Appears in the 2003 USENIX Security Symposium, August 2003.

High Coverage Detection of Input-Related Security Faults

Eric Larson and Todd Austin

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48105
larsone@eecs.umich.edu, austin@eecs.umich.edu

Abstract

Improperly bounded program inputs present a major class of program defects. In secure applications, these bugs can be exploited by malicious users, allowing them to overwrite buffers and execute harmful code. In this paper, we present a high coverage dynamic technique for detecting software faults caused by improperly bounded program inputs. Our approach is novel in that it retains the advantages of dynamic bug detection, scope and precision; while at the same time, relaxing the requirement that the user specify the input that exposes the bug. To implement our approach, inputs are shadowed by additional state that characterize the allowed bounds of input-derived variables. Program operations and decision points may alter the shadowed state associated with input variables. Potentially hazardous program sites, such as an array references and string functions, are checked against the entire range of values that the user might specify. The approach found several bugs including two high-risk security bugs in a recent version of OpenSSH.

1. Introduction

Bugs in software can have a devastating effect in today's world. Computer viruses can exploit software bugs in order to run malicious code or gain access to restricted data. A highly visible and damaging example of this type of defect includes improperly bounded checks on network data. A common example of this class of defect is the buffer overflow. Input data is obtained from the network without checking to see if it will fit within a program buffer. Malicious users can exploit this bug by overwriting stack buffers in a way that overwrites the function return address to direct control to arbitrary code.

To prevent a buffer overflow exploit, it is necessary for the program to check input data to ensure it does not exceed the bounds of any buffer it may be used to reference. However, many programs either fail to check input data or check the data incorrectly. Such cases are often hard to find. For example, the code sequence in Figure 1 contains an off-by-one error. Such an error may be difficult to find if the programmer writing the code to check the reference is not aware that the index is incremented before it is referenced.

```
  unsigned int x;
  int array[5];
  scanf("%d", &x);
  if (x > 4) fatal("Index out of bounds");
5 x++;
6 a = array[x];
```

Figure 1: Example of an array bounds error. This code segment will overflow the array if `x` is 4.

Another common source of security bugs is improper use of the string library functions in C. Since the functions provide no checking, the responsibility resides with the programmer. To complicate matters, there is little consistency on how the different string functions operate. For instance, the `strcpy` command always copies a null character but `strncpy` will not copy the null character unless one is present within the specified limit. An example of a bug involving strings is shown in Figure 2. In this case, there is a check to filter out strings that are greater than 16 characters. However, the `strlen` command does not count the null character. If the source string is exactly 16 characters (not including the null character), it will pass the check though it contains 17 characters, including the null character. As a result, the null character does not get copied by the `strncpy` command, creating a potentially dangerous `strcpy` because the source is not null terminated. This type of problem is difficult to catch during testing since it requires a source string of exactly 16 characters. Also, the bug may not manifest in an error in the output when such an input is presented; this is likely the case if the character after the `temp` array happens to be a null.

```
   char *bad_string_copy(char *src)
   {
     char *dest;
     char temp[16];
5
     if (strlen(src) > 16) return NULL;
     strncpy(temp, src, 16);
     dest = (char *) malloc(16);
     strcpy(dest, temp);
10   return dest;
11 }
```

Figure 2: Fault due to improper use of string library functions. If `src` has 16 characters (not including the null character), it will get copied into `dest` without a null character causing a problem in the subsequent `strcpy` function.

Code Segment	Value of x	Interval constraint on x
`unsigned int x;`		
`int array[5];`		
`scanf("%d", &x);`	2	$0 \le x \le \infty$
`if (x > 4) fatal("Index out of bounds");`	2	$0 \le x \le \infty$
`x++;`	2	$0 \le x \le 4$
`a = array[x];`	3	$1 \le x \le 5$ $\longrightarrow$ ERROR!

Figure 3: Detecting an array bounds error. The error is detected even though the input value of 2 does not directly cause an error.

One approach to prevent security exploits is to add run-time support that will prevent malicious behavior. For example, a technique created by Lhee and Chapin [20] append type information to arrays and intercepts string functions to ensure the bounds of the array are not exceeded. The added run-time support remains with the program after it has been deployed resulting in a performance penalty. To avoid this penalty, software teams will sometimes employ testing efforts to detect errors at design-time, thereby reducing the need for run-time checking once the software has been deployed. Often, dynamic bug detection tools are used to aid validators by finding bugs that do not necessarily manifest in an error in the program output. These tools use run-time knowledge of all variables including pointers and variables that reside on the heap. Dynamic techniques can find defects over a wide scope of the program including bugs that span multiple function boundaries, library functions, or even process boundaries. However, they are limited to exposing only those defects that testers can expose. For example, the bug in Figure 1 would only be detected when the input value of four is provided. Without extensive testing, dynamic techniques are best used to expose defects in common-use scenarios.

In this work, we introduce a dynamic high coverage approach to detecting security faults caused by improperly bounded inputs. Our technique possesses the scope and program knowledge of a run-time technique while relaxing the requirement that the validator specify a set of inputs that exposes the defect. We implement our approach by shadowing all input values (and variables derived from input) with a state variable. State variables are introduced into the program when external inputs are read. External inputs encompass a variety of sources: command line arguments, input files, environment variables, and network data.

Integers are shadowed by an *interval constraint variable* that stores the lower and upper bounds of the range of values that the given variable may hold. They have an initial value indicating the input is unbounded with maximum range that can be represented by the data type of the variable. During execution, control tests and operators may narrow input interval constraints. Finally, at potentially dangerous uses of inputs, such as array references and trusted system calls, the entire range of an input value is validated using the computed interval constraint. As a result, all input-related faults are exposed for a given control path, even if the user-specified input did not directly expose the fault.

In Figure 3, we show how our technique can find the off-by-one error in the code segment from the example in Figure 1. In a conventional dynamic bug detection implementation, an error will not be detected unless `x` is four. In our correctness model, we improve defect coverage by extending input values with interval constraints. When the value is first read from input, it is given a range to span all possible values. At the control points (after the `if` statement in the example), the interval constraint can be narrowed because the value of `x` is now known to be ≤ 4. When the value of `x` is incremented, the interval is adjusted up by one on both ends. When the array access occurs, the interval of `x` is compared to the size of the array. Even though `x` has the legal value of two, an error is flagged since it is possible for the input to be five, which exceeds the bounds of the array.

Input strings are shadowed by state variables that hold the maximum possible size of the string and a flag that indicates if the string is known to contain a null character. Like integers, strings from external input sources are considered to have an unbounded maximum size. Control predicates that test the length of an input string can decrease the maximum string length. The null flag is initially set on input strings and is initially clear on uninitialized arrays. In order to set the null flag, a string must be copied in a manner that guarantees that null is set. String functions are checked to ensure that all strings passed as a parameter are null terminated and there is sufficient room in the destination string for copy operations. Our approach will verify these functions for all possible string lengths up to the maximum size making it unnecessary for a test to have the exact string length that can trigger an error.

The string example is revisited in Figure 4. Assume that the input to the function is a null terminated string consisting of 8 characters taken directly from input. Upon entry to the function, its maximum size will be unbounded. Though the user entered an 8 character string, it could have an entered a string of any length. Since the string has 8 characters it passes the check but its maximum size is reduced to 17. (We count the null character in our definition of string size.) In the `strncpy` function, the maximum size of 17 can exceed the available destination size limit of 16. Consequently, there is no guarantee that the null character is copied, and the null flag remains off for the array `temp`. This leads to an error when `temp` is used in the `strcpy` function

Code Segment	State for src	State for temp	State for dest
`char *bad_string_copy(char *src)`	max_sz: ∞, known_null: T		
`{`			
`  char *dest;`			
`  char temp[16];`		max_sz: 16, known_null: F	
`  if (strlen(src) > 16) return NULL;`	max_sz: 17, known_null: T		
`  strncpy(temp, src, 16);`		max_sz: 16, known_null: F	
`  dest = (char *) malloc(16);`			max_sz: 16, known_null: F
`  strcpy(dest, temp);`			ERROR (temp may not be null terminated)
`  return dest;`			
`}`			

Figure 4: Detecting a string copy error. The error is detected because it is possible for the `strcpy` function to execute with a source that is not null terminated.

