Enhancing Security Using Legality Assertions

Lei Wang, James R. Cordy, Thomas R. Dean
School of Computing
Queen's University, Kingston, ON, Canada
{wanglei, cordy, dean} @cs.queensu.ca

Abstract

Buffer overflows have been the most common form of security vulnerability in the past decade. A number of techniques have been proposed to address such attacks. Some are limited to protecting the return address on the stack; others are more general, but have undesirable properties such as large overhead and false warnings. The approach described in this paper uses legality assertions, source code assertions inserted before each subscript and pointer dereference that explicitly check that the referencing expression actually specifies a location within the array or object pointed at run time. A transformation system is developed to analyze a program and annotate it with appropriate assertions automatically. This approach detects buffer vulnerabilities in both stack and heap memory as well as potential buffer overflows in library functions. Runtime checking through using automatically inferred assertions considerably enhances the accuracy and efficiency of buffer overflow detection. A number of example buffer overflow-exploiting C programs are used to demonstrate the effectiveness of this approach.

1. Introduction

With the combined nature of both high-level and assembly languages, C is a popular programming language widely used for software development, especially systems programming. While it provides programmers with the potential to accomplish tasks with flexibility and efficiency, the absence of run-time error checking poses some difficulty in programming and calls for care and responsibility on the part of programmers. One of the run-time error checks that C does not perform is bounds checking, which has led to the notorious buffer overflow problem.

Buffer overflows have been the most common form of security vulnerability in the past decade [1]. A common example is the remote network penetration vulnerability [15] where an anonymous Internet user exploits buffer overflows to gain partial or total control of a host. Since buffer overflows provide an easy way for attackers to inject and execute malicious code, buffer overflow attacks constitute a substantial portion of all security attacks. For instance, 9 of 13 CERT advisories from 1998 involved buffer overflows [1] and in 1999, they accounted for at least 50% of advisories issued by CERT [2]. Several papers presenting reverse engineering and transformations that relate to security have been presented at recent WCRE conferences [22,24,25,26,27].

Considerable research has been conducted to investigate the buffer overflows and seek solutions to detect and prevent them [3,4,5,6,7,8,9,10,11,12,22,30]. However, these solutions attack the problem from different perspectives and are usually effective only against certain kinds of attacks and vulnerabilities. Some approaches also suffer from significant performance penalties or imprecision.

In this paper, we propose to use legality assertions [13] to enhance the security of C programs. Legality assertions implement dynamic checking of restrictions on legal programs written in a certain language. They are most suited in cases where some conditions are difficult to check statically. Legality assertions were originally used by the Euclid compiler to check if source programs obey the semantic restrictions of the Euclid language [13]. In our approach, they are used as annotations on the source program to check if array indices and pointers are in the legitimate range.

We have developed a transformation system using TXL [14] to automatically insert legality assertions in the source program where array elements are accessed or pointers are dereferenced. Valid ranges of buffers are dynamically maintained in additional integer variables. In each assertion, an array subscript or pointer dereference is checked against its valid range at the particular point in the program execution. This approach has several advantages: (1) Legality assertions are annotated in the source program automatically. (2) It does not change the original representation of arrays and pointers and also detects the potential buffer overflows in library functions without any modification to their representation. (3) It provides protection against a wide range of attacks, including both stack and heap smashing exploits. (4) Since the assertions check buffer range at run time, checking is more accurate and there are no false warnings.

The remainder of this paper is organized as follows. Section 2 gives some background and introduces related research. Section 3 presents our legality assertions for buffers in C programs. Section 4 describes our transformation process using simple examples. Section 5 describes our early experiments with the transformation system. Finally, Sections 6 and 7 discuss future work and conclusions.

2. Background and Related Research

This section introduces the notion of *legality assertions*, and the source transformation tool TXL that we use in our research. We examine the nature of buffer overflows in C and review related research in buffer overflow security.

2.1 Legality Assertions

A *legality assertion*, by definition, is "*a Boolean expression that has the value true if a specific restriction is met when the assertion is executed*" [13]. Legality assertions implement dynamic checking of language restrictions in legal programs and is used in cases where some conditions are difficult or impossible to check statically. Legality assertions, originally generated by the Euclid compiler, are based on the semantic restrictions of the language and are used to check language restrictions on values of non-manifest (run-time) constants, integer subranges, overflow in arithmetic expressions, boolean expressions, expressions in non-statement contexts (e.g. conditional expressions in `if` statement), assignment statements, array and collection (pointer) indices and parameter values. In our approach, we use assertions to check the legality of array indices and pointer dereferences. For example, Figure 1 shows the legality assertion for an array index in a C program.

```
int main(){
    int x = 3;
    char A[5] = {'1','2','3','4','5'};
    assert ((x+2)>=0 && (x+2)<5);
    A[x+2] = 'j';
}
```

Figure 1. An example C program with assertion

2.2 Source Transformation using TXL

TXL [14] is a programming language designed to support software transformation tasks. It combines features of both functional and rule-based programming, and supports unification, implied iteration and deep pattern match. A TXL program consists of two parts: a context-free, possibly ambiguous grammar describing the syntactic structure of the artifacts to be transformed, and a set of by-example formal transformation rules that use pattern-replacement pairs to describe the desired transformations. TXL has been used in a range of applications from design recovery to artificial intelligence, in both academic and industrial contexts [14].

