boa

Buffer Overrun Analyzer

Edo Cohen	Tzafrir Rehan	Gai Shaked
039374814	039811880	036567055
sedoc@t2	tzafrir@cs	gai@tx

Project in advanced programming - program analysis (236503)

Computer Science Department, Technion - Israel Institute of Technology

Chapter 1

Introduction

The C programming language is riddled with security vulnerablitilies, the most prominent of which is the risk of buffer overruns. Programs written in C are widely used today, many of them with legacy code. There are two possible methods to try and detect buffer overruns - static and dynamic analysis.

Dynamic analysis attempts to intercept access outside the bounds of an array during runtime. While this can be effective with the use of rigorous testing with high coverage to discover potential overruns, this method does not supply a sound analysis, and some buffer overruns may be undetected. In addition, runtime checking of array boundaries may be used to prevent buffer overruns, but incurs a performance cost.

Static analysis attempts to detect overruns by analyzing the source code. This does not require any changes to the code or runtime environment, and can be implemented soundly (no false negatives). However, a precise analysis is not scalable, and some simplifications must be made, at the cost of possible false positives (false overrun indication).

1.1 Goal

Given a C program that performs buffer manipulations, statically (at compile time) identify whether the program may perform array access out of the array bounds.

Boa should operate C code as is, without requiring the programmer to provide any meta information. The provided analysis must be *sound* - report 100% of the possible buffer overruns in the code, with no *false negatives*[1] reported, while minimizing the amount of *false positives*. Finally, the output should provide brief information about the source code instructions that cause each reported overrun.

1.2 Assumptions

In order to achieve these requirements we also defined several assumptions on the input code:

- The programmer knows that the C string library requires that a string must end with the NULL terminating character '\0', and will never mutate the last byte of a buffer in a way that will cause an overrun.
- The programmer never uses an uninitialized value.

[1] False negative is a buffer which Boa report to be safe while, under certain scenario, the program might access the buffer out of bounds.

Chapter 2

Boa

2.1 Tool Overview

Boa is designed as a series of modular components. Each component receives input from the previous component, and passes on its analysis to the next one.

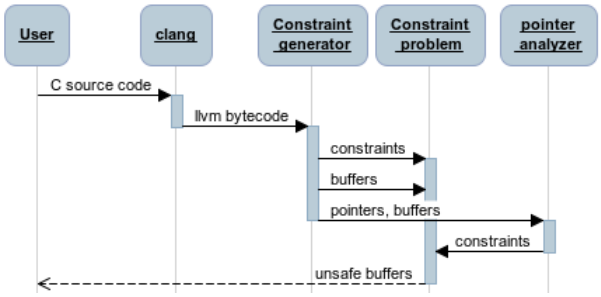

Figure 2.1: Main components and stages

- A C compilation module is compiled using the Clang open-source compiler. Clang outputs LLVM bitcode, which provides the benefits of SSA (Single Static Assignment) and a manageable intruction set.
- The constraint generator performs a single LLVM pass over the bitcode. During this pass, program statements are translated into linear constraints over constraint variables (section 2.2). In addition, the constraint generator collects a set of the buffers in the code, for use by the following components.
- The linear solver performs *taint analysis* on the set of constraints, removing any illegal constraints (section 2.3.1). Then the constraints are solved, and buffers with potential overruns are reported.
- The linear solver also offers a blame system (section 2.4), through which reported overruns may be root-caused. By reporting the set of statements which caused the overrun to be reported, the report may be validated as a possible overrun, or dismissed as a false positive.

The following sections expand on the different components in the boa framework.

2.2 Constraint Generator

Constraint generation is the process whereby LLVM operations are translated into a set of linear constraints. Constraint generation is performed in a flow-insensitive manner - meaning that the order of statements is not reflected in the constraints. This approach has several limitations, some as elementary as handling the statement '`i++`' (this is elaborated in Section 2.2). Unfortunately, flow sensitivity does

not scale well on even medium-sized programs. Some limitations of flow insensitivity are alieviated by LLVM's inherent SSA (Single Static Assignment) representation.

Boa applies a limited form of context sensitivity. String manipulation functions from the `string.h` and `stdio.h` libraries are individually modeled according to their effects on parameters and return value. User functions are modeled in a constext-insensitive manner - constraints for the function body are generated once for the formal parameters. On encountering a function call, Boa generates constraints between the actual parameters and the formal parameters.

Constraints are generated based on the approach outlined in [2, 3], with a few differences. Each integer '`i`' is modeled using a single pair of constraint variables '`i!max`' and '`i!min`', denoting the possible range of values for it. Each buffer '`buf`' is modeled using 3 pairs of linear constraint variables: '`buf!alloc!min`' and '`buf!alloc!max`' - which denote the possible range for the buffer's allocated size; '`buf!read!min`' and '`buf!read!max`' - which denote the possible range for read operations from the buffer; '`buf!write!min`' and '`buf!write!max`' - which denote the possible range for write operations to the buffer. Read and write operations must be modeled separately, to avoid adding implicit constraints between buffers (elaborated in section 2.2.3).

A fourth constraint variable pair, '`buf!used!min`' and '`buf!used!max`', is added for each buffer before passing the constraints to the linear solver. These variables are added with the following constraints:

$$\{\text{buf!write!max}\} \leq \{\text{buf!used!max}\} \leq \{\text{buf!read!max}\}$$

$$\{\text{buf!write!min}\} \geq \{\text{buf!used!min}\} \geq \{\text{buf!read!min}\}$$

This creates a simple variable denoting the possible range of access to the buffer.

The next sections describe the constraints generated for the different statements.

2.2.1 Integers

Integer assignments generate constraints on their values. Consider the C-statement

Listing 2.1: integer assignment

```
1 int i = 12;
```

The constraint generated from this assignment should reflect that i is in the range containing $[12, 12]$. Therefore, the constraints '$\{\text{i!max}\} \geq 12$', '$\{\text{i!min}\} \leq 12$' are generated and added to the constraint problem.