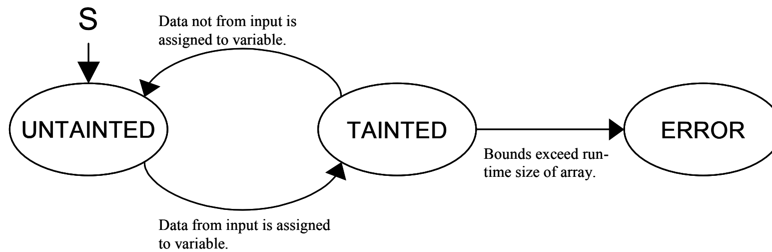

Figure 5: Program states for integer variables.

since it may not be null terminated. Even though the input string of 8 characters does not expose the error, our approach still detects it because all possible string lengths are verified. It is also worth noting that our approach would detect an error if the string `src` were directly copied into `dest` in the `strcpy` instead of using the `temp` array. In this case, the error would get signalled because the maximum size of the source (17 characters) is greater than the destination (16 characters).

Our approach is generic and can be applied to all programs. We have applied it to several programs and found a number of bugs including two major security bugs in a recent release of OpenSSH. It is portable and does not require any modifications to the source code. Unlike techniques that are designed to prevent malicious behavior while running, our technique is intended to be used to find faults before software is released. Given the degree of our analysis, the run-time performance impact of our technique is fairly high, making our approach only appropriate for the testing phase of development. Another limitation of our approach is that defect detection is control path sensitive. Good testing, however, can mitigate this effect by covering all of the interesting paths of a program.

The remainder of the paper is organized as follows. The next section describes our method for detecting input-related faults. Section 3 describes our methodology and our dynamic bug detection tool. Section 4 shows the types of bugs we have found using our approach and compares the run-time performance to an uninstrumented version of the program. Section 5 outlines related work, and Section 6 gives conclusions.

2. High coverage detection of software faults

Our high coverage scheme for detecting software faults centers on verifying all possible input values at dangerous points. At array references, the entire range of possible index values are checked to ensure the array bounds are not exceeded. This procedure is described in Section 2.1. Unsafe string functions such as `strncpy` are checked by assuming that input strings can have arbitrary length initially. String operations are checked to ensure there is sufficient room to store the largest possible string. Section 2.2 details our technique. Section 2.3 lists other situations we detect that lead to software faults.

2.1 Detecting dangerous array references

In order for an array reference to be considered safe, the index must be checked to determine if it is possible to exceed the bounds of the array. This is accomplished by attaching interval constraint information be to every variable that contains program input. To keep the maintenance of bounds information manageable, we identify at run-time which variables contain program input and only attach interval constraints to these variables. We start with a model of *tainted* data that is similar to that used in [1] and is summarized in Figure 5. Data that comes from unbounded input is considered tainted. Examples of unbounded input include environment variables, command line inputs, data read from files, and network packets. Tainted data is shadowed

Table 1: Sample integer rules and their effect on the bounds. `x′`, `y′`, and `a′` are tainted and `y` is untainted.

Rule	Operation	Input Interval Constraint
1	`a′ = x′`	`a′.lb = max(MIN_VAL(a′), x′.lb)` `a′.ub = min(MAX_VAL(a′), x′.ub)`
2	`a′ = x′ + y`	`a′.lb = max(MIN_VAL(a′), x′.lb + y)` `a′.ub = min(MAX_VAL(a′), x′.ub + y)`
3	`a′ = x′ + y′`	`a′.lb = max(MIN_VAL(a′), x′.lb + y′.lb)` `a′.ub = min(MAX_VAL(a′), x′.ub + y′.ub)`
4	`a′ = x′ % y′`	`a′.lb = 0, a′.ub = max(abs(y′.lb), abs(y′.ub))`
5	`if (x′ < y)`	`if (x′ < y):  x′.lb = x′.lb,         x′.ub = min(x′.ub, y - 1)` `else:         x′.lb = max(x′.lb, y), x′.ub = x′.ub`

6	`if (x′ < y′)`	`if (x′ < y′): x′.lb = x′.lb, x′.ub = min(x′.ub, y′.ub - 1)`
		`               y′.lb = max(y′.lb, x′.lb + 1), y′.ub = y′.ub`
		`else:          x′.lb = max(x′.lb, y′.lb), x′.ub = x′.ub`
		`               y′.lb = y′.lb, y′.ub = min(y′.ub, x′.ub)`

7	`if (x′ == y)`	`if (x′ == y):          x′.lb = y,     x′.ub = y`
		`else if (x′.lb == y): x′.lb = y + 1, x′.ub = x′.ub`
		`else if (x′.ub == y): x′.lb = x′.lb, x′.ub = y - 1`
		`else:                  x′.lb = x′.lb, x′.ub = x′.ub`

8	`if (x′ == y′)`	`if (x′ == y′):  x′.lb = y′.lb = max(x′.lb, y′.lb)`
		`                 x′.ub = y′.ub = min(x′.ub, x′.lb)`
		`else:            x′.lb = x′.lb, x′.ub = x′.ub`
		`                 y′.lb = y′.lb, y′.ub = y′.ub`

9	`while (x′ < y)`	`in loop:      x′.lb = x′.lb,         x.ub = min(x′.ub, y - 1)` `after loop:   x′.lb = max(x′.lb, y), x.ub = x′.ub`

with interval constraint variables that track the lower and upper bounds for the variable. When an access to an array occurs, the bounds of the array index are compared with the run-time size of the referenced array. An error is declared if there is an index that can exceed the bounds of the array. When a variable is assigned a value that is not dependent on input (untainted), the destination variable is reset to the untainted state. Since only tainted variables need interval constraints, any variable transition from tainted to untainted will release shadow state.

Since arrays may only be indexed by an integer, only variables with integer type (`char, int, unsigned int,` etc.) can become tainted. When an integer variable becomes tainted, it is assigned upper and lower bounds based on the precision of the type. For unsigned integers, the lower bound is zero. Otherwise, it is the most negative value the variable can hold, based on type. Similarly, the upper bound is the largest possible value.

As tainted variables are operated upon or tested with control predicates, their interval constraints must be adjusted accordingly. Table 1 shows a list of representative operations and their effect on the upper and lower bounds of an interval constraint. In the table, ticked variables `a′, x′`, and `y′` refer to tainted variables while `y` represents an untainted variable. The notation `x′.lb` represents the lower bounds of tainted variable `x′`. The expressions `MIN_VAL(a)` and `MAX_VAL(a)` refer to the minimum and maximum values that `a` can have based on its type precision.

For simple assignment operations (Rule 1), the bounds are copied into the assigned value in most cases. However, it may be necessary to restrict the bounds if the size of the destination type is smaller than the size of the source type. This may also occur when assigning a signed value into an unsigned value. State propagations are also required when integers are passed at function calls, returned from functions, assigned within structures, or when copied by system functions such as `memcpy`.

Addition and other arithmetic operations adjust the bounds of the destination variable. In the first addition pattern (Rule 2), a tainted value is added to an untainted value. The bounds of the destination variable are computed by adding the run-time value of the untainted variable to the bounds of the tainted variable. If both variables are tainted (Rule 3), the bounds are added together to form the new worst-case bounds. Rule 4 singles out the modulus operator because the new range is strictly dependent on the value of the second operand. We also detect overflow situations; this is mentioned in more detail in Section 2.3.