In our project, source transformations in TXL are used to implement a static analysis of arrays and pointers in the C source code and to generate the augmentation of the source code with appropriate legality assertions.

2.3 Buffer overflows

A *buffer* is a contiguous allocated region of memory, such as an array in C. Since C provides the programmer with direct low-level memory access and pointer arithmetic without automatic bounds checking, a user can potentially write beyond the allocated memory for the buffer, which may result in unexpected behavior. Figure 2 depicts the general arrangement of the stack when a C function is called. In general, the stack grows downward (towards lower memory addresses) from right to left.

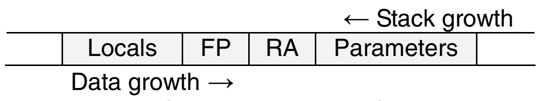

Figure 2. Stack frame for a C function call

In the case of a buffer overflow, when a fixed-sized memory allocation is used to store a variable-sized data entry in a local variable, the space allocated for the saved frame pointer (FP, the base address for local storage) and the return address (RA, the code address for the function to return to) can possibly be overwritten, which in turn can alter the program's execution path to execute malicious code input to the buffer [16]. A similar (but more complex) situation can occur on the heap, which may corrupt data on the heap to point to malicious code from a saved exception handler or function address [17].

In addition, the standard C library itself provides many unsafe functions that can write an unbounded amount of user input into a fixed-size buffer without any bounds checking. Examples of such functions include `strcpy`, `strcat`, `read`, `write`, `fgets` and `gets`.

2.4 Buffer Overflow Defenses

A number of techniques have been proposed to address buffer overflow vulnerabilities. Static analysis tools such as FlawFinder [3], RATS [4] and ITS4 [5] attempt to locate potential buffer overflows based on a lexical analysis of the program. These tools provide a fast and simple means for programmers to write more secure code. However, since they only operate on the lexical level of the program, the information they can provide is limited and imprecise. The LCLint Extension [6] and the integer analysis tool developed by Wagner et. al. [7], offer more advanced functionality. However, they have been found to generate a large number of false positives and the LCLint Extension requires programmers to manually add annotations to the source programs.

A more general technique to attack the buffer overflow problem is to add bounds checking in C programs [8, 9, 10, 11, 30]. At run time, extra information is maintained for every pointer: the address and size of the object to which it points. Every pointer dereference is instrumented to use the information to check whether the current value of the pointer is in bounds; if not, an error is reported. However, this approach has some serious drawbacks: It requires a change of data representation, which can be unacceptable in many applications, particularly systems programs and it adds high runtime overheads.

StackGuard [12] adds code to a compiled program to detect attacks on the return address. The advantages of this approach are that it requires no changes to the source code and introduces very little overhead; however, it does not address attacks on other vulnerable locations such as the heap.

Gemini [22], a tool presented by Chris Dahn and Spiros Mancoridis of Drexel University at WCRE 2003, uses TXL to secure C programs against the run-time stack buffer overflows by transforming stack allocated arrays into heap allocated arrays automatically at compile time. This approach eliminates the problem of stack buffer overflows by repositioning the buffers on the heap. However, this leaves the program vulnerable to heap exploits [28].

3. Legality Assertions for Buffers in C

In this section, we introduce our legality assertions for arrays and pointers in C. While Euclid was designed for expressing legality assertions, in C it is a much more challenging matter. For arrays, legality assertions are used to check the legal range of array indices. C array indices are zero-based, i.e. the index of the first element is `0`. Therefore, the legal range for an array index is `0` through `size-1`. The general form of the legality assertion for an array index is

```
    assert (index >= a.lowerBound &&
            index < a.upperBound)
```

The lower bound of an array index in C is always 0 and the upper bound of an array index is the maximum number of elements the array can hold.

A pointer is a variable that holds the memory address of another object, which we can refer to as the pointee, which can be referenced through the pointer using pointer dereferencing. When a pointer is initialized to the address of a pointee, it is usually assigned the lowest (starting) memory address of the pointee. Thus, the legal range for dereferencing through the pointer is between the lowest and highest memory address of the pointee. The general form of the legality assertion for a pointer dereference will therefore be:

```
    assert ((int)p >= p.lowerBound &&
            (int)p < p.upperBound)
```

Since a pointer is simply the numerical value of the address of a memory area, to simplify the comparison, we cast pointers into their corresponding integer values. Both lower and upper bounds are also integers. Unlike array boundaries, both the lower and upper bounds need to be calculated from the pointee. As pointers are most dangerous when the values they point to are accessed, legality assertions will be inserted where pointers are dereferenced usually using the "`*`" operation.

Many library functions in C take pointers as parameters and can be dangerous without bounds checking. Legality assertions can be used to check the state of buffers to ensure that no overflow will occur while the function is executed. Since each function takes different types and numbers of parameters, it is difficult to generalize a common assertion which applies to all functions. In the following, we introduce assertions for several library functions as a demonstration.