However, this case is a simple one. Consider the following code:

Listing 2.2: integer manipulation

```
1 int a = 10, b = 20;
2 int sum, prod, quotient;
3 sum = a + b;
4 prod = -3 * sum;
5 quotient = 2 / a;
```

The constraints generated for '`sum`' are still relatively simple:

$$\{\text{sum!max}\} \geq \{\text{a!max}\} + \{\text{b!max}\}$$

$$\{\text{sum!min}\} \leq \{\text{a!min}\} + \{\text{b!min}\}$$

Note that the range for sum is not immediately apparent, but is dependent on the value of two other constraint variables. The constraints generated for 'prod' are not as simple, since a negative factor flips the min and max values. Also, any multiplication between two variables or division by a variable results in a non-linear constraint, which our linear solver is incapable of handling. Therefore, results of such operations are considered unbounded, and the constraints generated are:

$$\{\text{quotient!min}\} \leq -\infty$$

$$\{\text{quotient!max}\} \geq \infty$$

2.2.2 Array Allocation

Listing 2.3: static array allocation

```
1  int i = 10;
2  char buf[10];
3  buf[i] = 'a';
```

Listing 2.4: dynamic array allocation

```
1  char *p;
2  p = (char*)malloc(4);
3
4  p[5] = 'a';
```

Array declarations such as the above supply the *alloc* value for arrays. Arrays may be statically or dynamically allocated - in both cases the treatment is similar. The code in the examples above would generate the following constraints:

$$\begin{aligned} \{\text{buf!alloc!max}\} &\geq 10 \\ \{\text{buf!alloc!min}\} &\leq 10 \end{aligned}$$

for the buffer '`buf`', and:

$$\begin{aligned} \{\text{dynamic!alloc!max}\} &\geq \{\text{i!max}\} \\ \{\text{dynamic!alloc!min}\} &\leq \{\text{i!min}\} \end{aligned}$$

for the return value of the `malloc()` call.

In the case of `malloc()` calls, the parameter may be a complex int expression, this is handled as far as possible for linear constraints - see section 2.3.

2.2.3 Buffer Aliasing

Certain statements may result in a buffer alias. This occurs when a pointer is set to point to another pointer or buffer, possibly with an offset (i.e. `p = buf` or `p = buf + 10`).

The direct approach towards modeling this aliasing as a linear constraint is to define a single set of constraint variables for each buffer or pointer - `buf!used!max` and `buf!used!min`. However, this simple modeling creates an unwanted side-effect in solving a linear-inequality system. Consider the following code:

Listing 2.5: array aliasing

```
1  char buf[] = "A string";
2  char *p = buf;
3  int len = strlen(p);
4  p[20] = '\0';
```

The pointer `p` is aliased to the array `buf`. This aliasing should be modeled so that the call to `strlen` returns the widest (most conservative) possible bounds of access to `p`, which is at least as wide as the bounds of `buf`. Therefore the following constraints must be generated:

$$\begin{aligned} \{\text{p!used!max}\} &\geq \{\text{buf!used!max}\} \\ \{\text{p!used!min}\} &\leq \{\text{buf!used!min}\} \end{aligned}$$

However, the write into `p[20]` should be treated as a (possible) write into `buf[20]`, resulting in the following constraints:

$$\begin{aligned} \{\text{buf!used!max}\} &\geq \{\text{p!used!max}\} \\ \{\text{buf!used!min}\} &\leq \{\text{p!used!min}\} \end{aligned}$$

This combined set of constraints implies an even stronger constraint between the two variables - they are forced to be equal, meaning

$$\begin{aligned} \{\text{buf!used!max}\} &= \{\text{p!used!max}\} \\ \{\text{buf!used!min}\} &= \{\text{p!used!min}\} \end{aligned}$$

Now consider a common case where the pointer `p` points to several different buffers in different contexts. These simplistic constraints demand that all buffers are equal through transitivity. Obviously, this is not good enough.

Instead, as mentioned in section 2.2, buffers and pointers are modeled using two separate sets of constraint variables, `buf!read!max`, `buf!read!min` and `buf!write!max`, `buf!write!min`. By trreating read and write operations on buffers separately, this problem can be eliminated. An aliasing is translated into the following constraints:

$$\begin{aligned} \{\text{p!read!max}\} &\geq \{\text{buf!read!max}\} \\ \{\text{p!read!min}\} &\leq \{\text{buf!read!min}\} \\ \{\text{p!write!max}\} &\leq \{\text{buf!write!max}\} \\ \{\text{p!write!min}\} &\geq \{\text{buf!write!min}\} \end{aligned}$$

The constraints on the read variables ensure that reads on `p` will return the largest possible range of access performed on `buf`. The constraints on the write variables ensure that writes to `p` will always be be reflected in `buf!write`.

Finally, whenever a buffer is added to the constraint problem, the constraints to link the two sets are added. These are added only for the buffers themselves:

$$\begin{aligned} \{\text{buf!read!max}\} &\geq \{\text{buf!used!max}\} \geq \{\text{buf!write!max}\} \\ \{\text{p!read!min}\} &\leq \{\text{buf!used!min}\} \leq \{\text{buf!write!min}\} \end{aligned}$$

This method of alias handling also provides a conservative flow-insensitive pointer analysis for first-order pointers, by capturing all possible pointer assignments.

2.2.4 Direct array access

Arrays are manipulated in code in several fashions, the simplest of which is direct access by index. Consider the statement:

Listing 2.6: direct access

```
1 buf[10] = 'a';
```

The resulting constraints render down to the expected:

$$\begin{aligned} \{\text{buf!write!max}\} &\geq 10 \\ \{\text{buf!write!min}\} &\leq 10 \end{aligned}$$

The constraint generator treats the value stored into `buf` as any other integer assignment, as described above.

2.2.5 Functions

Buffers are often manipulated through function calls, as parameter or return value. The constrain generator distinguishes between three different types of functions, and each is handled differently.

2.2.5.1 Functions With Definition

Listing 2.7: user functions

```
1 int foo(char *p, int x) {
2  // Do something with p and x
3 }
4
5 int main() {
6  char buf[10];
7  int res = foo(buf, 10);
8 }
```

These are functions with the definition available for analysis, usually functions that are defined in the same compilation unit. The constraint generator analyzes these functions in its single llvm pass, and generates constraints which may include its formal parameters and return value. In the example above, the constraints would contain `p` and `x`, with possible constraints on the int return value.

