

Statically Detecting Likely Buffer Overflow Vulnerabilities

David Larochelle

larochelle@cs.virginia.edu

University of Virginia, Department of Computer Science

David Evans

evans@cs.virginia.edu

University of Virginia, Department of Computer Science

Abstract

Buffer overflow attacks may be today's single most important security threat. This paper presents a new approach to mitigating buffer overflow vulnerabilities by detecting likely vulnerabilities through an analysis of the program source code. Our approach exploits information provided in semantic comments and uses lightweight and efficient static analyses. This paper describes an implementation of our approach that extends the LCLint annotation-assisted static checking tool. Our tool is as fast as a compiler and nearly as easy to use. We present experience using our approach to detect buffer overflow vulnerabilities in two security-sensitive programs.

1. Introduction

Buffer overflow attacks are an important and persistent security problem. Buffer overflows account for approximately half of all security vulnerabilities [CWPBW00, WFBA00]. Richard Pethia of CERT identified buffer overflow attacks as the single most important security problem at a recent software engineering conference [Pethia00]; Brian Snow of the NSA predicted that buffer overflow attacks would still be a problem in twenty years [Snow99].

Programs written in C are particularly susceptible to buffer overflow attacks. Space and performance were more important design considerations for C than safety. Hence, C allows direct pointer manipulations without any bounds checking. The standard C library includes many functions that are unsafe if they are not used carefully. Nevertheless, many security-critical programs are written in C.

Several run-time approaches to mitigating the risks associated with buffer overflows have been proposed. Despite their availability, these techniques are not used widely enough to substantially mitigate the effectiveness of buffer overflow attacks. The next section describes representative run-time approaches and speculates on why they are not more widely used. We propose, instead, to tackle the problem by detecting likely buffer overflow vulnerabilities through a static analysis of program source code. We have implement-

ed a prototype tool that does this by extending LCLint [Evans96]. Our work differs from other work on static detection of buffer overflows in three key ways: (1) we exploit semantic comments added to source code to enable local checking of interprocedural properties; (2) we focus on lightweight static checking techniques that have good performance and scalability characteristics, but sacrifice soundness and completeness; and (3) we introduce loop heuristics, a simple approach for efficiently analyzing many loops found in typical programs.

The next section of this paper provides some background on buffer overflow attacks and previous attempts to mitigate the problem. Section 3 gives an overview of our approach. In Section 4, we report on our experience using our tool on *wu-ftpd* and *BIND*, two security-sensitive programs. The following two sections provide some details on how our analysis is done. Section 7 compares our work to related work on buffer overflow detection and static analysis.

2. Buffer Overflow Attacks and Defenses

The simplest buffer overflow attack, *stack smashing* [AlephOne96], overwrites a buffer on the stack to replace the return address. When the function returns, instead of jumping to the return address, control will jump to the address that was placed on the stack by the attacker. This gives the attacker the ability to execute arbitrary code. Programs written in C are particularly

susceptible to this type of attack. C provides direct low-level memory access and pointer arithmetic without bounds checking. Worse, the standard C library provides unsafe functions (such as `gets`) that write an unbounded amount of user input into a fixed size buffer without any bounds checking [ISO99]. Buffers stored on the stack are often passed to these functions. To exploit such vulnerabilities, an attacker merely has to enter an input larger than the size of the buffer and encode an attack program binary in that input. The Internet Worm of 1988 [Spafford88, RE89] exploited this type of buffer overflow vulnerability in fingerd. More sophisticated buffer overflow attacks may exploit unsafe buffer usage on the heap. This is harder, since most programs do not jump to addresses loaded from the heap or to code that is stored in the heap.

Several run-time solutions to buffer overflow attacks have been proposed. StackGuard [CPMH+98] is a compiler that generates binaries that incorporate code designed to prevent stack smashing attacks. It places a special value on the stack next to the return address, and checks that it has not been tampered with before jumping. Baratloo, Singh and Tsai describe two run-time approaches: one replaces unsafe library functions with safe implementations; the other modifies executables to perform sanity checking of return addresses on the stack before they are used [BST00].

Software fault isolation (SFI) is a technique that inserts bit mask instructions before memory operations to prevent access of out-of-range memory [WLAG93]. This alone does not offer much protection against typical buffer overflow attacks since it would not prevent a program from writing to the stack address where the return value is stored. Generalizations of SFI can insert more expressive checking around potentially dangerous operations to restrict the behavior of programs more generally. Examples include Janus, which observes and mediates behavior by monitoring system calls [GWTB96]; Naccio [ET99, Evans00a] and PSLang/PoET [ES99, ES00] which transform object programs according to a safety policy; and Generic Software Wrappers [FBF99] which wraps system calls with security checking code.

Buffer overflow attacks can be made more difficult by modifications to the operating system that put code and data in separate memory segments, where the code segment is read-only and instructions cannot be executed from the data segment. This does not eliminate the buffer overflow problem, however, since an attacker can still overwrite an address stored on the stack to make the program jump to any point in the code segment. For programs that use shared libraries, it

is often possible for an attacker to jump to an address in the code segment that can be used maliciously (e.g., a call to `system`). Developers decided against using this approach in the Linux kernel since it did not solve the real problem and it would prevent legitimate uses of self-modifying code [Torvalds98, Coolbaugh99].