(1) strcpy (char *s1, const char *s2)
```
   assert((s1.ubound-s1.lbound)>=
          (s2.ubound-s2.lbound));
```
(2) strcat (char *s1, const char *s2)
```
   assert((s1.ubound-s1.lbound)>
          strlen(s2)+ strlen(s1));
```
(3) read/write (int fd, void * buf, unsigned count)
```
   assert (count <= buf.ubound-buf.lbound);
```
(4) fgets(char *str, int num, FILE * stream)
```
   assert((str.ubound-str.lbound)>=num);
```

4. The Legality Transformation Process

Our legality assertion transformation process is composed of six phases: *Preprocess*, *Mark up library*, *Unique rename*, *Add assertions*, *Remove markups* and *Unname* (shown in Figure 3). GCC is used to preprocess the source code in the first phase. When invoked with the '-E' flag, GCC resolves preprocessor statements without validating the input as C source code. The other five phases are all implemented using TXL [14].

4.1 Preprocess

C allows programmers to include various instructions to the compilers in the source code through pre-

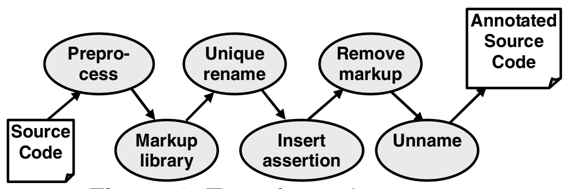

Figure 3. Transformation process

processors, such as macros, file inclusion and conditional compilation directives, which will be resolved at compile time. The use of preprocessors expands the scope of the C programming environment and enhances flexibility of the language. However, it also adds complication to source code analysis and transformation in that precise information may not be obtained unless the source code is preprocessed. Therefore, source code normally needs to be preprocessed by the C compiler before it is analyzed or transformed.

This is the case for our process also, and we use the GCC preprocessor as the first step in our process. However, unlike many analysis and transformation applications, automated legality assertions must be regenerated whenever the program is changed. Thus they are only of interest to the compiler, and preprocessing before analysis does not in any way limit or inconvenience the programmer using our system.

4.2 Markup Library Files

Because in general we do not have access to internal system library source, our transformation is mainly interested in programmer-written source code. While preprocessed code includes both standard files from the C library and user defined files, the included library files are separated from other code using markups and ignored during the transformation. A markup strategy, similar to that used to specify source code hot spots for the year 2000 problem [18], was adopted to mark and ignore the library contents.

Figure 4 shows an excerpt of preprocessed code. The code following "`# 1 "/usr/include/bits/sys_errlist.h" 1 3`" is included from the standard library files and the code following "`# 2 "exer1.c" 2`" is written by the programmer. Any code following preprocessor statements that contains "`/usr/include`" is marked with the XML tag `<lib> </lib>`. The markup process begins by finding all the library preprocessors includes and marking them with tags, after which the contents between a library preprocessor include and the next non-library statement is located and marked up.

4.3 Unique Naming C Variables

Based on C's namespace and scope rules, identifiers in different scopes and namespaces may have the same name. Compilers usually use a scope stack to keep track of the variables in different scopes and variables with same names will not be confused with each other. However, these variables may create ambiguity in static analysis. For example, it is possible that all of the following components in a C program could use the same name: structure tags, member variables in different structures, global variables, local variables, function parameters.

```
<lib> # 1 "/usr/include/bits/sys_errlist.h" 1 3
</lib>
<lib> # 27 "/usr/include/bits/sys_errlist.h" 3
</lib>
<lib> extern int sys_nerr;
</lib>
<lib> extern __const char* __const sys_errlist[];
</lib>
…
<lib> # 686 "/usr/lib/gcc-lib/i586-mandrake-
linux-gnu/3.2.2/include/stdio.h" 3
</lib>
# 2 "exer1.c" 2
int main () {
    …
}
```

Figure 4. Example of output of markup phase

A simple legal C program is shown in Figure 5. In this program there are seven entities with the same name `a`. Item 1 is a global variable. Item 2 is a structure tag. Items 3 and 4 are members of different structures. Item 5 is a local variable in `main()` that will mask the global variable `a`. Item 6 is a local variable in the `if` statement that will mask Item 5 in the scope of the `if` statement. Item 7 is a function parameter and will also mask the global variable `a` inside function `f()`.

To distinguish between these entities, a scope-based unique naming transformation [19] is used to give each variable identifier a unique name. Each unique name consists of the variable name and the name of the scope where the variable is visible. The scope name is prepended to the original variable name, separated with "S$", which is rarely used in C variable names. For

```
int a;          //1. global variable
struct a {      //2. structure tag
  int aS$a;     //3. member in structure a
};
struct b {
  int bS$a;     //4. member in structure b
};
int main () {
  int mainS$a =0;  //5. local variable
  if (mainS$a < 0){
    int main2S$a = 3;  //6. local variable
  }
}
void f (int fS$a) {  //7. function parameter
}
```

Figure 5. Example of renaming variables

example, after unique naming, the function parameter a, Item 7 in Figure 5, becomes fS$a. No scope names are added to global variables since they are unambiguous once all other items are uniquely named. To clearly present the remaining examples, we show the original names (i.e. not unique names).