Whenever a call to such a function is encountered, the actual parameters are constricted to the formal parameters. In the example above, the call to `foo()` would result in `p` being aliased to `buf` (as described above), and the constraints

$$\begin{aligned} \{\text{x!max}\} &\geq 10 \\ \{\text{x!min}\} &\leq 10 \end{aligned}$$

In addition, `res` is constrained to the return value, resulting in

$$\begin{aligned} \{\text{res!max}\} &\geq \{\text{foo!max}\} \\ \{\text{res!min}\} &\leq \{\text{foo!min}\} \end{aligned}$$

This flow-insensitive analysis is conservative, but may result in false buffer overrun warnings. It is possible to add some context sensitivity to this analysis by creating a separate instance of the formal parameters for every function call, and duplicating the constraints for them. Previous work [2] has attempted this and discovered a small reduction in false positives, at the price of inflating the constraint problem, impacting performance.

2.2.5.2 Known Library Functions

Some functions do not have a definition supplied, but are so frequently used that they must be analyzed. The constraint generator contains a bank of commonly used library functions which manipulate buffers, and models their effect upon their parameters and return value. For example:

Listing 2.8: library functions

```
1   char *str1 = "longer␣than␣ten", *str2 = "short";
2   char buf1[10], buf2[10];
3   strcpy(buf1, str1);
4   strcpy(buf2, str2);
```

The function strcpy copies its second parameter into the first parameter. Thus, for the example above the following constraints are created:

$$\begin{aligned} \{\text{buf1!write!max}\} &\geq \{\text{str1!read!max}\} \\ \{\text{buf1!write!min}\} &\leq \{\text{str1!read!min}\} \\ \{\text{buf2!write!max}\} &\geq \{\text{str2!read!max}\} \\ \{\text{buf2!write!min}\} &\leq \{\text{str2!read!min}\} \end{aligned}$$

The functions which are modeled are mostly from `string.h` and `stdio.h` with several others (`malloc` for example). This is performed in a conservative manner, meaning the worst case scenario is always assumed.

2.2.5.3 Unknown Functions

Some functions called are neither of the two above cases. For these functions, the constraint generator cannot assume anything about the body of the function. In order for analysis to be conservative, all buffer parameters are considered to be accessed outside of bounds, and any int return value is considered unbounded.

This is not as great a problem as it may appear. The vast majority of string manipulations is performed by user functions and `string.h`, `stdio.h` functions, while the blame system described later on provides a mechanism for detecting irrelevant overruns which result from unknown functions.

2.3 Linear Solver

The constraints generated represent a linear problem, and each solution of the problem suggests a set of ranges for the values each integer may recieve and the allocation and usage of each buffer. As we aim to find the tightest ranges, we direct our linear solver to find a solution maximizing -

$$Goal = \sum_{\textbf{Buffers}} \Big[\{\text{buf!used!min}\} + \{\text{buf!alloc!min}\} - \{\text{buf!used!max}\} - \{\text{buf!alloc!max}\} \Big]$$

A solution satisfying this goal will maximize the lower bounds and minimize the upper bounds of each buffer access, and thus assure we get the tightest solution.

Once we have the solution[1] we test each buffer to verify that -

$$\{\text{buf!used}\} \subseteq \{\text{buf!alloc}\}$$

Which means -

$$\begin{aligned} \{\text{buf!used!max}\} &< \{\text{buf!alloc!min}\} \\ \{\text{buf!used!min}\} &\geq 0 \end{aligned}$$

Note that we stick to the size and numbering conventions of C, safe access to a buffer of size n is any access to the cells $0 \ldots n-1$. If the solution does not satisfy one of the constraints, we report a possible buffer overrun in this specific buffer.

2.3.1 Handling infeasible problems

In many cases, the constraints we generate create an infeasible linear problem. The simplest example of such case is -

Listing 2.9: increment

```
1 int i;
2 i++;
```

The constraints generated from the second line will be -

$$\begin{aligned} \{\text{tmp!max}\} &\geq \{\text{i!max}\} + 1 \\ \{\text{i!max}\} &\geq \{\text{tmp!max}\} \end{aligned}$$

Which is obviously an infeasible set of constraints. The same problem holds in many different cases, including *strcat* (which concatenates one string to the end of another, and therefore implies an equivalent set of constraints to the string used length).

When our linear solver discovers that the constraints problem we have generated is infeasible, we wish to make the smallest change to the problem and make it feasible once again. There is a great body of work in the area of finding and eliminating IIS (*irreducibly inconsistent system*), and we follow the algorithms and terminology of Chinneck and Dravnieks[1]. The common and naive approach is the deletion filtering -

[1] The solution is a set of integer values, one for each of the problem variables, such that all the constraints are satisfied and the *Goal* value is maximized

1. input: Q is an infeasible set of constraints
2. [try to delete] FOREACH $q_i \in Q$ DO:
   (a) Test whether $Q \backslash q_i$ is feasible -
      i. IF infeasible - set $Q = Q \backslash q_i$
3. Q is an IIS

After one iteration the algorithm returns an IIS, which can be removed from the original problem. In case there are several IISs in the original problem - the algorithm should repeat until the problem become feasible. We have implemented this approch at first, and it did work well on small pieces of code, but naturally did not scale well - on the same testing system described in chapter 4 it took more than half an hour to eliminate the IISs in the 400 lines of source of *md5* library, and more than 8 hours to find the blames (section 2.4) as well. Therefore we read further and implemented an elastic filter for eliminating IIS.