Despite the availability of these and other run-time approaches, buffer overflow attacks remain a persistent problem. Much of this may be due to lack of awareness of the severity of the problem and the availability of practical solutions. Nevertheless, there are legitimate reasons why the run-time solutions are unacceptable in some environments. Run-time solutions always incur some performance penalty (StackGuard reports performance overhead of up to 40% [CBDP+99]). The other problem with run-time solutions is that while they may be able to detect or prevent a buffer overflow attack, they effectively turn it into a denial-of-service attack. Upon detecting a buffer overflow, there is often no way to recover other than terminating execution.

Static checking overcomes these problems by detecting likely vulnerabilities before deployment. Detecting buffer overflow vulnerabilities by analyzing code in general is an undecidable problem.¹ Nevertheless, it is possible to produce useful results using static analysis. Rather than attempting to verify that a program has no buffer overflow vulnerabilities, we wish to have reasonable confidence of detecting a high fraction of likely buffer overflow vulnerabilities. We are willing to accept a solution that is both unsound and incomplete. This means that our checker will sometimes generate false warnings and sometimes miss real problems. Our goal is to produce a tool that produces useful results for real programs with a reasonable effort. The next section describes our approach. We compare our work with other static approaches to detecting buffer overflow vulnerabilities in Section 7.

3. Approach

Our static analysis tool is built upon LCLint [EGHT94, Evans96, Evans00b], an annotation-assisted lightweight static checking tool. Examples of problems detected by LCLint include violations of information hiding, inconsistent modifications of caller-visible state or uses of global variables, misuses of possibly NULL references, uses of dead storage, memory leaks and problems with parameters aliasing. LCLint is actually

¹ We can trivially reduce the halting problem to the buffer overflow detection problem by inserting code that causes a buffer overflow before all halt instructions.

used by working programmers, especially in the open source development community [Orcero00, PG00].

Our approach is to exploit semantic comments (henceforth called *annotations*) that are added to source code and standard libraries. Annotations describe programmer assumptions and intents. They are treated as regular C comments by the compiler, but recognized as syntactic entities by LCLint using the @ following the /* to identify a semantic comment. For example, the annotation /*@nonnull@*/ can be used syntactically like a type qualifier. In a parameter declaration, it indicates that the value passed for this parameter may not be NULL. Although annotations can be used on any declaration, for this discussion we will focus exclusively on function and parameter declarations. We can also use annotations similarly in declarations of global and local variables, types and type fields.

Annotations constrain the possible values a reference can contain either before or after a function call. For example, the /*@nonnull@*/ annotation places a constraint on the parameter value before the function body is entered. When LCLint checks the function body, it assumes the initial value of the parameter is not NULL. When LCLint checks a call site, it reports a warning unless it can determine that the value passed as the corresponding parameter is never NULL.

Prior to this work, all annotations supported by LCLint classified references as being in one of a small number of possible states. For example, the annotation /*@null@*/ indicated that a reference may be NULL, and the annotation /*@nonnull@*/ indicated that a reference is not NULL. In order to do useful checking of buffer overflow vulnerabilities, we need annotations that are more expressive. We are concerned with how much memory has been allocated for a buffer, something that cannot be adequately modeled using a finite number of states. Hence, we need to extend LCLint to support a more general annotation language. The annotations are more expressive, but still within the spirit of simple semantic comments added to programs.

The new annotations allow programmers to explicitly state function preconditions and postconditions using *requires* and *ensures* clauses.² We can use these clauses to describe assumptions about buffers that are passed to functions and constrain the state of buffers when functions return. For the analyses described in

² The original Larch C interface language LCL [GH93], on which LCLint's annotation language was based, did include a notion of general preconditions and postconditions specified by *requires* and *ensures* clauses.

this paper, four kinds of assumptions and constraints are used: *minSet*, *maxSet*, *minRead* and *maxRead*.³

When used in a *requires* clause, the *minSet* and *maxSet* annotations describe assumptions about the lowest and highest indices of a buffer that may be safely used as an lvalue (e.g., on the left-hand side of an assignment). For example, consider a function with an array parameter *a* and an integer parameter *i* that has a precondition *requires maxSet(a) >= i*. The analysis assumes that at the beginning of the function body, *a[i]* may be used as an lvalue. If *a[i+1]* were used before any modifications to the value of *a* or *i*, LCLint would generate a warning since the function preconditions are not sufficient to guarantee that *a[i+1]* can be used safely as an lvalue. Arrays in C start with index 0, so the declaration

```
char buf[MAXSIZE]
```

generates the constraints

```
maxSet(buf) = MAXSIZE - 1 and
minSet(buf) = 0.
```

Similarly, the *minRead* and *maxRead* constraints indicate the minimum and maximum indices of a buffer that may be read safely. The value of *maxRead* for a given buffer is always less than or equal to the value of *maxSet*. In cases where there are elements of the buffer have not yet been initialized, the value of *maxRead* may be lower than the value of *maxSet*.

At a call site, LCLint checks that the preconditions implied by the *requires* clause of the called function are satisfied before the call. Hence, for the *requires maxSet(a) >= i* example, it would issue a warning if it cannot determine that the array passed as *a* is allocated to hold at least as many elements as the value passed as *i*. If *minSet* or *maxSet* is used in an *ensures* clause, it indicates the state of a buffer after the function returns. Checking at the call site proceeds by assuming the postconditions are true after the call returns.