4.4 Annotate Program with Legality Assertions

Now we are ready to do the real work. In this phase, information about arrays and pointers is collected through program analysis and for each array or pointer in the input program, two new limit variables (4 bytes each on 32 bit machines) are created to store boundary information. The boundary values are represented as integers and will be updated whenever the array or pointer is changed. Legality assertions are inserted in all places where buffers are accessed. In the assertions, the corresponding limit variables are used for boundary checking.

4.4.1 Annotation for Arrays. Adding assertions to array indices uses several steps:

(1) Add missing array sizes. The limit for an array index is between 0 and the size of the array. The size of an array can usually be obtained from the array declaration, where the size is given in square brackets (`[]`). However, C allows the size of an initialized array to be calculated from the number of initializers. Therefore, a normalization transformation is required to change the implicitly sized arrays into arrays with an explicit size by inserting the size in the square brackets (`[]`). If it is a character array, the size is the length of the string plus one more character for the null byte at the end of the string. Figure 6 shows a transformation rule to add missing array size in declarations as well as the input and output of this transformation rule.

Transformation rule:

```
rule addSubscripts
    skipping [markup]
    replace [init_declarator]
        RP [repeat ptr_operator] var [id]
        DE [declarator_extension]
        rest [repeat declarator_extension]
        IN [opt initialization]
    deconstruct DE
        '[ ']
    deconstruct * [list
        designated_initializer] IN
        L [list designated_initializer]
    construct N [number]
        _ [length L]
    construct newDE [declarator_extension]
        '[ N ']
    by  RP var newDE rest IN
end rule
```

Input: `int A[] = {1,2,3,4,5};`

Output: `int A[5] = {1,2,3,4,5};`

Figure 6. Add missing array size in declaration

(2) Add limit variable declarations to arrays. For each array, two new integer variables are created to store the lower and upper bound of the array. The lower bound is initialized to 0 and the upper bound to the size of the array as given in the declaration. The new variable names take the form of "`arrayName$lower`" and "`arrayName$upper`", so as to be distinguished from other variables. For external array variables, the boundary variables are declared as external without initialization. Values will be set in the file where they are defined and initialized.

(3) Add limit variable declarations to arrays in structures. The transformation in this step addresses arrays inside structure declarations. Similarly to the transformation in the previous step, boundary variables are added immediately after the array declaration in the structure. However, they will not be initialized until an instance of the structure is declared, since no initialization is allowed inside the structure declaration. Figure 7 shows transformation steps (2) and (3).

```
int A [5] = {1, 2, 3, 4, 5};
int A$lower = 0;   int A$upper = 5;
struct S {
    int C [5];
    int C$lower;   int C$upper;
};
```

Figure 7. Add limit variables to arrays

(4) Create temporary variables to store array subscripts. Legality assertions for arrays are added in places where an array is indexed. An array index can be a constant, such as `A[2]`, an expression, such as `A[x+1]`, `A[x++]`, or even a function call `A[f(x)]`. If the whole subscript expression is used in the assertion, for example, "`assert ((x++)>=0 && (x++)<A$upper));`", the value of `x` will be incremented two times. The same thing happens if a function call `f(x)` used as index has some side-effects. Therefore, to preserve the semantics of the input program, a new variable is created for each array subscript to hold the value of the subscript expression and the index is replaced by the new variable. The names of generated index variables are all in the form of "`i$N`", where `N` is a unique number.

The assignment statement to set the index variable is inserted just before the statement where the index appears. Due to the complexity of statement types, different rules have been composed to match the different patterns of `if`, `for`, `while`, `do-while` and `switch` statements. A simple optimization is performed at this step. If a subscript is a constant and is in the valid range, no new variable will be created for it and no legality assertions will be generated in the end.

(5) Create declarations for temporary index variables. Since new variables are introduced to hold index values, declarations for these new variables need to be added to the program. This transformation step finds all the new index variables in each scope, creates declarations and places them at the beginning of the scope, before the original declarations.

(6) Add assertions to array indices. Finally we are ready to generate the legality assertions themselves. In this step, the transformation system finds each array index in program statements and generates and inserts the actual assertions in the appropriate places. For example, assertions appear immediately before each simple statement that contains array indices (see Figure 8). For `if` and `switch` statements, the transformation must ensure that the assertions for array indices appearing in the conditional expressions are placed before the entire statements. However, in `for` and `while` statements, the assertions to arrays in conditional expressions must be placed before the statement and repeated inside the loop so that the indices will be checked in each cycle of the loop. In a `do` statement, the assertions for conditional expression are only needed inside the loop. Figure 8 shows the entire transformation output for arrays of an example program.

```
#include <assert.h>
int main () {
    long int i$1;
    long int i$2;
    long int i$3;
    int D [5] = {2, 4, 6, 8, 10};
    int D$lower = 0;
    int D$upper = 5;
    char str [6] = "hello";
    int str$lower = 0;
    int str$upper = 6;
    int n, m = 0;
    i$1 = m ++;
    assert (i$1 >= 0 && i$1 < D$upper);
    n = D [i$1];
    i$2 = n + 1;
    assert (i$2 >= 0 && i$2 < D$upper);
    if (D [i$2] > 0) {
        long int i$4;
        i$4 = i$2;
        assert (i$4 >= 0 && i$4 < D$upper);
        D [i$4] = D [1];}
    i$3 = m + 1;
    assert (i$3 >= 0 && i$3 < D$upper);
    while (D [i$3] < 0) {
        m ++;
        assert (i$3 >= 0 && i$3 < D$upper);
    }
}
```