2.3.1.1 Elastic filter

The main idea behind elastic filtering is adding a new *elastic variable* to each constraint, allowing it to *strech* and thus make the problem feasible. The elastic variable is simply a new variable, which appears on a single constraint, and is confined to non-negative values. The constraints are of the form -

$$\sum a_j X_j \geq c$$

And with the elastic variable added -

$$\sum a_j X_j + e \geq c$$

The elastic variable gives the linear solver one more degree of freedom for each constraint, and therefore makes it feasible. Using the elastic variable (i.e. - assigning it a non-zero value) is reffered to as *streching*. Once the linear system is feasible we can solve the new problem, but instead of solving it in order to maximize the original target value, we try to minimize the effect of the *elastic variables*. In this case the problem is feasible - therefore a solution can be found, and since we direct the linear solver to minimize the values of the elastic variable - each elastic variable assigned a non-zero value marks a constraint which is part on an IIS (i.e. - if it's elastic variable did not exist the problem was infeasible). Formally -

1. Initialize $S = \emptyset$ (will hold the IIS)
2. Q is an infeasible set of constraints of the form -

$$q_i : \sum_{j=1}^{n_i} a_{i_j} X_{i_j} \geq c_i$$

   where a_{i_j}, c_i are constants and X_{i_j} is a variable of the constraint problem.
3. Add an *elastic variable* e_i to each constraint $q_i \in Q$ such that - $q_i : \sum_{j=1}^{n_i} a_{i_j} X_{i_j} + e_i \geq c_i$
4. Limit the elastic variables to accept non-negative values, and set the goal of the linear problem to minimize the sum of the elastic variables - $Goal = -\sum e_i$
5. WHILE the problem is feasible -
   (a) Solve the linear problem, for each elastic variable e_i -
      i. If $e_i > 0$ -
         A. $S = S \cup \{q_i\}$
         B. remove e_i $(q_i : \sum_{j=1}^{n_i} a_{i_j} X_{i_j} \geq c_i)$
6. return S

The most time-consuming part of the process is trying to solve the constraint problem (the linear solver uses Simplex). The delete filter tries to solve n problems (n - the number of constraints in the original problem) and return one IIS. On the other hand - the elastic filter perform only m iterations (m - the number of constraints in the smaller IIS in the problem), and return m constraints of each of the IISs in the problem. A typical IIS in the problems generated by our constraint generator consist of 3 or 4 constraints, thus we have predicted the performence of the elastic filter will be mush better than the deleting filter, and indeed (as we describe in detail in chapter 4) the results achived using the elastic filter were much better.

2.3.1.2 Removing constraints

Once we located the IIS we aim to remove the problematic constraints - in order to make the problem feasible, but without changing much of the semantics of the constriants. More specifically, we must ensure we do not harm the soundness of our solution, removing an IIS must not turn an infeasible problem which was created based on a buffer overrun into a feasible problem with no overruns reported.

In order to preserve the soundness of the solution we take the following conservative approach - we remove all of the constraints in the IIS, but introduce other constraints instead -

- For each *max* variable in the removed constraints (e.g. {buf!used!max} or {i!max}) we introduce the constraint -

$$\{\text{var!max}\} \geq \infty$$

- For each *min* variable, we introduce -

$$\{\text{var!min}\} \leq -\infty$$

These constraints ensure that all of the variables which appear in the IIS accept their worst case values, thus conserving the soundness of the problem. This approch is not as aggresive as it may appear on first glance, since most of the infeasibilities Boa encounters are indeed caused by the incrementing operator (or similar cases such as *strcat*). These kind of statements, in flow insensitive analysis, mean that the value (of an integer) or length (of a string) goes to infinity - since we can not tell how many times, and under what conditions the command will be executed.

2.4 Blame system

In order to make Boa useful we aim not only to inform the programmer about possible overruns, but also to direct him into the problematic code, which made Boa generate the warning in the first place. We call this feature *blame,* since it allows Boa to blame specific operations in specific source lines, allowing the programmer to examine each warning quickly and understand whether there is a real overrun in the code or the warning is a false alarm.

As far as we know, previous works in this area did not deal with this issue ([3]) or used a third party tool in order to provide the user with all of the code references to the suspect buffer ([2]).

The input for our blame system is a list of *unsafe buffers*, buffers in which the linear solver detected a possible overrun. For each of the unsafe buffers the output is a set of constraints, this set should be the minimal necessary set of constraints for the generation of the overrun.

Note that the definition of this set resembles the definition of an IIS - in both cases we look for a minimal set of constraints which imply a global property of the constraints problem. Therefore we based our blaming system on reducing the blaming problem to the problem of finding an IIS, and then using the existing algorithms to find the desired set of constraints.

The reduction itself simply involves adding two constraints to the problem -

$$\begin{aligned} \{\text{buf!used!max}\} &< \{\boldsymbol{buf!alloc!min}\} \\ \{\text{buf!used!min}\} &\geq 0 \end{aligned}$$

These are the two constraints we use to ensure that all of the buffer accesses are safe, with one different - in this case the left hand expressions are constraint variables, and the right hand are values - 0 in

the second constraint and **the value** assigned to {buf!alloc!min} in the first solution of the constraint problem. That way the problem becomes infeasible, and removing the infesibility will ensure safe only accesses to the buffer left.

During the constraint generation proccess we add to each constraint a *blame string*, which contains a brief explanation about the nature of the command that caused this constraint (e.g. "malloc call", "add instruction" etc) and its source location (source file name and line number). Once the elastic filter returns the set of constraints which cause the infeasibility (and therefore - cause the overrun) we print the *blame strings* of these constraints.

2.4.1 Constraint priorities

In order to provide the user with the most relevant blame results, we introduce three levels of constraints

Structural constraints which model the relations between {buf!read}, {buf!used} and {buf!write}. These constraints are part of Boa's internal implementation (disscused in detail in section 2.2.3), and do not supply the user with information about her possible buffer overrun.

Buffer aliasing constraints disscussed in detail in section 2.2.3, which usually do not cause an overrun by themselves, but might be very important for the user to understand how writing to a pointer *ptr* causes an overrun on buffer buf.

Normal constraints are any other constraints, involving integer arithmetics, library function calls etc.

The cause of the overrun is usually a *normal constraint*, while *buffer aliasing* constrains might be vital for the understanding the connection between the line where the overrun actually happens and the buffer (decleration) itself. *Structural constraints*, on the other hand, will always be part of an IIS, but the are irrelevant for the user.