Rules 5-9 narrow the input interval constraint based on knowledge gained from control predicates. In Rule 5, if the `if` condition is true, the upper bound is reduced to `y-1` unless the existing upper bound is already lower than `y-1`. If the condition is false, the lower bound must be at least `y`. Rule 6 refers to a situation where two tainted variables are compared to one another. If `x' < y'` is true, then no change is necessary to the lower bound of `x'` and the upper bound of `y'`. The upper bound of `x'` must be at least one less than the upper bound of `y'` in order to make the equality true. `x'` also cannot exceed its own upper bound. Similarly, the lower bound of `y'` is the maximum of the lower bound of `x'+1` and the lower bound of `y'` before the statement.

The equality test to an untainted variable (Rule 7) will set both bounds to `y` if the tainted value is indeed equal to the value. If the tainted value `x'` is not equal to `y`, there is no change unless `y` happens to equal one of the bounds. In this case, the bound is adjusted accordingly. While in this case it would be possible to split the interval, this is not necessary as only the lower and upper bounds are needed to validate array accesses. In Rule 8, two tainted variables are compared for equality. If they are equal, each variable will have an identical new range that is formed and by taking the highest lower bound and lowest upper bound. If they are not equal, no change is made[1].

The effect of a `while` loop comparison is shown in Rule 9. When the body of the loop is entered, the condition `x' < y` is true and the bounds are updated appropriately. Upon exiting the loop, the bounds are updated to reflect that the condition is now false. `for` loops and `do` loops are handled in the same manner except that the bounds are not updated during the first pass in a `do` loop since the condition is not tested until the end of the loop. The case where two tainted values are compared as a loop condition is analogous to the `if` statement.

Notable omissions from the list are the logical-or (`||`) and logical-and (`&&`) operators. A simplification phase, discussed in Section 3, converts these short-circuited operators into the appropriate if-then-else constructs.

To perform interval bounds checks, it is necessary to keep track of the sizes of all arrays in the program. The size of globally and locally declared arrays are known at compile-time and are straightforward to process. Dynamic variables pose an interesting challenge in our target language C. Since all dynamic memory allocations are considered to be untyped, we consider all dynamic allocations to be a single array with a size equal to that of the memory allocation.

1. There are some cases that we ignore where the bounds could be adjusted. One example would be when the equality is false and `x'.lb == x.'ub == y'.ub`. In this case, `y'.ub` should be lowered by one. Such cases can easily be added if they result in false alarms.

2.1.1 Array reference example

To illustrate our approach, we will describe a bug that was discovered in OpenSSH. It occurred in the channel code (*channel.c*). The relevant code is shown in Figure 6. In function `channels_new`, the channels array is a dynamically growing array with a size equal to `channels_alloc`. The starting size of the array is 10.

Some time after `channel_new` is called, the function `channel_input_data` is invoked. At line 36, an integer is obtained from a packet using `packet_get_int` (an OpenSSH function that grabs the next integer from the current network packet). Upon return of `packet_get_int`, `id` will be tainted with a lower bound that is equal to $-\infty$ and an upper bound equal to ∞. The value of `id` is passed into the function `channel_lookup`, so the parameter `id` upon entry will have the same bounds.

At line 46, there is a check to make sure that index is within the bounds of the array. If the run-time value of `id` is out of the range 0 ≤ `id` ≤ `channels_alloc`, the error would not be detected since the function call returns before the array access. At the array access in line 50, the interval constraint of `id` has a lower bound of zero and an upper bound equal to the run-time value of `channels_alloc`. The `channels` array has a run-time size equal to `channels_alloc`, indexed from zero to `channels_alloc-1`. Since the upper bound of `id` is `channels_alloc`, it exceeds the bounds of the array and an error is declared. If `id` is in the range 0 ≤ `id` ≤ `channels_alloc`, the error will be detected despite the fact that an error only occurs when `id` equal to `channels_alloc`. For any value of `id` that executes the array access, our technique will detect the error. To fix this bug, line 46 must be changed to use '>='.

2.2 Detecting misuse of string functions

String functions such as `strcpy` can lead to software faults since no check is performed to determine if the source string will fit in the destination buffer. In order to check these functions, strings that come from user input are tracked. Since a string provided by a user can have arbitrary length, our technique assumes that all input strings can have infinite length initially. Comparisons made to the length of the string adjust the maximum length of the string. When a string copying function is called, the maximum length of the string is checked to ensure it will fit in the destination.

Another common problem is manipulating strings that are not terminated with a null character. While input strings automatically may contain a null character when they are first created, they could be copied using functions such as `strncpy` which do not copy the null character in all cases. Another common mistake is to forget that the `strlen` command does not include the null character in its count, lead-

```
   /* Pointer to an array containing all allocated channels. The array is
    * dynamically extended as needed. */
   static Channel **channels = NULL;

5  /* Size of the channel array. */
   static int channels_alloc = 0;

   Channel *
   channel_new(...)
10 {
     int i, found;
     Channel *c;

     /* Do initial allocation if this is the first call. */
15   if (channels_alloc == 0) {
       channels_alloc = 10;
       channels = malloc(channels_alloc * sizeof(Channel *));
       ...
     }
20   ...
     if (found == -1) {
       channels_alloc += 10;
       channels = realloc(channels, channels_alloc * sizeof(Channel *));
       ...
25   }
     ...
   }

   void
30 channel_input_data(int type, int plen, void *ctxt)
   {
     int id;
     Channel *c;

35   /* Get the channel number and verify it. */
     id = packet_get_int();                                   -∞ ≤ id ≤ ∞
     c = channel_lookup(id);                                  -∞ ≤ id ≤ ∞
     ...
   }
40
   Channel *
   channel_lookup(int id)                                     -∞ ≤ id ≤ ∞
   {
     Channel *c;
45
     if (id < 0 || id > channels_alloc) {                     -∞ ≤ id ≤ ∞
       log("channel_lookup: %d: bad id", id);
       return NULL;
     }
50   c = channels[id];                                        0 ≤ id ≤ channels_alloc
     return c;
52 }
```

Figure 6: OpenSSH channel bug. The `channels` array has a size of `channels_alloc`. The bug occurs in `channel_lookup` where it is possible to access `channels[channels_alloc]` which is outside the bounds of the array.

ing to an off-by-one error if used incorrectly. Since we consider input strings to have an arbitrary length, it is not necessary for a user to supply a string of the precise length in order to find such errors.

All strings and arrays in the program are tracked with the three fields: `actual_size`, `max_str_size`, and `known_null`. The field `actual_size` stores the actual run-time size of the array and cannot change (except for calls to `realloc`). The field `max_str_size` stores the maximum size of the string in the array. It refers to the largest possible size of a string that a user can supply. For example, strings that come from the command line have an initial `max_str_size` of infinity (`INT_MAX`) and strings that are created using `fgets` have a `max_str_size` equal to the supplied limit. For arrays that do not contain strings, `max_str_size` is equal to `actual_size`. The `known_null` field is a flag that is true if the string is known

Table 2: Representative Rules for String Buffer Overflow Checking. `s` is an array, `p` is a pointer, `n` and `c` are integers, and `m''` and `n''` are integers that store a string length.

Array Creation		
1	`s = argv[i]`	`s.actual_size = strlen(s)+1; s.max_str_size = INT_MAX; s.known_null = TRUE;`
2	`s: string constant`	`s.actual_size = s.max_str_size = strlen(s)+1; s.known_null = TRUE;`
3	`char s[n]`	`s.actual_size = s.max_str_size = n; s.known_null = FALSE;`
4	`s = malloc(n)`	`s.actual_size = s.max_str_size = n; s.known_null = FALSE;`

5	`s = malloc(n'')`	`s.actual_size = n'';`
		`s.max_str_size = (n''.string).max_str_size + n''.size_diff;`
		`s.known_null = FALSE;`

String Length Manipulation		
6	`n'' = strlen(s)`	`Assert: s.known_null == TRUE` `n''.string = s; n''.size_diff = -1;`
7	`n'' = m'' + 1`	`n''.string = m''.string; n''.size_diff = m''.size_diff + 1;`

8	`if (n'' <= c)`	`if (n'' <= c) (n''.string).max_str_size =`
		`MIN((n''.string).max_str_size, c + n''.size_diff);`
		NOTE: no change is necessary if `(n'' > c)`

Basic Array Operations		
9	`s[n] = c`	`if (c == 0) s.known_null = TRUE;`
10	`*p = c`	`if (c == 0) (p.array_base).known_null = TRUE;`

to contain a null. If it is false, it is not known if a null character is present or not present. During checking, we assume that the string is not null terminated if `known_null` is false. We will represent accesses to these fields using structure notation: `s.max_str_size` refers to the field `max_str_size` associated with the array `s`.