Figure 8. Assertions to array indices

4.4.2 Annotation for Pointers. In the legality assertion for a pointer dereference, the pointer expression is checked against its valid boundaries. As before, code for maintaining runtime boundaries is generated through static analysis. Similarly to array bounds, two limit variables are created for each pointer to store the valid pointer range. The limit variable names are the pointer name suffixed with "`$lower`" and "`$upper`". The limit variables are initialized according to the initial value of the pointer and updated whenever the value of the pointer changes.

Assertions for pointers are inserted in places where a pointer is dereferenced. In the assertion, the integer value of the pointer is checked against its limit variables, which provides the latest valid range for this particular pointer. Since pointers are widely used, the transformation process is much more complicated than for arrays. Steps for transforming pointers are as follows.

(1) Create limit variables for each pointer. Limit variables are created for each pointer. If the pointer is initialized, the limit variables are initialized accordingly. For pointers in structures, only the limit variable declarations are inserted. The following examples show some cases of pointer initialization, for which the limit variables are initialized in different ways.

1) A pointer declared with no initialization. The limit variables declarations are created with no initialization.
```
e.g. int * p;
     int p$lower;
     int p$upper;
```
2) A pointer initialized to null. The limit variables are initialized to zero.
```
e.g. int * p = null;
     int p$lower = 0;
     int p$upper = 0;
```
3) A pointer initialized to an array or a structure. The limit variables are initialized to the start and end of the construct respectively.
```
e.g. int a[3]; int * p = a;
     int p$lower = (int)&a;
     int p$upper = (int)&a+sizeof(a);
```
4) A pointer initialized to another pointer.
```
e.g. int * p;
     int * q = p;
     int q$lower = p$lower;
     int q$upper = p$upper;
```
5) A pointer initialized to a string
```
e.g. char * s = "hello";
     int s$lower = (int)&s;
     int s$upper = (int)&s + 5 + 1;
```
6) A pointer array declaration. Two arrays are created to hold the limit values for each pointer in the array.
```
e.g. int * a [10];
     int a$lowerArry [10];
     int a$upperArry [10];
```

7) A pointer initialized by malloc().

```
e.g int * p = (int*) malloc(100);
    int p$lower = (int) p;
    int p$upper = (int) p + 100;
```

8) A pointer initialized by calloc().

```
e.g. int * p = (int*) calloc(2, 100);
     int p$lower = (int) p;
     int p$upper = (int) p + 2*100;
```

9) A pointer initialized by realloc().

```
e.g. int * p = (int*) realloc(p, 100);
     int p$lower = (int)p;
     int p$upper = (int)p + 100;
```

10) A pointer initialized by a user defined function. The limit variables will be inserted before pointer declaration and passed to the function, which initializes the limit variables inside the function.

```
e.g. int p$lower;
     int p$upper;
     int * p = f(&p$lower, &p$upper);
```

(2) Update limit variables in statements. A program, can change the object a pointer points to; when this happens, its limit variables must be updated accordingly. The assignment of a value to a pointer is essentially the same as in pointer initialization (but without the type name). Hence, this step includes almost the same cases as listed in the previous step. Limit variables are also updated in the same way. In general, for each pointer assignment, update statements are inserted following the assignment of the pointer. However, there are two special situations that must be treated separately. They are pointer reassignments in the conditions of `if` and `for` statements. For an `if` statement, the assignment of a pointer in the conditional expression is extracted and moved before the `if` statement as a separate statement and then pointer limits are updated as for other pointer assignments. For pointer assignments in `for` statement, update statements are inserted inside the subscopes.

(3) Add limit variables to pointer arguments in user defined functions. For pointers passed as arguments in user defined functions, the corresponding limit variables need to be passed to the function as well, as they may be used for pointer boundary checking inside the function. In this step, the transformation process checks all the arguments in user defined functions and adds the corresponding limit variables after each pointer argument. In order to keep the consistency and semantics of the program, transformation must be done consistently to arguments in function declarations, function definitions and function calls (a "coupled" transformation). An example is shown in Figure 9.