When we run the elstic filter in order to discover the blaming set we first allow the elstic filter to choose only *normal constraints*, and than run the elastic filter again, allowing it to choose *aliasing constraints* as well. That way we ensure that the most relelvant blame results will appear first, the aliasing blames will also appear, but the *structural constraints* will not distract the user from the actual problem in her source code. A similar approch is taken when we use the elstic filter to turn the initial infeasible problem into a feasible problem - in that case we allow the elastic filter to remove *normal* or *aliasing* constraints, but not *structural* constrains, since the latter reflect an internal structure we intend to keep - and can not be the cause of the overrun.

2.5 Implementation

Boa is realized as a LLVM optimizer plugin, and is compiled into a shared object file which is dynamically loaded by *opt*, the LLVM optimizer.

The interface provided by LLVM for our purposes is a use of the *Visitor* design pattern. We created a class named *boa* that subclasses LLVM's *ModulePass* class, and implements the method *runOnModule*. LLVM passes a *Module* (LLVM's representation of a whole program) to this method, which then runs Boa's full analysis on the module and exits.

Almost all of Boa's work is done in the class *ConstraintGenerator*. This class is responsible for analyzing every code instruction in the module, and adding instances of the class *Constraint* to an instance of the class *ConstraintProblem*.

A Constraint instance is a representation of an expression of the form $[C \geq a_1x_1 + ... + a_nx_n]$ with some attributes such as source code location.

While analyzing the source code instructions, instances of concrete implementations of the abstract class *VarLiteral* are created. The concrete implementations are *Integer, Pointer* and *Buffer*, and are used to represent the source code variables modeled by Boa.

After analyzing all instructions in the program, the method *Solve* of the constraint problem is called, and the solution of the problem is analyzed to find buffer overruns, which are then displayed to the user.

The basic relationships between Boa's classes can be seen in the attached class diagram, and more detailed documentation can be found in Boa's doxygen documentation[2] and inlined in the code.

[2] `http://boateam.github.com`

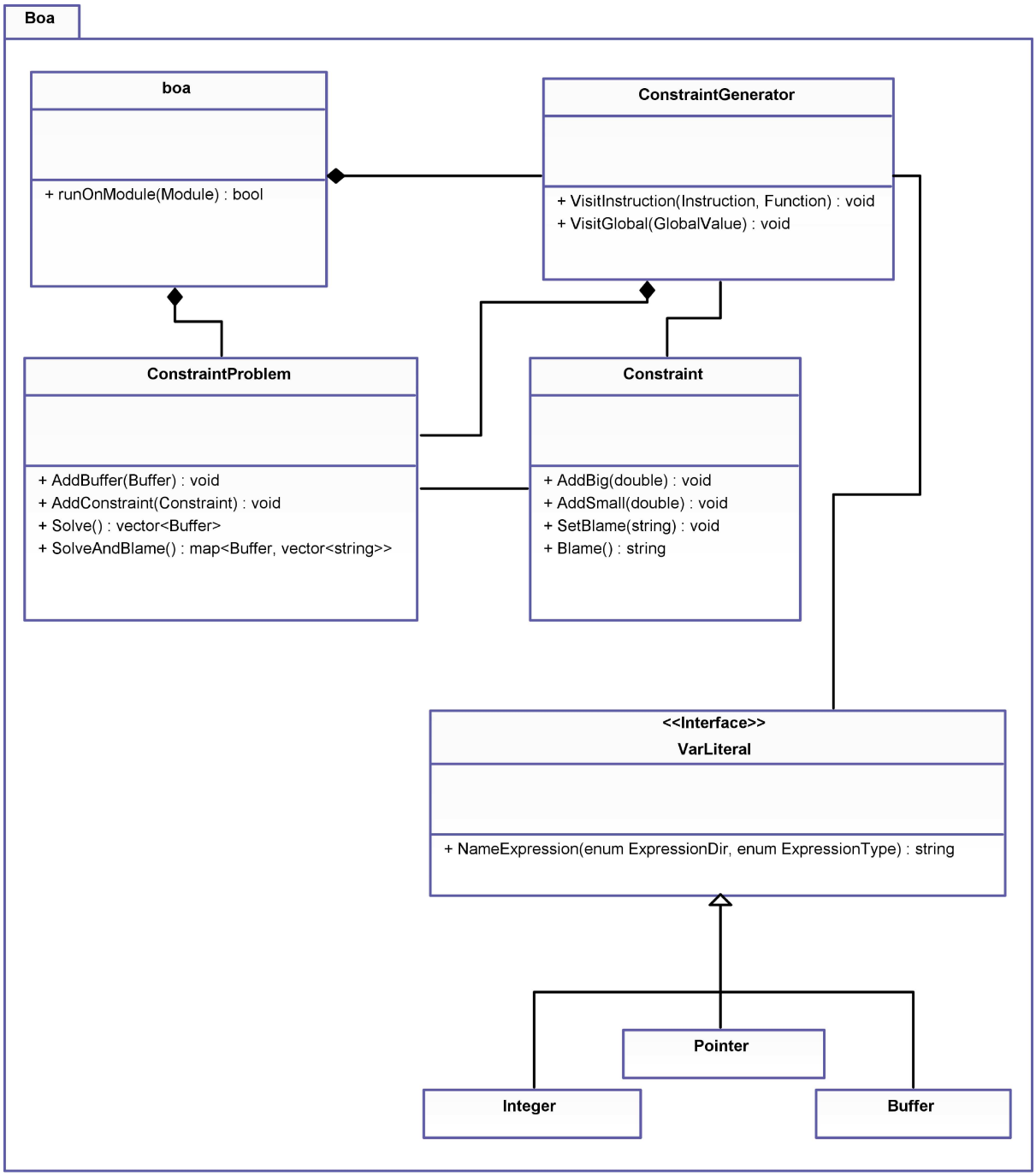

Figure 2.2: Class diagram

Chapter 3

Development Process

3.1 Initial Requirements

We set out to develop Boa by first defining a preliminary set of requirements:

- Provide analysis over any valid[1] C code as is, without requiring the programmer to provide any meta information.
- Provide *Soundness*: Report 100% of the buffer overruns in the code, with no *False Negatives* reported.
- Provide output in a machine readable format that is also easily readable by humans.
- Minimize false positives.

These requirements were defined over lenient assumptions as specified in the Introduction section.

3.2 Research and Technology Survey

After defining the initial requirements we began researching past work on the subject of buffer overrun static analysis, and set out to find which tools already exist that can be used to build Boa.