For checking, we use an annotated version of the standard library headers. For example, the function *strcpy* is annotated as⁴:

³ LCLint also supports a *nullterminated* annotation that denotes storage that is terminated by the null character. Many C library functions require null-terminated strings, and can produce buffer overflow vulnerabilities if they are passed a string that is not properly null-terminated. We do not cover the *nullterminated* annotation and related checking in this paper. For information on it, see [LHSS00].

⁴ The standard library specification of *strcpy* also includes other LCLint annotations: a *modifies* clause that indicates that the only thing that may be modified by *strcpy* is the storage referenced by *s1*, an *out* annotation on *s1* to indicate that it

```
char *strcpy (char *s1, const char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead(s2)
/\ result == s1@*/;
```

The `requires` clause specifies the precondition that the buffer `s1` is allocated to hold at least as many characters as are readable in the buffer `s2` (that is, the number of characters up to and including its null terminator). The postcondition reflects the behavior of `strcpy` – it copies the string pointed to by `s2` into the buffer `s1`, and returns that buffer. In `ensures` clauses, we use the `result` keyword to denote the value returned by the function.

Many buffer overflows result from using library functions such as `strcpy` in unsafe ways. By annotating the standard library, many buffer overflow vulnerabilities can be detected even before adding any annotations to the target program. Selected annotated standard library functions are shown in Appendix A.

4. Experience

In order to test our approach, we used our tool on `wu-ftpd`, a popular open source ftp server, and `BIND` (Berkeley Internet Name Domain), a set of domain name tools and libraries that is considered the reference implementation of DNS. This section describes the process of running LCLint on these applications, and illustrates how our checking detected both known and unknown buffer overflow vulnerabilities in each application.

4.1 wu-ftpd

We analyzed `wu-ftp-2.5.0`⁵, a version with known security vulnerabilities.

Running LCLint is similar to running a compiler. It is typically run from the command line by listing the

need not point to defined storage when `strcpy` is called, a unique annotation on `s1` to indicate that it may not alias the same storage as `s2`, and a returned annotation on `s1` to indicate that the returned pointer references the same storage as `s1`. For clarity, the examples in this paper show only the annotations directly relevant to detecting buffer overflow vulnerabilities. For more information on other LCLint annotations, see [Evans96, Evans00c].

⁵ The source code for `wu-ftpd` is available from <http://www.wu-ftpd.org>. We analyzed the version in <ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.5.0.tar.gz>. We configured `wu-ftpd` using the default configuration for FreeBSD systems. Since LCLint performs most of its analyses on code that has been pre-processed, our analysis did not examine platform-specific code in `wu-ftpd` for platforms other than FreeBSD.

source code files to check, along with flags that set checking parameters and control which classes of warnings are reported. It takes just over a minute for LCLint to analyze all 17 000 lines of `wu-ftpd`. Running LCLint on the entire unmodified source code for `wu-ftpd` without adding any annotations resulted in 243 warnings related to buffer overflow checking.

Consider a representative message⁶:

```
ftpd.c:1112:2: Possible out-of-bounds store. Unable to
resolve constraint:
  maxRead ((entry->arg[0] @ ftpd.c:1112:23)) <= (1023)
needed to satisfy precondition:
  requires maxSet ((ls_short @ ftpd.c:1112:14))
    >= maxRead ((entry->arg[0] @ ftpd.c:1112:23))
derived from strcpy precondition:
  requires maxSet (<param 1>) >= maxRead (<param 2>)
```

Relevant code fragments are shown below with line 1112 in bold:

```
char ls_short[1024];
...
extern struct aclmember *
getaclentry(char *keyword,
            struct aclmember **next);
...
int main(int argc, char **argv,
        char **envp)
{
    ...
    entry = (struct aclmember *) NULL;
    if (getaclentry("ls_short", &entry)
        && entry->arg[0]
        && (int)strlen(entry->arg[0]) > 0)
    {
        strcpy(ls_short, entry->arg[0]);
    }
    ...
}
```

This code is part of the initialization code that reads configuration files. Several buffer overflow vulnerabilities were found in the `wu-ftpd` initialization code. Although this vulnerability is not likely to be exploited, it can cause security holes if an untrustworthy user is able to alter configuration files.

The warning message indicates that a possible out-of-bounds store was detected on line 1112 and contains information about the constraint LCLint was unable to resolve. The warning results from the function call to `strcpy`. LCLint generates a precondition constraint corresponding to the `strcpy` `requires` clause

⁶ For our prototype implementation, we have not yet attempted to produce messages that can easily be interpreted by typical programmers. Instead, we generate error messages that reveal information useful to the LCLint developers. Generating good error messages is a challenging problem; we plan to devote more effort to this before publicly releasing our tool.

maxSet(s1) >= maxRead(s2) by substituting the actual parameters:

```
maxSet (ls_short @ ftpd.c:1112:14)
  >= maxRead (entry->arg[0] @ ftpd.c:1112:23).
```

Note that the locations of the expressions passed as actual parameters are recorded in the constraint. Since values of expressions may change through the code, it is important that constraints identify values at particular program points.

The global variable `ls_short` was declared as an array of 1024 characters. Hence, LCLint determines `maxSet (ls_short)` is 1023. After the call to `getaclentry`, the local `entry->arg[0]` points to a string of arbitrary length read from the configuration file. Because there are no annotations on the `getaclentry` function, LCLint does not assume anything about its behavior. In particular, the value of `maxRead (entry->arg[0])` is unknown. LCLint reports a possible buffer misuse, since the constraint derived from the `strcpy` requires clause may not be satisfied if the value of `maxRead (entry->arg[0])` is greater than 1023.