(4) Create new variables for each pointer returned in user defined functions. In the array transformation, new variables were created to store array subscripts to avoid side-effects, when the subscripts are arithmetic expressions or function calls. This same technique is used for pointers to be returned in user defined functions, since the pointer to be returned in a function may be a function call or an expression instead of a simple pointer variable. The new variable created for the returned pointer takes the form of "`ret$Var`". The new variable declaration is added to the beginning of the function definition. Before each `return` statement, the new variable is assigned to the returning pointer expression. The limits of this new pointer variable are computed as described in Step (2).

```
int i = 5;
int * p = & i;
int p$lower = (int) & i;
int p$upper = (int) & i + sizeof (i);
void foo (int *, int, int); //declaration
void foo (int * p, int p$lower, int p$upper) {
}      //function definition
int main () {
    foo (p, p$lower, p$upper); //function call
}
```

Figure 9. Add limit variables for pointer arguments

(5) Add limit arguments in functions returning pointers. For functions that return a pointer, two additional integer pointer parameters, `int * ret$lower`, `int * ret$upper`, are appended to the end of the parameter list. In the call to these functions, the additional formal parameters will be replaced by the addresses of the limit variables of the pointer that will be assigned the returned value. In this way, the limit variables can be updated from the value assigned inside the function. Figure 10 illustrates this transformation step.

```
int * foo (int i, int * ret$lower, int *
          ret$upper) {
    int * ret$Var;
    int ret$Var$lower;  int ret$Var$upper;
    int * p = (int *) malloc (i);
    int p$lower = (int) p;
    int p$upper = i + (int) p;
    ret$Var = p;
    ret$Var$lower = p$lower;
    ret$Var$upper = p$upper;
    {
        * ret$lower = ret$Var$lower;
        * ret$upper = ret$Var$upper;
        return ret$Var;
    }}
int main () {
    int p$lower;  int p$upper;
    int * p = foo (10, & p$lower, & p$upper);}
```

Figure 10. Add limits to functions returning pointers

(6) Add annotations to pointers. Finally, the actual legality assertions for pointer expressions are generated and inserted before the statements which contain the pointer dereferences. The general form of a pointer assertion is "`assert ((int) ptr>= ptr$lower && (int) ptr < ptr$upper`", where the current integer value of the pointer is checked against its valid boundaries. Similarly to assertions for array buffers, the transformation inserts the assertions in the appropriate place for each different type of statement. Special attention has been given to pointer deferencing combined with pointer increment and decrement, e.g. `*(++p)` and `*(--p)`. In these cases, the pointer must be incremented or decremented before it is dereferenced. The assertion that goes before it must therefore check its validity after the change. Figure 11 shows a simple example of pointer transformation and bounds checking.

```
#include <assert.h>
int main () {
    int a [3] = {1, 2, 3};
    int a$lower = 0;
    int a$upper = 3;
    int * p = a;
    int p$lower = (int) & a;
    int p$upper = (int) & a + sizeof (a);
    assert ((int) p >= p$lower && (int) p <
p$upper);
    * p = 4;
}
```

Figure 11. Example of pointer bounds checking

4.4.3 Library functions. Since the implementation of library functions is inaccessible, no change can be made to the function prototypes. Instead, buffer checking assertions are added as preconditions to library function calls. The following table summarizes the assertions created for some specific functions.

Table 1: Assertions to some library functions

Library Functions	Assertions
`strcpy (d,s)`	`assert ((d$upper - d$lower) >= (s$upper - s$lower));`
`strcat (d,s)`	`assert ((d$upper - d$lower) >= (strlen(d) + strlen(s)));`
`read/write (f,b,c)`	`assert (c <= (b$upper - b$lower));`
`fgets (s,n,fd)`	`assert (n <= (s$upper - s$lower));`

4.5 Remove Markup and Unname

After annotating legality assertions for all the buffers in the program, unique names in the program are reverted to their original names by a simple TXL program which removes the scope names that have been added to each variable. Finally, markup tags are removed from the included library files, yielding a compilable result.

5. Case Studies

To demonstrate the effectiveness of using legality assertions to detect buffer overflow in C, the transformation system has first been applied to example C programs from the security community with a variety of known stack and heap exploits.

5.1 CESG Vulnerability Code

The first example is a set of small security vulnerablility programs used as demonstration code by U.K. Government, Communications-Electronics Security Group (CESG), Network Defence Team [20]. The demonstration code from CESG consists of three small vulnerable programs, which present three different types of buffer overflow problems.

The first program reads a message from the first argument, copies it to a character array with a fixed size of 128 bytes through the library function `strcpy()` and then prints the message to the standard output. A stack buffer overflow will occur when the argument is larger than the character array. The transformation program inserts the legality assertion which checks the sizes of the message and the buffer. When the message is bigger than the buffer, the program aborts with an error message as shown as follows.

```
a.out: ex1.c:17: echo: Assertion `((int)
buf$upper - (int) buf$lower) >= ((int)
msg$upper - (int) msg$lower)' failed.
Aborted
```

Program 2 is a reimplementation of the first program with programmer's own bounded copy instead of using the library function. However, there is an off-by-one error in the programmer's bounds checking. The legality assertion checks the array index between 0 and 127 and the following error message is produced when the index is beyond its valid range.

```
a.out: ex2.c:22: echo: Assertion `i$1 >= 0
&& i$1 < buf$upper' failed.
Aborted
```