Our research reached the conclusion of using a linear problem solver, solving a set of linear constraints generated from each instruction in the source code, in order to find buffer overruns, as described in chapter 2.

We set a goal of using only open source tools which are publically available for free use under an Open-Source compatible license, and converged towards the use of the following tools:

- GLPK - the GNU Linear Programming Kit[2], available under version 3 of the GNU Public License. Used to solve the linear constraints using the Simplex Algorithm.
- The Clang C Front-End[3], available under the University of Illinois/NCSA Open Source License. We planned on using Clang's plug-in system in order to go over the code's Abstract Syntax Tree and generate linear constraints according to the instructions in the code.

We then set out to create prototypes. First we created example, "*Hello World*" style, programs that make use of Clang and GLPK's interfaces separately, and then combined the two prototypes to a single system that works according to our system design, produces constraints by traversing the source code, solves the constraints system and reports buffer overruns accordingly.

At this stage we created constraints such that any buffer access and use and any function call created an overrun on all buffers involved, and from this point on we could work on detecting all buffers in the program and reducing false positives.

[1] We define *Valid C* code as any code that is compiled by *gcc* with the -*Wall* flag without any warnings.
[2] GLPK - http://www.gnu.org/software/glpk/glpk.html
[3] Clang C Front-End - http://clang.llvm.org/

3.3 Test system

To facilitate the construction of Boa, we worked using *Test Driven Development.*

3.3.1 Black Box Testing

Since our entire system is hosted as a Clang plugin, *White Box* testing was not reasonable for our purposes.

We therefore created a *Black Box* testing system which allowed us to write tests for the full system behaviour.

A test case is comprised of a source code (.c file) and assertions (.asserts file), which define which buffers should be marked as having overruns, and which buffers should be marked as safe. Buffers may be defined by their name, by their source code location, or by both.

Example:

Listing 3.1: safe.c

```
1 #include <stdio.h>
2
3 int main() {
4   char cString[] = "I am safe!";
5   char overflow[8192];
6   puts(cString);
7   gets(overflow);
8   return 0;
9 }
```

Listing 3.2: safe.asserts

```
1 HAS ByName overflow
2 NOT ByName cString
3 BLAME ByName overflow unsafe function call gets
4 BLAME ByLocation tests/testcases/safe.c:7 gets
```

We then wrapped this testing system with scripts that run the tests over all available test cases and indicate failure if any test case failed, and incorporated this script into our makefile, such that running *make tests* compiles binaries as necessary and runs the tests on the result. This allows us to run all our tests after every change in the code, thus giving us confidence that old behaviour is not broken, and allowing us to start developing a feature by first defining a failing test case, then writing code until the test passes.

We later expanded the testing system to verify the existence and content of *Blame* information per a given buffer.

3.3.2 Unit Testing

For those cases where we could generalize common behaviour, we also wrote *White Box* unit tests.

We used *Google C++ Testing Framework*[4] (commonly known as *googletest*) as the framework that runs the tests so we could focus on writing the tests. In some cases we made adjustments to our code so that internal Boa classes could be subclassed.

3.4 Version Control

We used *git*[5] as our versioning control system. We leveraged the distributed nature of git and the cheap branching of git to create a workflow based on *feature branches*, where each of us worked on a feature on a separate branch of the code (usually working on more than one feature at a time), which was later merged into our integration branch, *"master"*.

[4] *http://code.google.com/p/googletest/*
[5] *http://git-scm.com/*

3.4.1 Online Hosting

We used Github[6] as an online code repository. This also gave us basic project management tools such as issue tracking and code reviews.

3.4.2 Code Reviews

Github allows developers to send each other code reviews (named *Pull Requests* in git terminology). We embraced this feature into our work flow such that all of the code that goes into *master* has to be reviewed by a team member.

For each code review, one of the team members which did not write the code would read the code and a discussion would start, where the reviewer requested improvement to the code, tests or documentation such that the code is correct and readable. By using this workflow, we ensured that the final code is not only correct, but also readable, since each code snippet had to be read by a programmer that did not see the code being written before being integrated into *master*.

3.5 Construction

During the Construction phase we used the following workflow:

- Create a local branch which is a copy of *master*.
- Write a failing test case.
- Write code that handles the failures of the test case, detects all possible overruns and does not report a false positive where not needed, until the test passes.
- Upload the branch to the online repository.
- Send a *Pull Request* to the team.
- Repeat.

Git's distributed nature allowed us to use this workflow with minimal overhead - On the one hand, orthogonal changes could be developed in parallel, with one programmer usually working on more than one change simultaneously in different branches. On the other hand, when programmer A needed to base his changes on code written by programmer B which was not yet reviewed and integrated, he could pull the work-in-progress branch from A's repository and base his own work without being blocked by the review process. *Git* also makes it easy to later merge additional changes made by programmer B (for example, as a result of the code review process).

3.6 Porting To LLVM

After we developed some initial features such as generating constraints according to integer arithmetic instructions, we began seeing some difficulties with our use of the Clang front-end as our constraint generation mechanism:

- The representation of various instructions are scattered across many nodes in Clang's representation of the AST, which made it hard to write generalized code that handles multiple cases.
- Clang's *RecursiveASTVisitor* implementation has only been written recently, and as cutting edge code it proved to be somewhat uncomfortable to work with.

Also, working directly on the AST of a high level language meant that the space of instructions that Boa needs to handle is quite large.

[6] http://github.com

We therefore decided it would be best to use the LLVM[7] (*Low Level Virtual Machine*) infratructure. Boa would first compile the code into LLVM bitcode, and then generate constraints according to that bitcode using an *LLVM Pass*.

We first prototyped an LLVM pass in the same manner we prototyped Clang, and then rebased Boa to use LLVM instead of clang in its architecture. All of our test cases were still valid (and at this stage, failing), so we continued our construction cycle, iterating until all existing test cases pass, usually using the previously written code as a reference as to how to handle a certain instruction (e.g., the constraints generated for a multiplication of two integers).

Within two weeks all our existing test cases passed, and we could refocus on further reducing false positives.