To fix this problem, we modified the code to handle these values safely by using `strncpy`. Since `ls_short` is a fixed size buffer, a simple change to use `strncpy` and store a null character at the end of the buffer is sufficient to ensure that the code is safe.⁷

In other cases, eliminating a vulnerability involved both changing the code and adding annotations. For example, LCLint generated a warning for a call to `strcpy` in the function `acl_getlimit`:

```
int acl_getlimit(char *class,
                char *msgpathbuf) {
    int limit;
    struct aclmember *entry = NULL;

    if (msgpathbuf) *msgpathbuf = '\0';
    while (getaclentry("limit", &entry)) {
        ...
        if (!strcasecmp(class, entry->arg[0]))
        {
            ...
            if (entry->arg[3]
                && msgpathbuf != NULL)
                strcpy(msgpathbuf, entry->arg[3]);
            ...
        }
    }
}
```

If the size of `msgpathbuf` is less than the length of the string in `entry->arg[3]`, there is a buffer overflow. To fix this we replaced the `strcpy` call with a safe call to `strncpy`:

⁷ Because `strcpy` does not guarantee null termination, it is necessary to explicitly put a null character at the end of the buffer.

```
strncpy(msgpathbuf, entry->arg[3], 199);
msgpathbuf[199] = '\0';
```

and added a requires clause to the function declaration:

```
/*@requires maxSet(msgpathbuf) >= 199@*/
```

The requires clause documents an assumption (that may be incorrect) about the size of the buffer passed to `acl_getlimit`. Because of the constraints denoted by the requires clauses, LCLint does not report a warning for the call to `strcpy`.

When call sites are checked, LCLint produces a warning if it is unable to determine that this requires clause is satisfied. Originally, we had modified the function `acl_getlimit` by adding the precondition `maxSet (msgpathbuf) >= 1023`. After adding this precondition, LCLint produced a warning for a call site that passed a 200-byte buffer to `acl_getlimit`. Hence, we replaced the requires clause with the stronger constraint and used 199 as the parameter to `strncpy`.

This vulnerability was still present in the current version of `wu-ftpd`. We contacted the `wu-ftpd` developers who acknowledged the bug but did not consider it security critical since the string in question is read from a local file not user input [Luckin01, Lundberg01].

In addition to the previously unreported buffer overflows in the initialization code, LCLint detected a known buffer overflow in `wu-ftpd`. The buffer overflow occurs in the function `do_elem` shown below, which passes a global buffer and its parameters to the library function `strcat`. The function `mapping_chdir` calls `do_elem` with a value entered by the remote user as its parameter. Because `wu-ftpd` fails to perform sufficient bounds checking, a remote user is able to exploit this vulnerability to overflow the buffer by carefully creating a series of directories and executing the `cd` command.⁸

```
char mapped_path [200];
...
void do_elem(char *dir) {
    ...
    if (!(mapped_path[0] == '/'
        && mapped_path[1] == '\0'))
        strcat (mapped_path, "/");
    strcat (mapped_path, dir);
}
```

⁸ Advisories for this vulnerability can be found at <http://www.cert.org/advisories/CA-1999-13.html> and ftp://www.auscert.org.au/security/advisory/AA-1999.01.wu-ftpd.mapping_chdir.vul.

LCLint generates warnings for the unsafe calls to `strcat`. This was fixed in latter versions of `wu-ftpd` by calling `strncat` instead of `strcat`.

Because of the limitations of static checking, LCLint sometimes generates spurious error messages. If the user believes the code is correct, annotations can be added to precisely suppress spurious messages.

Often the code was too complex for LCLint to analyze correctly. For example, LCLint reports a spurious warning for this code fragment since it cannot determine that $((1.0*j*rand()) / (RAND_MAX + 1.0))$ always produces a value between 1 and j :

```
i = passive_port_max
  - passive_port_min + 1;
port_array = calloc (i, sizeof (int));
for (i = 3; ... && (i > 0); i--) {
  for (j = passive_port_max
        - passive_port_min + 1;
        ... && (j > 0); j--) {
    k = (int) ((1.0 * j * rand())
              / (RAND_MAX + 1.0));
    pasv_port_array [j-1]
      = port_array [k];
```

Determining that the `port_array[k]` reference is safe would require far deeper analysis and more precise specifications than is feasible within a lightweight static checking tool.

Detecting buffer overflows with LCLint is an iterative process. Many of the constraints we found involved functions that are potentially unsafe. We added function preconditions to satisfy these constraints where possible. In certain cases, the code was too convoluted for LCLint to determine that our preconditions satisfied the constraints. After convincing ourselves the code was correct, we added annotations to suppress the spurious warnings.

Before any annotations were added, running LCLint on `wu-ftpd` resulted in 243 warnings each corresponding to an unresolved constraint. We added 22 annotations to the source code through an iterative process similar to the examples described above. Nearly all of the annotations were used to indicate preconditions constraining the value of `maxSet` for function parameters.

After adding these annotations and modifying the code, running LCLint produced 143 warnings. Of these, 88 reported unresolved constraints involving `maxSet`. While we believe the remaining warnings did not indicate bugs in `wu-ftpd`, LCLint's analyses were not sufficiently powerful to determine the code was safe. Although this is a higher number of spurious warnings than we would like, most of the spurious warnings can

be quickly understood and suppressed by the user. The source code contains 225 calls to the potentially buffer overflowing functions `strcat`, `strcpy`, `strncat`, `strncpy`, `fgets` and `gets`. Only 18 of the unresolved warnings resulted from calls to these functions. Hence, LCLint is able to determine that 92% of these calls are safe automatically. The other warnings all dealt with classes of problems that could not be detected through simple lexical techniques.