Program 3 takes two arguments and uses heap memory, which is allocated by `malloc()`, to store the arguments. The transformation program adds the boundary variables in the appropriate places and records the

Error message for buffer 1 overflow:
```
a.out: ex3.c:52: main: Assertion `((int)
buf1$upper - (int) buf1$lower) >= ((int)
argv$upperArry [i$1] - (int) argv$lowerArry
[i$1])' failed.
Aborted
```
Error message for buffer 2 overflow:
```
a.out: ex3.c:58: main: Assertion `((int)
buf2$upper - (int) buf2$lower) >= ((int)
argv$upperArry [i$7] - (int) argv$lowerArry
[i$7])' failed.
Hello Aborted
```

boundary value based on the pointer initialization. Legality assertions are inserted before the `strcpy()` functions. The assertions successfully detect the buffer overflow problem in either copying the first argument or the second argument

5.2 Server Example

The second example is a vulnerable server program, designed by Dr. Thomas Dean for his Operating System course offered at Department of Electrical and Computing Engineering, Queen's University. It contains a stack buffer overflow problem which may occur when an `fgets()` function attempts to write 129 bytes to a 100-byte buffer. The transformed program successfully detected the problem and produced the following error message.

```
$ ./server 1234567 2345
Student Number  1234567
Port Number  2345
Stack = bfffed6c
server waiting
socket = 4
1234567
server: getline.c:11: GetLine: Assertion
`130 <= buffer$upper - buffer$lower' failed.
Aborted
```

6. Future Work

Besides the examples discussed in Section 5, we are applying this method to a much more complicated application system *Ospfd* in *Zebra*, an advanced routing software package that provides TCP/IP based routing protocols, which has been found to have heap overflow problems [23]. Up till now, the transformation has been conducted on 24 files in *Ospfd* and 32 files in the *Zebra* library. The total number of lines of code to be transformed is over 500,000 lines after preprocessing.

Our current system has transformed all of the programs necessary to get a successful compile of the program, indicating that the approach will scale to large programs. However, these programs are significantly more complex than the example programs, exercising many of the ways in which C can manipulate memory. Our system has grown, but we do not yet cover every feature of the C language. Transformations are needed to handle the different forms of pointers including those made by `typedefs`, and the various forms of structure initialization.

While as our system matures, we are confident that these forms can be handled, this situation highlights are potential weakness in our method – it must accurately recognize all variants of deferencing and indexing in C programs, and C is notoriously inconsistent in its pointer handling syntax.

The particular C grammar specification we use for transformation also suffers from some ambiguities that we have not resolved. A clearer grammar definition will make the transformation process simpler and more efficient.

The Euclid compiler is able to remove many of the legality assertions using static analysis and a logic simplification algorithm. As a result, the runtime overhead of legality assertions in Euclid programs was less than 10%. Our current transformation system inserts legality assertions at every buffer access, such as array index and pointer dereference. Some of these generated assertions are clearly redundant. Our current examples are too small to obtain a good measure of the overhead and we do not yet eliminate unnecessary assertions. We intend to continue this work by adapting the Euclid approach along with more recent research in static analysis. Once we can eliminate the redundant assertions, we can then characterize the overhead of our approach in time and space.

A similar approach is explored by Beyer et al. [29]. They use model checking to improve on the approach taken by CCured [30]. CCured uses fat pointers and dynamic checking to implement pointer safety in C. Static analysis is used by CCured to determine which pointer variables must use fat pointers and if some dynamic checks can be removed.

7. Conclusion

Buffer overflows have ranked high among security vulnerabilities in the past decade. The most common attacks use an unchecked string copy to cause a buffer overrun, thereby overwriting the return address. When the function returns, control is likely transferred to the attacker's code. A number of techniques have been proposed to address such attacks. Some are limited to protecting the return address on the stack only; others are more general, but have undesirable properties such as large overhead, false positives or negatives.

Adding legality assertions to code does not remove the need for testing, and can be used in conjunction with testing. Penetration testing can be used to evaluate the effectiveness of the transformations, and the legality assertions can also be used to help with penetration testing. For example a test case that overflows a large buffer, but not sufficiently to cause an observable error will be flagged by the legality assertion associated with the buffer use.

The approach described in this paper uses legality assertions to check buffer overflows at run time. A transformation system has been developed to analyze the input program and annotate it with appropriate assertions automatically. It saves programmer effort in adding assertions manually and provides a fast and convenient way to help check the correctness of the program with respect to buffer vulnerabilities. The proposed approach has been proven to be able to detect examples of known buffer vulnerabilities in both the stack and heap memory. It has also demonstrated its ability to detect potential buffer overflows in library functions without any modification to their representation. Runtime checking through legality assertions also considerably enhances the accuracy and efficiency of buffer overflow detection. When an error is detected, the program terminates immediately, which prevents it from being exploited by attackers.