Having specific test cases defining the required handling of specific C instructions, a modular design where the mathematical work surrounding solving a linear constraint problem is separated from the handling of source code instructions, and the encapsulation of our algorithmic code all allowed us to very easily handle this major change in our system architecture, without having to rewrite the entire system from scratch.

3.7 Additional Requirements

As work on Boa progressed and enough bitcode instructions were properly handled, we could begin testing against code snippets from real world code. It soon became clear that it is not enough to notify the programmer which buffers are overrun, and Boa should also supply the programmer with information as to which code-level statements caused the buffer to be overrun.

Also, we noticed that most of the operations on character buffers in the real world are using library functions from *string.h* and *stdio.h*, and not direct array access, and that a common cause for buffer overruns is the use of *unsafe* library functions which do not use information about the length of the buffer, and may cause an overrun.

We therefore defined the following additional requirements:

- Provide machine readable *Blame* information which is also easily readable by humans.
- Model 100% of the functions in *string.h* and *stdio.h* such that all safe accesses will not be marked as unsafe, and all unsafe uses of library functions will mark the involved buffers as having overruns.
- Provide blame information per unsafe library functions such that if an unsafe function is used, the blame information will explicitly contain a warning against that specific function and will provide the source location of the unsafe function call.

[7] http://llvm.org/

Chapter 4

Results

We tested Boa on several widespread real world programs. We tested to see whether Boa discovers real buffer overruns, and also to evaluate the number of false alarms and their main causes. The source files used in all of the experiments are availble in Boa git repository[1].

4.1 Performance

Table 4.1 summarizes the performance of Boa on several programs, the reported running times are the results of experiments ran on a Dell Vostro 1310 laptop, with Intel Core2 Duo CPU T8100 2.10GHz and 2GB RAM running Debian GNU/Linux Wheezy (7.0.0), clang 2.9, llvm 2.9 and GLPK 4.43. On this humble configuration Boa can analyze few thousands lines of code within seconds, proving that the use of elastic filter did pay off and Boa can be used to efficiently analyze any reasonable piece of C code.

	fingerd	md5	syslog	ssh
Source lines	230	432	332	3483
Constraints	958	2932	894	3418
Running time	0.308s	0.916s	0.328s	1.348s
Running time (blame)	0.604s	2.532s	0.804s	8.557s
Buffers	34	2	15	94
Overruns reported	5	2	8	17
Real overruns	1	0	1	0

Table 4.1: Boa performance on various real world examples

4.1.1 Running time

The main factor affecting Boa's runtime is the number of generated constraints, which is proportional to the number of source lines manipulating integers, buffers or strings. The implementation of md5 is just 100 lines of source code longer than syslog, but since almost every line in the cryptographic algorithm of md5 perform buffer access/manipulation, md5 generates about 4 times the constraints of syslog, and this ratio also appear in the running times.

Another important factor is the amount of infeasible constraint sets have to be removed, but this measure can hardly be predicted simply by looking at the source code. When Boa runs the Blame algorithm, another important factor emerges - the number of overruns reported, each overrun causing the blame algorithm to solve an infeasible problem again, and thus the running time is approximately multiplied by the number of overruns detected.

[1] Boa's git repository - https://github.com/tzafrir/boa

4.2 Detailed disscusion

Over the next sections we will present and describe in details some of the possible overruns Boa detected in the programs menthioned above. We will use these examples to demonstrate how Boa can be used to detect overruns and how one can the blame feature to understand the behaviour of Boa and analyze false alarms.

4.2.1 fingerd

We tested Boa using *fingerd*, unix finger deamon. We altered the current source code to reflect the well known buffer overrun, used by the *Internet worm* in 1988. The overrun is caused by using the unsafe function *gets* to read data into the 1024[2] bytes buffer *line*. As far as we know, this is the only real buffer overrun in the 230 lines of source code.

Running on *fingerd* source, Boa reported overruns on 6 out of the 34 buffers. Next we present Boa's blame for three of them, and analyze the reason for the reported overrun -

line is the only real overrun in *fingerd*

```
line tests/realworld/fingerd/fingerd.c:85
- unsafe function call gets [tests/realworld/fingerd/fingerd.c:121]
- buffer alias with offset [tests/realworld/fingerd/fingerd.c:121]
```

The overrun discovered by Boa, and the real cause reported briefly. Note another result of *gets* - every other buffer access based on *line*'s length will be reported as an overrun.

rhost is a char buffer meant to hold the host name

```
rhost tests/realworld/fingerd/fingerd.c:86
- unknown function call realhostname_sa [tests/realworld/fingerd/fingerd.c:128]
- buffer alias with offset [tests/realworld/fingerd/fingerd.c:128]
```

This false alarm is caused by the use of *realhostname_sa*, from *socket.h*. This false alarm could be avoided if Boa would model *socket.h* functions, but even now the output lets the user identify the cause immediately and decide manually whether this call is safe or not.

malloc is a generic name for any buffer created by a malloc call, one can distinguish between two malloc calls by their source location (filename and line number)

```
malloc tests/realworld/fingerd/fingerd.c:141
- buffer alias - Phi Node
- buffer alias with offset [tests/realworld/fingerd/fingerd.c:149]
- buffer alias - Phi Node
```

This blame might seem wierd at a first look, how comes buffer alias alone cause an overrun? But the solution appears quickly by looking at the source lines (141, 149) reffered by the blame -

Listing 4.1: fingerd.c

```
141     if ((t = malloc(sizeof(line) + 1)) == NULL)
142       logerr("malloc: %s", strerror(errno));
```

...

```
149   for (end = t; *end; end++)
150     if (*end == '\n' || *end == '\r')
151       *end = ' ';
```

The programmer allocates a buffer large enough to include *line*, and then iterates through the array using the ++ operator on a pointer. Since Boa is a flow-insensitive analyzing tool, we can not assure that the inceremental pointer aliasing will be limited to the buffer size - and therefore Boa reports a possible buffer overrun.

[2]Back in 1988 *line* was 512 bytes, but it does not matter for the analysis.