4.2 BIND

BIND is a key component of the Internet infrastructure. Recently, the Wall Street Journal identified buffer overflow vulnerabilities in BIND as a critical threat to the Internet [WSJ01]. We focus on named, the DNS sever portion of BIND, in this case study. We analyzed BIND version 8.2.2p7⁹, a version with known bugs. BIND is larger and more complex than `wu-ftpd`. The name server portion of BIND, named, contains approximately 47 000 lines of C including shared libraries. LCLint took less than three and a half minutes to check all of the named code.

We limited our analysis to a subset of named because of the time required for human analysis. We focused on three files: `ns_req.c` and two library files that contain functions which are called extensively by `ns_req.c`: `ns_name.c` and `ns_sign.c`. These files contain slightly more than 3 000 lines of code.

BIND makes extensive use of functions in its internal library rather than C library functions. In order to accurately analyze individual files, we needed to annotate the library header files. The most accurate way to annotate the library would be to iteratively run LCLint on the library and add annotations. However, the library was extremely large and contains deeply nested call chains. To avoid the human analysis this would require, we added annotations to some of the library functions without annotating all the dependent functions. In many cases, we were able to guess preconditions by using comments or the names of function parameters. For example, several functions took a pointer parameter (`p`) and another parameter encoding its size (`psize`), from which we inferred a precondition `MaxSet(p) >= (psize - 1)`. After annotating selected BIND library functions, we were able to check the chosen files without needing to fully annotate all of BIND.

LCLint produces warnings for a series of unguarded buffer writes in the function `req_query`. The code in

⁹ The source code is available at <ftp://ftp.isc.org/isc/bind/src/8.2.2-P7/bind-src.tar.gz>

question is called in response to a specific type of query which requests information concerning the domain name server version. BIND appends a response to the buffer containing the query that includes a global string read from a configuration file. If the default configuration is used, the code is safe because this function is only called with buffers that are large enough to store the response. However, the restrictions on the safe use of this function are not obvious and could easily be overlooked by someone modifying the code. Additionally, it is possible that an administrator could reconfigure BIND to use a value for the server version string large enough to make the code unsafe. The BIND developers agreed that a bounds check should be inserted to eliminate this risk [Andrews01].

BIND uses extensive run time bounds checking. This type of defensive programming is important for writing secure programs, but does not guarantee that a program is secure. LCLint detected a known buffer overflow in a function that used run time checking but specified buffer sizes incorrectly.¹⁰

The function `ns_req` examines a DNS query and generates a response. As part of its message processing, it looks for a signature and signs its response with the function `ns_sign`. LCLint reported that it was unable to satisfy a precondition for `ns_sign` that requires the size of the message buffer be accurately described by a size parameter. This precondition was added when we initially annotated the shared library. A careful hand analysis of this function reveals that due to careless modification of variables denoting buffer length, it is possible for the buffer length to be specified incorrectly if the message contains a signature but a valid key is not found. This buffer overflow vulnerability was introduced when a digital signature feature was added to BIND (ironically to increase security). Static analysis tools can be used to quickly alert programmers to assumptions that are broken by incremental code changes.

Based on our case studies, we believe that LCLint is a useful tool for improving the security of programs. It does not detect all possible buffer overflow vulnerabilities, and it can generate spurious warnings. In practice, however, it provides programmers concerned about security vulnerabilities with useful assistance, even for large, complex programs. In addition to aiding in the detection of exploitable buffer overflows, the process of adding annotations to code encourages a disciplined style of programming and

produces programs that include reliable and precise documentation.

5. Implementation

Our analysis is implemented by combining traditional compiler data flow analyses with constraint generation and resolution. Programs are analyzed at the function level; all interprocedural analyses are done using the information contained in annotations.

We support four types of constraints corresponding to the annotations introduced in Section 2: `maxSet`, `minSet`, `maxRead`, and `minRead`. Constraints can also contain constants and variables and allow the arithmetic operations: `+` and `-`. Terms in constraints can refer to any C expression, although our analysis will not be able to evaluate some C expressions statically.

The full constraint grammar is:

```

constraint ⇒ (requires | ensures)
constraintExpression relOp constraintExpression
relationalOp ⇒ == | > | >= | < | <=
constraintExpression ⇒
    constraintExpression binaryOp constraintExpression
    | unaryOp ( constraintExpression )
    | term
binaryOp ⇒ + | -
unaryOp ⇒ maxSet | maxRead | minSet | minRead
term ⇒ variable | C expression | literal | result

```

Source-code annotations allow arbitrary constraints (as defined by our constraint grammar) to be specified as the preconditions and postconditions of functions. Constraints can be conjoined (using `^`), but there is no support for disjunction. All variables used in constraints have an associated location. Since the value stored by a variable may change in the function body, it is important that the constraint resolver can distinguish the value at different points in the program execution.

Constraints are generated at the expression level and stored in the corresponding node in the parse tree. Constraint resolution is integrated with the checking by resolving constraints at the statement level as checking traverses up the parse tree. Although this limits the power of our analysis, it ensures that it will be fast and simple. The remainder of this section describes briefly how constraints are represented, generated and resolved.