References

[1] Schneider, F. B., S. M. Bellovin, M. Branstad, J. R. Catoe, S. D. Crocker, C. Kaufman, S. T. Kent, J. C. Knight, S. McGeady, R. R. Nelson, A. M. Schiffman, G. A. Spix, and D. Tygar, *Trust in Cyberspace*. National Academy Press, 1999, Committee on Information Systems Trustworthiness, National Research Council.

[2] S. Bellovin, "Buffer Overflows and Remote Root Exploits", *Personal Communications*, October, 1999.

[3] D. Wheeler. FlawFinder. http://www.dwheeler.com/flawfinder/.

[4] SecureSoftware. RATS. http://www.securesoft.com/rats.php.

[5] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "ITS4: A Static Vulnerability Scanner for C and C++ Code", *16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December, 2000, pp.257-268.

[6] D. Larochelle, and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities", In *USENIX Security Symposium*, Washington D. C., 2001.

[7] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", *Network and Distributed System Security Symposium*, February 2000.

[8] S.C. Kendall, "Bcc: Run-time Checking for C programs", *USENIX Toronto 1983 Summer Conference Proceedings*, USENIX Association, El. Cerito, California, USA, 1983.

[9] J. L. Steffen, "Adding Run-time Checking to the Portable C Compiler", *Software – Practice and Experence*, 22(4), 1992, pp.305-316.

[10] R. W. M. Jones, and P. H. J.Kelly. "Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs", *Automated and Algorithmic Debugging*, 1997, pp.13-26.

[11] R. Hastings, and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors", *Proceedings of the Winter USENIX Conference*, 1992, pp.125-136.

[12] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks", *7th USENIX Security Conference*, San Antonio, TX, January, 1998, pp.63-77.

[13] D. B. Wortman, "On Legality Assertions in Euclid", *IEEE Transactions on Software Engineering*, Vol.4, July 1979, pp.359-367.

[14] J.R. Cordy, "TXL - A Language for Programming Language Tools and Applications", *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications, Electronic Notes in Theorectical Computer Science* 110, December 2004, pp. 3-31.

[15] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", Invited talk at *System Administration and Network Security (SANS)* 2000.

[16] A. One, "Smashing Stack for Fun and Profit". www.insecure.org/stf/smashstack.txt.

[17] J. Pincus, and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns", *IEEE Security & Privacy*, Vol. 2, No. 4, 2004, pp.20-27.

[18] J.R. Cordy, K.A. Schneider, T.R. Dean, and A.J. Malton, "HSML: Design Directed Source Code Hot Spots", *Proc. IWPC 2001 - IEEE 9th International Workshop on Program Comprehension,* Toronto, May 2001, pp. 145-154.

[19] J.R.Cordy, T. Dean, A. Malton, and K. Schneider, "Source Transformation in Software Engineering Using the TXL Transformation System", Special Issue on Source code Analysis and Manipulation. *Journal of Information and Software Technology*, 44(13), pp.827-837.

[20] U.K. Government, Communications-Electronics Security Group, Network Defence Team, 2004

[21] T.R. Dean, J.R. Cordy, A.J. Malton, and K.A. Schneider, "Agile Parsing in TXL", *Journal of Automated Software Engineering* 10, 4, October 2003, pp.311-336.

[22] C. Dahn and S. Mancoridis, "Using Program Transformation to Secure C Programs Against Buffer Overflows", *The 10th Working Conference on Reverse Engineering*, British Columbia, Canada, November, 2003, pp.323-332.

[23] O. Tal, S. Knight, and T. R. Dean, "Syntax-based Vulnerability Testing of Frame-based Network Protocols", *Proc. 2nd Annual Conference on Privacy, Security and Trust*, Fredericton, Canada, October 2004, pp.155-160.

[24] A. Lakhotia, and M. Mohhammed, "Imposing Order on Program Statements and its implication to AV Scanners", *The 11th Working Conference on Reverse Engineering*, Delft, Netherlands, November, 2004, pp.161-170.

[25] A. Lakhotia, and P. Pathak, "Virus Analysis: Techniques, Tools, and Research Issues Tutorial", *The 11th Working Conference on Reverse Engineering*, Delft, Netherlands, November, 2004, pp.2.

[26] P. Thiran, G.-J. Houben, J.-L. Hainaut, and D. Benslimane, "Updating Legacy Databases Through Wrappers: Data Consistency Management", *The 11th Working Conference on Reverse Engineering*, Delft, Netherlands, November, 2004, pp.58-67.

[27] M. Marin, A. van Deursen, and L. Moonen, "Identifying Aspects Using Fan-in Analysis", *The 11th Working Conference on Reverse Engineering*, Delft, Netherlands, November, 2004, pp.132- 141.

[28] http://lists.virus.org/dailydave-0311/msg00020.html.

[29] D. Beyer, T. H. Henzinger, R. Jhala, and R. Majumdar, "Checking Memory Safety with Blast", *Proc Fundamental Approaches to Software Engineering*, LNCS 3442, Edinburgh, Scotland, April, 2005, pp. 2-18.

[30] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe Retrofitting of Legacy Code", *Proc Principles of Program. Lang.*, Portland, January, 2002, pp. 49-61.