Strings can be created in a variety of ways as shown in the first five rules of Table 2. Strings that come from the command line or environment variables (Rule 1 in Table 2) will be marked as having an infinite maximum string size since the user could have supplied a string of any length[1]. Since these strings are automatically null-terminated, the `known_null` field is set to true. String constants (Rule 2) are not dependent on the user and thus have a maximum string size equal to its actual size. Rules 3-5 assume the arrays are uninitialized, making the initial value of the `known_null` flag false. Initializers can be viewed as assignments after the array has been created. Locally or globally declared arrays (Rule 3), do not store strings when they are first created and have a maximum string size equal to its actual size. In most cases, dynamically allocated arrays are processed identically to the creation of arrays declared at compile-time (Rule 4). One exception is when the size of the allocation is dependent on the size of another string. This case (Rule 5) is described in the next paragraph.

1. We conservatively use infinite size even though the operating system imposes a limit on the length of a command line.

Similar rules exist for `calloc`, except that `known_null` is initialized to true.

In order to properly adjust the maximum size of a string, it is necessary to track integers that store string lengths. Any integer that is storing a string length will have state that stores the starting address of the corresponding string (denoted using the field `string`). In addition, a `size_diff` field is also stored that is the difference between the value stored in the integer and the actual length of the string. This is important as our approach includes the null character in the length of a string while the `strlen` command does not. Therefore, the initial size difference for a `strlen` result is -1. In Table 2, variables that store string lengths are represented with two tick marks (such as `n''`). Rule 6 shows the `strlen` call. Since `strlen` requires the input string to be null terminated, a check is made to make sure that is the case. Addition and subtraction operations (Rule 7) on the string length adjust the size difference appropriately. For example, adding one to a `strlen` result to account for the null character will result in a size difference of zero. The maximum length of the string can be reduced with a control operation; this is illustrated in Rule 8. If (`n'' <= c`) is true, the maximum size of the string s is adjusted to `c + n''.size_diff` unless the maximum size is already smaller. If (`n'' <= c`) is false, no adjustment is made to `s` since there is no restriction on the maximum size of `s`. Refer back to Rule 5 where a string length is used as the size parameter to a dynamically allocated array. This is

Table 3: Representative Rules for String Buffer Overflow Checking (continued). `s`, `d`, `set`, `fmt` are strings, and `p` is a pointer to a string. `n` is an integer and refers to a parameter that restricts the number of a characters written into a destination buffer. The macro `SIZE(s)` is equal to `MAX(s.actual_size, s.max_str_size)`.

String Functions		
11	strcpy(d,s)	Assert: s.known_null == TRUE Assert: s.max_str_size <= SIZE(d) d.max_str_size = s.max_str_size; d.known_null = TRUE;

12	strncpy(d,s,n)	Assert: s.known_null == TRUE
		Assert: (n <= SIZE(d))
		d.max_str_size = MIN(s.max_str_size, n);
		d.known_null = (s.max_str_size <= n);

13	strcat(d,s)	Assert: s.known_null == TRUE && d.known_null == TRUE
		Assert: s.max_str_size <= SIZE(d) - strlen(d)
		d.max_str_size = s.max_str_size + strlen(d);
		d.known_null = TRUE;

14	strncat(d,s,n)	Assert: s.known_null == TRUE && d.known_null == TRUE
		temp_src_size = MIN(n + 1, s.max_str_size)
		Assert: temp_src_size <= SIZE(d) - strlen(d)
		d.max_str_size = temp_str_size + strlen(d);
		d.known_null = TRUE;

15	strchr(p,s) also: strrchr	Assert: s.known_null == TRUE if (p) p.array_base = s;
16	strstr(p,s,set) also: strpbrk, strppbrk, strtok, strsep	Assert: s.known_null == TRUE && set.known_null = TRUE if (p) p.array_base = s;

17	d = strdup(s)	Assert: s.known_null == TRUE
		d.actual_size = d.max_str_size = s.max_str_size;
		d.known_null = TRUE;

18	fgets(d, n, stream)	Assert: n <= SIZE(d) d.max_str_size = n; d.known_null = TRUE;
19	gets(d)	Automatic error! d.known_null = TRUE;

20	scanf(fmt, d)
	Also: fscanf, sscanf
		Get width from fmt (width = 0 if no width was given)
		Assert: width != 0 && width <= SIZE(d)
		if (width != 0) d.max_str_size = width;
		d.known_null = TRUE;

21	sprintf(d, fmt, s)	Assert: s.known_null == TRUE Check to make sure the sum of all source strings does not exceed SIZE(d), non strings are ignored in this calculation d.known_null = TRUE;
22	snprintf(d, n, fmt, s)	Assert: s.known_null == TRUE Assert: n <= SIZE(d) If the sum of all source strings exceeds SIZE(d), then d.known_null = FALSE; otherwise d.known_null = TRUE; d.max_str_size = n;
23	strcmp, strpos, strrpos, strspn, strcspn, atof, atoi, atol, str-tod, strtol, strtoul, strcoll	Check that all input source strings are null terminated.

commonly done before a string copy to ensure the destination as enough space to hold the source string. As a result, the field `max_str_size` of the newly allocated region is initialized to properly reflect the maximum size of `n''.string` rather than using `n''`. Our current implementation limits the usage of string length integers. We do not handle arithmetic operations except addition and subtraction and we do not handle operations that involve two string length integers. For more information on these restrictions and the limitations they cause, see Section 3.1.

Assigning zero to an element of an array will set the known_null flag to true (Rule 9). An assignment of a value other than zero has no effect on the state of the array. The known_null flag is also set to true when functions bzero and memset (to zero) are called. String arrays are often accessed via pointers. This case is handled by shadowing the pointer with the base address of the array (denoted using array_base). Pointers that do not point to arrays are not shadowed. In the event, the string is addresses using an interior pointer, the state of the array is obtained using the shadowed base address. This is illustrated in Rule 10. For brevity, we will omit this level of indirection and assume arrays are used in all future rules.

Table 3 shows how string library functions are handled. Rules 11 and 12 illustrate how string copies are handled. In strcpy, the source must be null terminated and the size of the maximum string size must fit in the destination. The size of the destination is determined by taking the maximum of actual_size and max_str_size. In the case where max_str_size is larger than actual_size, max_str_size is chosen since the only way this situation can occur is when the array was dynamically allocated with a string length. Our approach naively assumes that the string length used referred to the length of the source string. This assumption could lead to undetected bugs and is discussed in more detail in Section 3.1. For brevity, we use SIZE(s) = MAX(s.actual_size, s.max_str_size) to represent the size of the destination buffers. If both the null check and size check pass, the destination will then have known_null set to true and a max_str_size equal to that of the source. In strncpy, a check ensures that the destination size is less than the supplied size (n) parameter. This is done regardless of the size of the source string since nulls are padded at the end if the source is smaller. The destination will have a max_str_size that is the smaller of n and max_str_size of the source. The known_null is only true if the entire source string is copied.

The strcat functions (Rules 13 and 14) are handled similarly to strcpy. The key differences are the destination must be null terminated and the run-time size of the destination string is subtracted from the size comparisons. Unlike strncpy, strncat will always add a null character and as a result could copy n + 1 characters. The substring extraction functions (Rules 15 and 16) check for null terminated input strings. The destination, if not NULL, is a pointer to somewhere in the original source string. As a result, the pointer gets shadowed with the address of the source string. This has the effect of assuming that the substring is identical to the original string, which in the worse case, could be true. Token extraction routines such as strtok are handled the same way, each token is the assumes to be the same size of the original source string. The strdup function (Rule 17) is straightforward. A new string is created with the exact same characteristics as the source string.