Constraints are generated for C statements by traversing the parse tree and generating constraints for each subexpression. We determine constraints for a statement by conjoining the constraints of its

¹⁰ An advisory for this vulnerability can be found at <http://lwn.net/2001/0201/a/covert-bind.php3>.

subexpressions. This assumes subexpressions cannot change state that is used by other subexpressions of the same expression. The semantics of C make this a valid assumption for nearly all expressions – it is undefined behavior in C for two subexpressions not separated by a sequence point to read and write the same data. Since LCLint detects and warns about this type of undefined behavior, it is reasonable for the buffer overflow checking to rely on this assumption. A few C expressions do have intermediate sequence points (such as the comma operator which specifies that the left operand is always evaluated first) and cannot be analyzed correctly by our simplified assumptions. In practice, this has not been a serious limitation for our analysis.

Constraints are resolved at the statement level in the parse tree and above using axiomatic semantics techniques. Our analysis attempts to resolve constraints using postconditions of earlier statements and function preconditions. To aid in constraint resolution, we simplify constraints using standard algebraic techniques such as combining constants and substituting terms. We also use constraint-specific simplification rules such as $\text{maxSet}(ptr + i) = \text{maxSet}(ptr) - i$. We have similar rules for maxRead , minSet , and minRead .

Constraints for statement lists are produced using normal axiomatic semantics rules and simple logic to combine the constraints of individual statements. For example, the code fragment

```
1   t++;
2   *t = 'x';
3   t++;
```

leads to the constraints:

```
requires maxSet(t @ 1:1) >= 1,
ensures maxRead(t @ 3:4) >= -1 and
ensures (t @ 3:4) = (t @ 1:1) + 2.
```

The assignment to `*t` on line 2 produces the constraint $\text{requires maxSet}(t @ 2:2) \geq 0$. The increment on line 1 produces the constraint $\text{ensures}(t @ 1:4) = (t @ 1:1) + 1$. The increment constraint is substituted into the maxSet constraint to produce $\text{requires maxSet}(t @ 1:1 + 1) \geq 0$. Using the constraint-specific simplification rule, this simplifies to $\text{requires maxSet}(t @ 1:1) - 1 \geq 0$ which further simplifies to $\text{requires maxSet}(t @ 1:1) \geq 1$.

6. Control Flow

Statements involving control flow such as `while` and `for` loops and `if` statements, require more complex analysis than simple statement lists. For `if` statements and loops, the predicate often provides a guard that makes a possibly unsafe operation safe. In order to

analyze such constructs well, LCLint must take into account the value of the predicate on different code paths. For each predicate, LCLint generates three lists of postcondition constraints: those that hold regardless of the truth value of the predicate, those that hold when the predicate evaluates to true, and those that hold when the predicate evaluates to false.

To analyze an `if` statement, we develop branch specific guards based on our analysis of the predicate and use these guards to resolve constraints within the body. For example, in the statement

```
if (sizeof (s1) > strlen (s2))
    strcpy(s1, s2);
```

if `s1` is a fixed-size array, $\text{sizeof}(s1)$ will be equal to $\text{maxSet}(s1) + 1$. Thus the `if` predicate allows LCLint to determine that the constraint $\text{maxSet}(s1) \geq \text{maxRead}(s2)$ holds on the true branch. Based on this constraint LCLint determines that the call to `strcpy` is safe.

Looping constructs present additional problems. Previous versions of LCLint made a gross simplification of loop behavior: all `for` and `while` loops in the program were analyzed as though the body executed either zero or one times. Although this is clearly a ridiculous assumption, it worked surprisingly well for the types of analyses done by LCLint. For the buffer overflow analyses, this simplified view of loop semantics does not provide satisfactory results – to determine whether `buf[i]` is a potential buffer overflow, we need to know the range of values `i` may represent. Analyzing the loop as though its body executed only once would not provide enough information about the possible values of `i`.

In a typical program verifier, loops are handled by requiring programmers to provide loop invariants. Despite considerable effort [Wegbreit75, Cousot77, Collins88, IS97, DLNS98, SI98], no one has yet been able to produce tools that generate suitable loop invariants automatically. Some promising work has been done towards discovering likely invariants by executing programs [ECGN99], but these techniques require well-constructed test suites and many problems remain before this could be used to produce the kinds of loop invariants we need. Typical programmers are not able or willing to annotate their code with loop invariants, so for LCLint to be effective we needed a method for handling loops that produces better results than our previous gross simplification method, but did not require expensive analyses or programmer-supplied loop invariants.

Our solution is to take advantage of the idioms used by typical C programmers. Rather than attempt to handle all possible loops in a general way, we observe that a large fraction of the loops in most C programs are written in a stylized and structured way. Hence, we can develop heuristics for identifying and analyzing loops that match certain common idioms. When a loop matches a known idiom, corresponding heuristics can be used to guess how many times the loop body will execute. This information is used to add additional preconditions to the loop body that constrain the values of variables inside the loop.

To further simplify the analysis, we assume that any buffer overflow that occurs in the loop will be apparent in either the first or last iterations. This is a reasonable assumption in almost all cases, since it would be quite rare for a program to contain a loop where the extreme values of loop variables were not on the first and last iterations. This allows simpler and more efficient loop checking. To analyze the first iteration of the loop, we treat the loop as an `if` statement and use the techniques described above. To analyze the last iteration we use a series of heuristics to determine the number of loop iterations and generate additional constraints based on this analysis.