4.2.2 MD5

Since the early working prototypes of Boa we have tested Boa against *RSA Data Security, Inc.* implementation of *MD5*, the a well known cryptographic hash function. Since MD5 is a cryptographic algorithm, the implementation consists of a lot of bitwise operations on the input string as well as on the predefined constant tables. Modelling of repeating bitwise operations precisely using linear equations is somewhat impossible, therefore Boa takes the safe (sound) side and declares buffer accesses based on such operations as unsafe.

Nevertheless, we bring MD5 as an example in order to demonstrate the limitations of the linear equations approch for static analysis, and also as an example for a code which causes Boa to generate a big amount of constraints per source line, and a large number of infeasible constraint sets. These challenges caused early prototypes of Boa to run for many hours before reporting possible overruns in MD5, while the current version fully analyzes and blames MD5's possible overruns within seconds.

4.2.3 syslog

Syslog is a simple application that writes messages into a unix system log, and yet another example of a widespread application with a buffer overrun vulnerability in its history. The syslog vulnerability was first reported on 1995, since syslog is a basic unix subroutine - the vulnerability affected not only syslog itself but many other applications, including *sendmail*, *super* and many other applications.

The buffer overrun in syslog happens when the log string to be printed is longer than *tbuf* (a buffer used by syslog in order to format and print the message), syslog calls *vsprintf* in order to write into the buffer, but does not verify that *tbuf* can hold the number of bytes to be copied. The solution to the overrun is even more interesting - in order to write such messages nevertheless, syslog mentions the buffer overrun - and checks the length of the written string after the hazardous call -

Listing 4.2: syslog.c

```
198   vsprintf(p, fmt_cpy, ap);
199   p += strlen(p);
200   cnt = p - tbuf;
201   if (cnt > sizeof tbuf) {
202     /* Panic condition. */
203     panic = 1;
204   }
```

If an overrun have occured, syslog sets the flag *panic* and then after printing the message to the log it will abort instead of returning, as the comment briefly explain -

Listing 4.3: syslog.c

```
253   /*
254    * If we had a buffer overrun, log a panic and abort.
255    * We can't return because our stack is probably toast.
256    */
257   if (panic) {
258     syslog(LOG_EMERG, "SYSLOG BUFFER OVERRUN - EXITING");
259     abort();
260   }
```

When Boa runs on syslog's source code it identifies the possible overrun on tbuf, but also reports 7 other possible overruns, all of them seems to be false alarms. Most of the false alarms happen simply because syslog calls POSIX library functions which Boa does not model, for example -

```
string literal "127.0.0.1"
- unknown function call inet_addr [tests/realworld/syslog/syslog.c:290]
- buffer alias with offset
string literal "%h %e %T "
- unknown function call strftime [tests/realworld/syslog/syslog.c:160]
- buffer alias with offset
```

We have already disscussed this kind of false alarms, and the means to analyze and prevent them previously as we disscussed *fingerd*.

4.2.4 ssh

Ssh (Secure Shell) is a very common program used to log into another computer over a network and to execute commands on a remote machine. Here we will present the results reported by Boa upon the major source code module of Tatu Yloen's implementation of ssh.

When we first tried to analyze ssh using Boa, we got 41 possible buffer overruns reported, out of 94 buffers in the source code. Most of these buffers were string literals, which are used as errors, warnings or log messages. Using the blame feature quickly revealed the reason for this massive amount of false alarms -

```
string literal "Too many identity files specified (max %d)"
- unknown function call fatal [tests/realworld/ssh/ssh.c:362]
- buffer alias with offset
string literal "Using rsh.  WARNING: Connection will not be encrypted."
- unknown function call log [tests/realworld/ssh/ssh.c:185]
- buffer alias with offset
string literal "setuid:  %s"
- unknown function call fatal [tests/realworld/ssh/ssh.c:189]
- buffer alias with offset
string literal "rsh_connect returned"
- unknown function call fatal [tests/realworld/ssh/ssh.c:566]
- unknown function call fatal [tests/realworld/ssh/ssh.c:592]
- buffer alias with offset
string literal "Secure connection to %.100s on port %d refused%s."
- unknown function call log [tests/realworld/ssh/ssh.c:582]
- buffer alias with offset
...
```

The reason Boa generated these false alarms is simple - this implementation of ssh uses four functions for errors, log and debug - *fatal, log, error* and *debug*. The implementation of these functions is not part of ssh main source module, and therefore Boa could not scan the implementation of these functions and forced to treat them as unsafe - functions that generate overrun on each of the buffers they accept.

As a solution to this exact problem, Boa provides the *-safe_functions* flag, which allows us to declare specific functions as safe, although the source code is not availble to Boa. Running Boa with *-safe_functions=log,error,fatal,debug* reduced the number of reported overruns to 17 out of 94.

Within the remaining 17 buffer overruns, there still are other false alarms generated because of ssh functions which are implemented in different modules (e.g. *process_config_line, perror*) but each of these functions appears only once or twice, thus locating and removing them from Boa's output is equivalent to analyzing any other false alarm manually - and therefore we did count these on the table above.

4.3 Conclusions

We have presented Boa's abilities to analyze decent amount of real, wide spread C source code within seconds. In all of these cases Boa was able to detect and briefly explain the real possible overruns in the source code. On the other hand - Boa does suffer from neglectable amount of false alarms, which are part of the nature of sound static analysis as a whole, and more specifically - the nature of flow- and context-insensitive analysis, and the use of integer ranges and other techniques that allows Boa's solution to scale well.

We have also presented several examples to how Boa's blame output can be efficiently used to analyze the false alarms, and how repeating false alarms which are caused by the same reason (e.g. - unknown function) can be easily eliminated.

We did not present here some more interesting buffer overrun detections - such as off by one (mainly because the programmer did not count C's '\0') or buffer overruns caused by integer arithmetics. These kind of overruns seems to be much more rare in the real sofware world, but Boa does cope well with these examples as can be seen by running Boa over the specific atomic test cases we have produced for such cases.

Bibliography

[1] Chinneck, J. W., and Dravnieks, E. W. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing 3*, 2 (1991), 157–168.

[2] Ganapathy, V., Jha, S., Chandler, D., Melski, D., and Vitek, D. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 345–354.

[3] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium* (San Diego, CA, Feb. 2000), pp. 3–17.