Strings that come from input functions such as fgets (Rule 18) will have a maximum size equal to the size that was supplied as the limit to the function and are null terminated. These input functions are checked to ensure that input will fit in the supplied buffer. As a result, the function gets (Rule 19), which has no checking, automatically flags an error each time it is used. The scanf family of functions (Rule 20) are also unsafe unless a field width is supplied for each input string. If a field width is present, it will checked to ensure the input string fits in the destination buffer. The function sprintf (Rule 21) is implemented to ensure that the sum of the maximum sizes of all source strings does not exceed the destination size. In snprintf (Rule 22), the size is limited by the size parameter but null is not written if the source strings can exceed the destination. String functions that only read strings only check to see if all input strings are properly null terminated (Rule 23).

A programmer may mimic the behavior of a strcpy by using a pointer to walk the elements of an array and copying each element individually. Copying via indirect references (*d = *s, for example) does not alter state unless the last element of the array s is copied into d. In this case, we assume that the entire array is copied from s into d and the statement is treated like a string copy. While this is not the case in all situations, current tool limitations prohibit more sophisticated analysis. This is addressed as future work.

2.2.1 String example

A detailed example illustrating how string errors can be found is shown in Figure 7. The two buffers buf0 and buf2 have an initial maximum size equal to their static sizes. In line 6, the input value argv[1] is copied into buf0 using strncpy. While the specified size of 12 does not cause an overflow, the null flag for buf0 remains off since a null would not have been copied if argv[1] has at least 12 characters. The strdup in line 7 will duplicate the state values of argv[2] into buf1. If value is true, execution will continue to line 10 where the pointer p is assigned to point to the second element of buf0. This causes p to be shadowed with the base address of buf0. When p is used in the strcpy, an error gets properly signalled because buf0 may not be null terminated. In the case where value is false, a comparison is made based on the length of buf1. Assuming it is less than or equal to 6, control will be taken to line 14 and the maximum size of buf1 will be restricted to 7 (6 plus 1 for the null character that strlen does not count). The known_null flag is set for buf0 in line 14. In line 15, an error results because the sum of the maximum sizes of the two source buffers (19) can exceed the size of the destination (18).

2.3 Other improper uses of input data

Our approach can also be used detect other situations where

```
   char buf0[12];                         buf0.max_str_size = 12, buf0.known_null = FALSE
   char *buf1;
   char buf2[18];                         buf2.max_str_size = 18, buf2.known_null = FALSE
   char *p;
5
   strncpy(buf0, argv[1], 12);            buf0.max_str_size = 12, buf0.known_null = FALSE
   buf1 = strdup(argv[2]);                buf1.max_str_size = ∞, buf1.known_null = TRUE

   if (value) {
10   p = buf0 + 1;                        p.array_base = buf0
     strcpy(buf2, p);                     p.array_base is buf0 → buf0.known_null == FALSE → ERROR
   }
   else if (strlen(buf1) <= 6){           buf1.max_str_size = 7, buf1.known_null = TRUE
     buf0[12] = 0;                        buf0.max_str_size = 12, buf0.known_null = TRUE
15   sprintf(buf2, "%s%s", buf0, buf1);   (buf0.max_str_size + buf1.max_str_size = 19) > (buf2.max_str_size = 18) → ERROR
16 }
```

Figure 7: Example of detecting string bugs. The `strcpy` in line 11 can fail because `buf0` is not null terminated. The `sprintf` in line 15 can fail because the sizes of the two source strings could exceed the size of the destination.

```
    unsigned int nresp;
    nresp = packet_get_int();                           0 ≤ nresp ≤ ∞
    if (nresp > 0) {
      response = malloc(nresp * sizeof(char*));         1 ≤ nresp ≤ ∞
5     for (i = 0; i < nresp; i++)                       1 ≤ nresp ≤ ∞
        response[i] = packet_get_string(NULL);
7   }
```

Figure 8: OpenSSH challenge bug. Unbounded data from a packet can cause overflow when calling `malloc`.

input data could be used dangerously and possibly lead to software faults. Unconstrained input used to control the number of loop iterations, the size of a memory copy, or the size of a memory allocation could be dangerous [1]. We check to make sure that the variables controlling these uses have been constrained in some fashion. An error is signalled if the upper bound is equal to the maximum value allowed by the type of the variable. An error is also reported if the value could be negative. For these situations, it is important to note that these types of uses are not always errors. In some cases, input is constrained later within the loop and malicious behavior is properly thwarted. In other cases, the loop may not do anything that can be exploited. Memory allocations are likely not dangerous if the output is properly checked to ensure the allocation was successful.

Another related problem is arithmetic overflow. A common example of the case of adding two large signed integers where the destination is not large enough to store the result, leading to a negative number with large magnitude. As with the previous case, this usually occurs when input data is unconstrained. Overflow and underflow is detected using the bounds associated with integers. Once an operation is completed, the resulting upper and lower bounds are analyzed to determine if it can fit into the destination variable. An error is signalled if the resulting value cannot fit.

While this type of problem doesn't necessarily lead to an security exploit, it often does. For example, we describe another security bug found in OpenSSH. The code is listed in Figure 8. Data is received from a packet and then is subsequently used to allocate an array. At the time of the `malloc`, no restriction has been placed on the input. A malicious user could supply an extremely large value in order to cause an overflow on the multiplication in the `malloc` call resulting in a small allocation. Since the same input value controls the number of iterations within the loop that follows, the array accesses within the loop can be used to access memory outside of the array.

Our approach can also be used to find potential bugs when integers are casted. For example, an assignment of a signed long integer into an unsigned short integer can be problematic if the signed long integer has a negative value or a value larger than the maximum size of the unsigned short integer. However, during our testing, we were unable to find any defects due to improper casting. We did find several cases where casting of this sort was done intentionally and correctly causing false alarms. As a result, we disabled casting checking for our experiments in Section 4.

3. Implementation

Our dynamic checker is built on a general purpose source-level instrumentation tool, called *MUSE*, that we designed to facilitate the construction of dynamic defect detection tools. Unlike most dynamic verifiers that focus on one particular property, our system is general purpose. A user, using our checking specification language, can specify program properties they wish to validate.

MUSE is a source-level general purpose program instrumentation tool, allowing users to specify the properties they are interested in checking. The tool is built as a compilation phase in the GNU GCC compiler. Given a hand-written model of program correctness, MUSE will automatically

locate the instrumentation points used by the model, graft in the necessary program instrumentation, and link in any needed additional run-time support. No modifications to the program source code are needed. When the instrumented program is executed, any violations of the correctness properties specified are detected and reported to the user. Program instrumentation is performed at the abstract syntax tree (AST) level, thus source code is required to add instrumentation. Functions without source code, such as system libraries, may be instrumented at their entry and exit points.

The first step of the instrumentation process is to simplify the program. We convert the program into *Elemental C*, an intermediate C representation similar to the simple grammar developed by Hendren *et al.* [15]. The purpose of simplification is to reduce the complexity of identifying and instrumenting relevant program points. Complex C statements are broken down into simple statements with at most two operands and a single assignment to an l-value (such as a = b + c). Side effects and short-circuited operators are eliminated via program transformations.

The MUSE correctness specification consists of program source patterns and associated model actions that are specified as a collection of <pattern, action> tuples. Program source patterns are simple regular expressions, including wildcards, that are matched against elemental C statements. The actions are completely written in C and contain the instrumentation functions that perform state management and error checking. Actions are compiled with the instrumented program to form an instrumented executable. The patterns correspond to complete statements or subexpressions within the elemental C language. In addition, patterns exist that match special program events. Examples of special events include the beginning of a function or use of a variable as an r-value. It is beyond the scope of this paper to present the specification language.