An example loop heuristic analyzes loops of the form

```
for (index = 0; expr; index++) body
```

where the *body* and *expr* do not modify the *index* variable and *body* does not contain a statement (e.g., a `break`) that could interfere with normal loop execution. Analyses performed by the original LCLint are used to aid loop heuristic pattern matching. For example, we use LCLint's modification analyses to determine that the loop body does not modify the index variable.

For a loop that matches this idiom, it is reasonable to assume that the number of iterations can be determined solely from the loop predicate. As with `if` statements, we generate three lists of postcondition constraints for the loop test. We determine the terminating condition of the loop by examining the list of postcondition constraints that apply specifically to the true branch. Within these constraints, we look for constraints of the form $index \leq e$. For each of these constraints, we search the increment part of the loop header for constraints matching the form $index = index + 1$. If we find a constraint of this form, we assume the loop runs for e iterations.

Of course, many loops that match this heuristic will not execute for e iterations. Changes to global state or other variables in the loop body could affect the value of e . Hence, our analysis is not sound or complete. For the

programs we have tried so far, we have found this heuristic works correctly.

Abstract syntax trees for loops are converted to a canonical form to increase their chances of matching a known heuristic. After canonicalization, this loop pattern matches a surprisingly high number of cases. For example, in the loop

```
for (i = 0; buffer[i]; i++) body
```

the postconditions of the loop predicate when the body executes would include the constraint `ensures i < maxRead(buffer)`. This would match the pattern so LCLint could determine that the loop executes for `maxRead(buffer)` iterations.

Several other heuristics are used to match other common loop idioms used in C programs. We can generalize the first heuristic to cases where the initial index value is not known. If LCLint can calculate a reasonable upper bound on the number of iterations (for example, if we can determine that the initial value of the index is always non-negative), it can determine an upper bound on the number of loop iterations. This can generate false positives if LCLint overestimates the actual number of loop iterations, but usually gives a good enough approximation for our purposes.

Another heuristic recognizes a common loop form in which a loop increments and tests a pointer. Typically, these loops match the pattern:

```
for (init; *buf; buf++)
```

A heuristic detects this loop form and assumes that loop executes for `maxRead(buf)` iterations.

After estimating the number of loop iterations, we use a series of heuristics to generate reasonable constraints for the last iteration. To do this, we calculate the value of each variable in the last iteration. If a variable is incremented in the loop, we estimate that in the last iteration the variable is the sum of the number of loop iterations and the value of the variable in the first iteration. For the loop to be safe, all loop preconditions involving the variable must be satisfied for the values of the variable in both the first and last iterations. This heuristic gives satisfactory results in many cases.

Our heuristics were initially developed based on our analysis of `wu-ftpd`. We found that our heuristics were effective for `BIND` also. To handle `BIND`, a few additional heuristics were added. In particular, `BIND` frequently used comparisons of pointer addresses to ensure a memory accesses is safe. Without an appropriate heuristic, LCLint generated spurious warnings for these cases. We added appropriate heuristics to

handle these situations correctly. While we expect experience with additional programs would lead to the addition of new loop heuristics, it is encouraging that only a few additional heuristics were needed to analyze BIND.

Although no collection of loop heuristics will be able to correctly analyze all loops in C programs, our experience so far indicates that a small number of loop heuristics can be used to correctly analyze most loops in typical C programs. This is not as surprising as it might seem – most programmers learn to code loops from reading examples in standard texts or other people’s code. A few simple loop idioms are sufficient for programming many computations.

7. Related Work

In Section 2, we described run-time approaches to the buffer overflow problem. In this section, we compare our work to other work on static analysis.

It is possible to find some program flaws using lexical analysis alone. Unix `grep` is often used to perform a crude analysis by searching for potentially unsafe library function calls. ITS4 is a lexical analysis tool that searches for security problems using a database of potentially dangerous constructs [VBKM00]. Lexical analysis techniques are fast and simple, but their power is very limited since they do not take into account the syntax or semantics of the program.

More precise checking requires a deeper analysis of the program. Our work builds upon considerable work on constraint-based analysis techniques. We do not attempt to summarize foundational work here. For a summary see [Aiken99].

Proof-carrying code [NL 96, Necula97] is a technique where a proof is distributed with an executable and a verifier checks the proof guarantees the executable has certain properties. Proof-carrying code has been used to enforce safety policies constraining readable and writeable memory locations. Automatic construction of proofs of memory safety for programs written in an unsafe language, however, is beyond current capabilities.

Wagner, et al. have developed a system to statically detect buffer overflows in C [WFBA00, Wagner00]. They used their tool effectively to find both known and unknown buffer overflow vulnerabilities in a version of `sendmail`. Their approach formulates the problem as an integer range analysis problem by treating C strings as an abstract type accessed through library functions and modeling pointers as integer ranges for allocated size

and length. A consequence of modeling strings as an abstract data type is that buffer overflows involving non-character buffers cannot be detected. Their system generates constraints similar to those generated by LCLint for operations involving strings. These constraints are not generated from annotations, but constraints for standard library functions are built in to the tool. Flow insensitive analysis is used to resolve the constraints. Without the localization provided by annotations, it was believed that flow sensitive analyses would not scale well enough to handle real programs. Flow insensitive analysis is less accurate and does not allow special handling of loops or `if` statements.

