$ *if* $\inf f(R) < \inf R$*; also, (2)* $\sup S = c$ *if* $\sup f(R) > \sup R$*. If neither clause (1) nor clause (2) applies, we have* $S = R$*. If both clauses apply, we have* $S = [-\infty, \infty]$.

Proof. Clause (1) is an immediate consequence of Lemma 1: if $\inf g_j(R) < \inf R$, then $-\infty \in S$, since $\inf f(S) \leq \inf g_j(S)$ for all S. To prove clause (2), note that $\sup S \leq c$, so it suffices to prove that $\sup S \geq c$. Suppose not, i.e., that $\sup S < c$. Let $e = \sup R$. Since $\sup f(R) > \sup R$, we have $g_j(e) > e$ for all j. Also, by Lemma 1, $g_j(x) > x$ for all $x \geq e$ and for each j. Since $f(S) \subseteq S$, we must have $\min\{g_1(S), \ldots, g_n(S), c\} \leq \sup S$, and so $g_j(\sup S) \leq \sup S$ for all j. At the same time, $R \subseteq S$ implies $\sup S \geq e$, so $g_j(\sup S) > \sup S$, a contradiction, which establishes clause (2). Finally, if neither clause applies, then $f(R) \subseteq R$, and by the minimality of S we have $S = R$. □

The algorithm could be further improved with slightly more sophisticated techniques. For example, we could compute the acyclic component graph (where each strongly connected component is shrunk down to one vertex) and then iteratively process each strongly connected component in topologically sorted order, using a depth-first search to discover the cycles within each strongly connected component. However, we have not explored these possibilities for optimization, because the existing solver is already much faster than necessary.