During the instrumentation phase, the patterns are parsed and the program is traversed one elemental C statement at a time. When there is match, the action code is inserted at the matching site. Instrumentation can either be added before or after the matching site depending on how the instrumentation was specified. In addition, various run-time values may be passed to the instrumentation in order to parameterize model actions. After instrumenting the program AST, the remaining compiler phases are executed to produce an instrumented executable. Compiler optimizations may be enabled during this portion of the compilation. Optimizations can recoup some of the performance penalty that is incurred by instrumentation.

The model is implemented using three tables that contain the shadowed state associated with the variables. One table stores state for arrays, another for pointers, and one for integers. All arrays are inserted in the array table when they are created and are indexed by their base address. Each entry in the array table contains four fields: `actual_size`, `max_str_size`, `known_null`, and `is_input`. The first three fields are described in Section 2.2. The `actual_size` field is also used in the array reference checking. The `is_input` field is used to mark arrays that contain program input. If an array from input is used in a function that converts a string to an integer, such as `atoi`, the resulting integer will be treated as input. The pointer table only stores pointers that refer to an array and are indexed by the address of the variable. Each entry contains a single field, the base address of the array that the pointer points to. The integer table also only contains entries for integers that require shadowed state. An integer can require state for three reasons: (a) it contains input data, (b) it contains string length data, or (c) it is a boolean value that will narrow the bounds if used in a conditional expression. A flag is used to distinguish between the three cases. In case (a), there are upper and lower bound fields as described in Section 2.1. In case (b), there are string and size difference fields as in Section 2.2. In case (c), the entry contain an address indicating the variable to be updated, the appropriate bounds (`lb`, `ub`, and `max_str_size`) when the condition is true and bounds for when the condition is false. If the conditional value is used in a control statement, the bounds of each variable is updated using this information.

When an error is detected, the error message will displayed that includes the file name, line number, matching MUSE pattern, and a descriptive message describing the error. The error is detected at the point of the dangerous use such as an array reference or string function. However, the source of the error may not be near the dangerous use. With the help of a debug mode, that prints out a message every time state has changed, it was usually very straightforward to find the source of the error or to classify the bug as a false alarm.

3.1 Limitations

Since our approach is dynamic and relies on the particular control path taken through a program, it is an *unsound* approach, meaning that is possible to miss the detection of actual bugs. With respect to a particular control path, our approach is also unsound. One problem stems from the use of run-time data. An example is detecting when a zero gets written into an array (see Rules 9 and 10 in Table 2). Another case is the actual size of the array is used during array checks. On a different run with the same control path, the size of the array, if controlled by input, could be smaller and be subjected to an array buffer overflow. This is illustrated in Figure 9. The size of the array is controlled by user input and could be any value from 1 to 10. However, the filter of illegal accesses for the index in line 13 is valid when the array size if 10 and invalid for all other accesses. As a result, an error will be missed if 10 is supplied as the array size.

```
   unsigned int size;
   unsigned int index;
   int *array;
   int x;
5
   size = getchar();
   if (size <= 0 || size > 10) exit();
   array = (int *) calloc(size, sizeof(int));

10 /* initialize array */

   index = getchar();
   if (index < 0 || index > 9) exit();
14 y = array[index];
```

Figure 9: Example of an unsound control path. This code segment will overflow the array if x is 4.

Fully addressing these problems would require symbolic analysis. Other shortcomings occur due to a lack of symbolic analysis. While most result in false alarms, a case that results in a missed bug is when the string length of is used to allocate an array and a subsequent `strcpy` operation copies an entirely different string into the destination. This is a bug if the size of the second string is larger than the first. If the `max_str_size` of the first string is high, an error will likely be missed. However, using `actual_size` results in too many false alarms. Lastly, we are also unsound in that we do not attempt to catch every type of buffer overflow that is possible.

Our technique is also *incomplete* in that can produce false alarms, signalled bugs that are not actually bugs. As eluded to earlier, these often occur due to a lack of symbolic analysis. Not all possible relationships between different strings or variables are tracked and this can cause operations that narrow bounds to be missed. A specific example of where symbolic analysis is not present is the limited functionality associated with integers that store string lengths. When an unsupported operation occurs, a warning is emitted, and the result is no shadowed. Another problem arises in that our technique does not keep track of which position the null character is in. In order to maximize the number of detected bugs, we assume it is in the last position. This assumption also leads to an increase in false alarms. In practice, we found the number of false alarms to be manageable. We describe the false alarms triggered in Section 4.1.

4. Results

Our dynamic input analysis checker has been applied to the eight programs listed in Table 4. Programs were compiled using GNU GCC with an -O4 optimization level. Using these programs, we sought to find bugs and measure the effect our instrumentation had on run-time performance.

Three of the programs (*anagram*, *ks*, and *yacr2*) are from the pointer-intensive benchmark suite [26] and were selected due to difficulty of analyzing these programs statically. Each benchmark included several test inputs. All inputs were run for testing purposes. For performance testing, the largest input was selected.

Table 4: Programs used during testing

Program	Description of Program	Defects Found	False Alarms
anagram	anagram generator	2	0
betaftpd	file transfer protocol daemon	1	1
gaim	instant messenger	1	1
ghttpd	web server	3	2
ks	graph partitioning	4	0
openssh	openssh secure shell	3	1
thttpd	web server	0	1
yacr2	yet another channel router	2	1

The other five programs are networking applications. The popular secure shell program *openssh* was tested by targeting the two known bugs that were discovered. Additional testing focused on different modes in both the server and client. Performance testing was done used a scripted session that involved transferring large files and does not attempt to exhaustively execute all of the code. We also tested *gaim*, a popular instant messaging program. Testing was purely interactive and several different instant messaging protocols (MSN, Yahoo, etc.) were used. The one bug found in *gaim* was in the initialization code and was independent of the protocol used. Due to the interactive nature of *gaim*, it was not used in the performance experiment. For the FTP and web servers, testing was done by having the server process several FTP and HTTP requests in different configurations. Performance testing was done by a script that consisted of several requests for a file or web page.

In our testing, we did not strive to exhaustively test all possible code within a program. Using MUSE, the user can add a coverage mechanism similar to one in [14]. The coverage technique may be used to make sure that all potentially dangerous statements (array references, pointer dereferences, and string functions) are executed at least once. Like normal statement coverage, executing all of the dangerous statements once does not guarantee that all bugs will be found since bugs could be dependent on the particular control path that is executed.

4.1 Bugs Detected

With our tool, we were able to find 16 bugs, shown in Table 4. In order to compare our results with static analysis, we used the tool from [29] and analyzed six of the eight programs. Problems processing the source code prohibited analysis of *gaim* and *openssh*. The tool was unable to detect any of the bugs that were discovered using our approach and did not detect any additional bugs.

Two of the three defects found in *openssh* (described in Section 2) are both security flaws present in version 3.0.2. The channel id bug would be difficult to locate via static analysis. The array is dynamically allocated and its size can change during execution. In addition, creation of the array, reading of input, and accessing the channel array each occur in three distinct functions. Any static approach to locating this bug would require interprocedural analysis. The third defect discovered in *openssh* is an addition overflow problem where two numbers read from network data are added together. While this bug does not create a security exploit or a crashing program, it could lead to unexpected program behavior.

In *gaim*, a defect occurs when reading the configuration file. Each field is placed into a large temporary buffer. The fields are processed and copied into the appropriate data structure. In some cases, the fields are copied into a smaller buffer without checking to see if it will properly fit. Examples of fields where this occurs are the username and password. While this bug could not be exploited remotely, it could cause the program to crash. The three defects in *ghttpd* were all due to misuse of string functions. In one case, a `strncat` function contains a limit that does not account for the null character. For a given limit `n`, it is possible for `n + 1` characters to be written since a null character is always written. Another defect was caused by calling strstr on an uninitialized local array. The third defect in *ghttpd* and the defect found in *betaftpd* were the result of using data received from the network without any guarantee that there is a null character.

One bug in *anagram* permits a user to overflow a buffer with characters from an input file. The buffer is dynamically allocated in proportion to the size of the input file. Extra space is added to store additional information about each word in the file. The size of the extra space is controlled by a fixed compile-time constant representing the maximum number of words allowed in the file. If the file contains more words than this constant, the buffer could overflow. The other bug is the result of using `gets`, automatically a dangerous function. In *ks,* two bugs resulted from an input being used to reference an array without any checking to see if it exceeded the array bounds. The other two defects were due to undetected arithmetic overflow with an input value and a loop based on input that is not checked. Both defects discovered in *yacr2* were due to a multiplication overflow. These bugs are very similar to the OpenSSH challenge bug in Figure 8.

Another important factor in bug detection systems is the number of false alarms - situations where an error is signalled when no defect occurs. During the course of our testing, we detected seven false alarms. Three of the seven cases (*betaftpd*, *gaim*, *ghttpd*) were situations where a loop controlled by input did not result in a bug. The other false alarm in *ghttpd* was due to `sprintf` function that was used to concatenate two strings into a new string. The destination buffer had a size equal to the combined sizes of the two strings. This is a case where lack of string length support for the addition of two string lengths leads to a false alarm.

In *openssh*, an arithmetic overflow bug was detected but a consistency check after the operation would correctly signal a failure. In *thttpd*, a false alarm occurs because our approach conservatively assumes that the null character is in the last possible position and does not track the precise location of the null character within an array. A buffer overflow is signalled incorrectly because the program guarantees that a null is in the first position of an array. The false alarm in *yacr2* is due to reading the input file twice. It is read once to set the array sizes and a second time to initialize the array values. Since the array sizes were based on the input, no errors can occur. Our tool currently has no mechanism for determining that the input is actually constrained when it is read the second time.

4.2 Performance

In order to test run-time performance, all of the programs were run on a lightly loaded 1.8 GHz Pentium IV computer running Linux. Program run-times measured using the Unix `time` command were compared to an uninstrumented program running the same test input. The results of the experiment are shown in Table 5. It is important to point out that we have not focused any effort on performance optimization at this point. The results show that there is a large opportunity for improvement possible. The amount of slow-down experienced is dependent on the program. The five server programs exhibited the least amount of slow down with *betaftpd* having the least with 13x. The three pointer intensive benchmarks suffered significant slowdown from a factor of 162x in *anagram* to 220x in *ks*. The disparity in the results can be attributed to the fact that the pointer intensive benchmarks have more integer processing than the servers.

The breakdown of the dynamic instrumentation sites is shown in Table 6. *Array state* sites keep track of array information such as dynamic array sizes and state associated with strings. *Array references* sites include a call to an array reference check function when the index has been controlled by input. *Pointer manipulation* sites are calls to track pointers with their associated array. *Integer state* sites call an associated function to propagate and adjust interval constraints and string lengths. *Control points* are calls that narrow interval constraints or maximum string lengths. *String functions* sites include calls to check the input strings. *Other* includes miscellaneous instrumentation that does not fit the earlier categories. The *Useless* column refers to instrumentation that did not manipulate any of the instrumented states. This includes 1) integer operations and

Table 5: Run-time performance. Performance slow-down is large for computation heavy programs such as *anagram* and *yacr2*. The slow-down is considerably less for server programs *openssh* and *thttpd*.

Program	Simple Stmts	Code Size (Kb)			Run Time (seconds)			Static Sites	Dynamic Sites	Heap Array Refs
		Orig	New	Increase	Orig	New	Increase			
anagram	1,274	11	190	16.8	0.11	17.79	162	920	69,881,404	10.3%
betaftpd	6,325	31	657	21.1	0.08	1.09	13	5,363	3,768,054	69.2%
gaim[1]	374,623	1620	35500	21.9	N/A	N/A	N/A	262,303	N/A	11.0%
ghttpd	3,755	27	424	15.6	0.34	6.70	20	3,971	97,017,000	0.0%
ks	2,066	13	237	18.0	8.75	1923.62	220	2,086	1,889,043,968	0.0%
openssh	155,835	885	9235	10.4	0.02	0.38	19	105,900	421,551	1.8%
thttpd	22,758	138	2301	16.7	0.32	8.47	26	15,530	29,210,392	50.0%
yacr2	7,999	33	786	23.8	0.55	96.79	176	6,454	392,839,706	100.0%

1. Due to the interactive nature of gaim, we were unable to accurately measure its run time performance.

Table 6: Breakdown of dynamic instrumentation calls.

Program	Array State	Array References	Pointer Manipulation	Integer State	Control Points	String Functions	Other	Useless
anagram	2.7%	0.9%	15.0%	3.7%	1.3%	0.0%	2.8%	73.7%
betaftpd	4.4%	0.0%	10.1%	0.0%	0.0%	0.0%	4.3%	81.2%
gaim	2.7%	0.0%	13.2%	1.9%	0.4%	0.4%	4.7%	77.3%
ghttpd	0.8%	0.0%	1.6%	0.0%	0.0%	0.3%	0.5%	96.7%
ks	0.0%	0.7%	34.2%	7.5%	6.3%	0.0%	1.1%	50.1%
openssh	2.1%	0.0%	2.6%	5.7%	3.6%	2.1%	5.1%	78.8%
thttpd	2.4%	0.0%	14.3%	3.1%	1.4%	0.2%	0.8%	77.8%
yacr2	0.7%	7.6%	3.3%	11.3%	1.4%	0.0%	1.2%	75.2%

control points that did not manipulate input data or string lengths, 2) pointer operations that were not associated with an array, and 3) array references that did not have an index that was controlled by input. Clearly, a high percentage of instrumentation calls perform no useful action. Many of these instrumentation sites could likely be eliminated by introducing instrumentation-specific optimizations to the compiler, such as performing copy propagation for interval constraints. Eliminating useless instructions will also reduce the code size overhead. Programs with the fewest number of useless instrumentation exhibited the largest slowdowns. This is to be expected because a useful instrumentation site executes more code than a useless one. The effect on code-size is fairly significant. While the instrumentation functions consume 91KB of space, most of the overhead is due to the added instrumentation calls.

Another interesting statistic is the percentage of array accesses on the heap, shown in the last column of Table 5. Heap array sizes are not generally known at compile-time; thus they are more challenging to analyze statically. The programs *betaftpd*, *thttpd*, and *yacr2* have more heap array references than non-heap array references. In fact, *yacr2* has no non-heap array references. The other programs have significantly fewer non-heap array accesses with *ks* and *ghttpd* having none.

5. Related Work

Several dynamic defect detection tools have been developed. Haugh and Bishop [14] check all of the interesting string library functions by comparing the allocated sizes of the arrays. This approach is similar to our string library maximum size checks. Their tool tracks coverage to ensure that each interesting string function is executed once. Our technique can potentially find more defects because it checks for proper null termination and array references. Examples of memory access checking tools include GNU's checker [6] and Purify [13] which detect memory bugs by keeping track of the state of dynamically allocated memory. Electric Fence [24] places inaccessible pages before and after each dynamically allocated object. A segmentation fault occurs if an access occurs outside the object. CCured [22] uses static program analysis to prove as many pointers to be memory safe as possible. For pointers where this is not possible, run-time checks are inserted into the program. Safe C [2] associates extra state with each pointer that holds the bounds of the object the pointer references. Accesses are compared to the bounds to see if an error occurs. Parasoft's Insure++ [23] checks for a variety of different errors such as memory reference errors, memory leaks, unsafe I/O operations and data conversion errors. The checking is accomplished by adding checking and testing instrumentation around each line of source code. CodeCenter [19] interprets C code and provides run-time type checking and

memory access checking. Fuzz testing [12] found several bugs by injecting a random input stream into Windows and UNIX applications. The work clearly demonstrates that even mature programs pose vulnerabilities to improperly bound inputs.

Several efforts have focused on preventing malicious behavior [8, 20, 21, 25]. The taint mode of Perl [25] can be used to prevent untrusted programs from gaining superuser access. StackGuard [8] is a run-time approach that adds a randomized canary word just below the return address. If the canary word is modified, an error occurs. Buffer overflow attacks that overwrite the return address will also overwrite the canary word negating a jump to the attacker's code. Wahbe *et al.* [30] introduce address sandboxing. An untrusted code module (binary) is instrumented to restrict accesses to that module's segment(s). Related to the issue of security is trust. Proof-carrying code [21] involves embedding a proof into a binary that shows the code satisfies a particular property. An untrusted binary can be verified to see if the proof is valid.

Earlier efforts have used similar bug detection techniques, but the analysis is performed at compile-time using symbolic execution [1, 5, 7, 9, 27, 29]. In these systems, input values are assumed to take on any value and symbolic calculations are used to check if array accesses are within the bounds of the array. Coen-Porisini *et. al.* [7] give a good overview of the approach for a subset of the C programming language and have applied their technique to safety-critical software systems. The advantage of our approach is that our analyses can yield greater precision since they can use run-time information in situations that are difficult to analyze statically. In [1], Ashcraft and Engler constructed models to catch inappropriate uses of tainted data. In their model, tainted data becomes untainted if any check occurs. They only validate that checks are executed. Their approach is unable to determine if the checks are correct. While they have some support for finding errors across functions, their analysis is predominantly local. They have found several bugs in Linux and OpenBSD. PREfix [5] uses a bottom-up approach for model checking. The call graph for the program is created and leaf functions are processed first and replaced with a summary model when called by other functions. Evans and Larochelle [9] and Rugina and Rinard [27] use static analysis to find potential buffer overflows. In [29], arrays or strings are represented as ranges and the problem is transformed into a system of integer range constraints.

Another method for preventing malicious behavior is to enforce safety at the programming language level. Cyclone [18] is a modified form of C that checks pointer accesses using fat pointers. The programmer can limit the number of checks by declaring pointers to be safe. To ensure safety, additional restrictions (for example, disallowing arithmetic) are placed on safe pointers. Shankar *et. al.* [28] introduce a tainted type qualifier to detect vulnerabilities in format strings at compile-time. Data that come from untrusted sources will have a tainted qualified type. When a tainted typed variable is used in potentially dangerous situations, an error is signalled.

A closely related area of study is the elimination of array bounds checks [4, 11]. The approach works by propagating the constraints implied by array bounds checks through the dataflow graph of a program. Similar to our dynamic constraint modifications, program operators and control points adjust propagated constraints accordingly. Array bounds checks that see reaching definitions of earlier (sufficient) checks can be eliminated. These optimizations are directly applicable to the optimization of our input bounds checks, and they form the basis for our on-going static optimization work details in the paper conclusions.

Model checking is another formal method of proving that a program is bug-free. These techniques are very powerful but in order to prove that no errors exist, the number of possible states that must be searched is often extremely large, making the proof infeasible. To reduce the search space, further abstractions are often used that limit the scope of the search. Consequently, the ability to prove that a property is satisfied may be lost, but previous efforts have shown that a large number of bugs can still be found. Static software verification systems include Microsoft's SLAM system [3] which converts a program into a boolean program using predicate abstraction [10]. Reachability analysis is used to determine if an error state can be reached. BLAST [16] uses lazy abstraction, an automated abstraction and refinement process that abstracts the program to the proper amount of precision necessary to verify a particular property. The SPIN model checker [17] is popular for verifying distributed system protocols.

6. Conclusions and Future Work

In this paper, we describe an approach for dynamically checking for software faults caused by improperly bounded program input. Our dynamic approach overcomes many limitations of static analysis while reducing the dependence on the input. On a given program path, the range of all possible input values is validated to ensure that no input-related errors can occur. This is accomplished by shadowing all inputs with state variables.

Integers are shadowed with an interval constraint containing the bounds an input variable may hold. Control decisions narrow the interval constraint and operations adjust the constraints. At potentially dangerous array access sites, the range of variable is checked against the bounds of the referenced array. Strings are shadowed by the maximum possible length the string may hold and a flag that indicated

if the string must be null terminated or not. Control decisions based on the string length can reduce the maximum size. String functions are checked to make sure that source strings are properly null terminated and destination strings have sufficient space for all possible string sizes.

Our approach is generic and can be applied to any program. We applied our technique to eight programs and found a total of 16 bugs including two security flaws in OpenSSH. The cost of our checker's accuracy is run-time performance, with some programs experiencing more than two orders of magnitude slowdown. As such, in its current form our approach is best targeted to development testing where precision is more important.

In the future, we plan to address the performance impacts by adding an analysis phase to the compiler that will eliminate unnecessary instrumentation. This phase will be aware of how the instrumentation works, making possible optimizations such as copy propagation on the input interval constraints. The static analysis can also be used to improve the quality and scope of bug detection. For example, static analysis could identify manual `strcpy` loops that copy elements individually. Another avenue for future work is to address areas of unsoundness by adding symbolic analysis support. This will also reduce the number of false alarms.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work is supported under the DARPA/MARCO Gigascale Silicon Research Center and a National Science Foundation Graduate Fellowship. Equipment support was provided by Intel.

References

[1] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002.

[2] T. Austin, S. Breach, and G. Sohi. Efficient Detection of All Pointer and Array Access Errors. Technical Report #1197, Computer Science Department, University of Wisconsin, Dec. 1993.

[3] T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Workshop on Model Checking of Software, May 2001.

[4] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. Proceedings of the Conference on Programming Language Design and Implementation, June 2000.

[5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software Practice and Experience, July 2000.

[6] Checker. http://www.gnu.org/software/checker/checker.html

[7] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using Symbolic Execution for Verifying Safety-Critical Systems. Proceedings of the 9th International Symposium on Foundations of Software Engineering, Sept. 2001.

[8] 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. Proceedings of the 7th USENIX Security Conference, Jan. 1998.

[9] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. IEEE Software, Jan./Feb. 2002.

[10] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. Proceedings of the Symposium on Principles of Programming Languages, Jan. 2002.

[11] R. Gupta. A Fresh Look at Optimizing Array Bound Checks. Conference on Programming Language Design and Implementation, June 1990.

[12] J. Forrester and B. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. Proc. of the 4th USENIX Windows System Symposium, Aug. 2000.

[13] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. 1992 Winter USENIX Conference, Jan. 1992.

[14] E. Haugh and M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. Proceedings of the 10th Network and Distributed System Security Symposium, Feb. 2003.

[15] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, Aug. 1992.

[16] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proceedings of the Symposium on Principles of Programming Languages, Jan. 2002.

[17] G. Holzmann. The Spin Model Checker. IEEE Transactions on Software Engineering, May 1997.

[18] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. Proceedings of the USENIX Annual Technical Conference, June 2002.

[19] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-base programming environment for the C language. Proceedings of the Summer USENIX Conference, 1988.

[20] K. Lhee and S. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. Proceedings of the 11th USENIX Security Symposium, Aug. 2002.

[21] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. Proceedings of Operating Systems Design and Implementation, Oct. 1996.

[22] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. Proceedings of the Symposium on Principles of Programming Languages, January 2002.

[23] Parasoft Corporation. Insure++: An Automatic Runtime Error Detection Tool. Technical Report PS961-INS1.

[24] B. Perens. Electric Fence. <http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz>

[25] Perl v5.6 Documentation: perlsec. <http://www.perldoc.com/perl5.6/pod/perlsec.html>

[26] Pointer-Intensive Benchmark Suite <http://www.cs.wisc.edu/~austin/ptr-dist.html>

[27] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accesses Memory Regions. Proceedings of the Conference on Programming Languages Design and Implementation, June 2000.

[28] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. Proceedings of the 10th USENIX Security Symposium, Aug. 2001.

[29] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed Security Symposium, Feb. 2000.

[30] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. Proceedings of the 14th Symposium on Operating System Principles, June 1993.