Dor, Rodeh and Sagiv have developed a system that detects unsafe string operations in C programs [DRS01]. Their system performs a source-to-source transformation that instruments a program with additional variables that describe string attributes and contains `assert` statements that check for unsafe string operations. The instrumented program is then analyzed statically using integer analysis to determine possible assertion failures. This approach can handle many complex properties such as overlapping pointers. However, in the worst case the number of variables in the instrumented program is quadratic in the number of variables in the original program. To date, it has only been used on small example programs.

A few tools have been developed to detect array bounds errors in languages other than C. John McHugh developed a verification system that detects array bounds errors in the Gypsy language [McHugh84]. Extended Static Checking uses an automatic theorem-prover to detect array index bounds errors in Modula-3 and Java [DLNS98]. Extended Static Checking uses information in annotations to assist checking. Detecting array bounds errors in C programs is harder than for Modula-3 or Java, since those languages do not provide pointer arithmetic.

8. Conclusions

We have presented a lightweight static analysis tool for detecting buffer overflow vulnerabilities. It is neither sound nor complete; hence, it misses some vulnerabilities and produces some spurious warnings. Despite this, our experience so far indicates that it is useful. We were able to find both known and previously unknown buffer overflow vulnerabilities in `wu-ftpd` and `BIND` with a reasonable amount of effort using our approach. Further, the process of adding annotations is a constructive and useful step for understanding of a program and improving its maintainability.

We believe it is realistic (albeit perhaps optimistic) to believe programmers would be willing to add annotations to their programs if they are used to efficiently and clearly detect likely buffer overflow vulnerabilities (and other bugs) in their programs. An informal sampling of tens of thousands of emails received from LCLint users indicates that about one quarter of LCLint users add the annotations supported by previously released versions of LCLint to their programs. Perhaps half of those use annotations in sophisticated ways (and occasionally in ways the authors never imagined). Although the annotations required for effectively detecting buffer overflow vulnerabilities are somewhat more complicated, they are only an incremental step beyond previous annotations. In most cases, and certainly for security-sensitive programs, the benefits of doing so should far outweigh the effort required.

These techniques, and static checking in general, will not provide the complete solution to the buffer overflow problem. We are optimistic, though, that this work represents a step towards that goal.

Availability

LCLint source code and binaries for several platforms are available from <http://lclint.cs.virginia.edu>.

Acknowledgements

We would like to thank the NASA Langley Research Center for supporting this work. David Evans is also supported by an NSF CAREER Award. We thank John Knight, John McHugh, Chenxi Wang, Joel Winstead and the anonymous reviewers for their helpful and insightful comments.

References

- [Aiken99] Alexander Aiken. *Introduction to Set Constraint-Based Program Analysis*. Science of Computer Programming, Volume 35, Numbers 2-3. November 1999.
- [AlephOne96] Aleph One. *Smashing the Stack for Fun and Profit*. BugTraq Archives. <http://immunix.org/StackGuard/profit.html>.
- [Andrews01] Mark Andrews. Personal communication, May 2001.
- [BST00] Arash Baratloo, Navjot Singh and Timothy Tsai. *Transparent Run-Time Defense Against Stack-Smashing Attacks*. 9th USENIX Security Symposium, August 2000.
- [Collins88] William J. Collins. *The Trouble with For-Loop Invariants*. 19th SIGCSE Technical Symposium on Computer Science Education, February 1988.
- [Coolbaugh99] Liz Coolbaugh. *Buffer Overflow Protection from Kernel Patches*. Linux Weekly News, <http://lwn.net/1999/1230/security.php3>.
- [Cousot77] Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. Fourth ACM Symposium on Principles of Programming Languages, January 1977.
- [CPMH+98] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. *Automatic Detection and Prevention of Buffer-Overflow Attacks*. 7th USENIX Security Symposium, January 1998.
- [CBDP+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen. *Protecting Systems from Stack Smashing Attacks with StackGuard*. Linux Expo. May 1999. (Updated statistics at <http://immunix.org/StackGuard/performance.html>)
- [CWPBW00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie and Jonathan Walpole. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. DARPA Information Survivability Conference and Exposition. January 2000.
- [DLNS98] David Detlefs, K. Rustan M. Leino, Greg Nelson and James B. Saxe. *Extended Static Checking*. Research Report, Compaq Systems Research Center. December 18, 1998.
- [DRS01] Nurit Dor, Michael Rodeh and Mooly Sagiv. *Cleanness Checking of String Manipulations in C Programs via Integer Analysis*. 8th International Static Analysis Symposium. To appear, July 2001.
- [ES99] Úlfar Erlingsson and Fred B. Schneider. *SASI Enforcement of Security Policies: A Retrospective*. New Security Paradigms Workshop. September 1999.
- [ES00] Úlfar Erlingsson and Fred B. Schneider. *IRM Enforcement of Java Stack Inspection*. IEEE Symposium on Security and Privacy. May 2000.
- [ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold and David Notkin. *Dynamically Discovering Likely Program Invariants to Support Program Evolution*. International Conference on Software Engineering. May 1999.

- [EGHT94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. SIGSOFT Symposium on the Foundations of Software Engineering. December 1994.
- [Evans96] David Evans. *Static Detection of Dynamic Memory Errors*. SIGPLAN Conference on Programming Language Design and Implementation. May 1996.
- [ET99] David Evans and Andrew Twyman. *Flexible Policy-Directed Code Safety*. IEEE Symposium on Security and Privacy. May 1999.
- [Evans00a] David Evans. *Policy-Directed Code Safety*. MIT PhD Thesis. February 2000.
- [Evans00b] David Evans. *Annotation-Assisted Lightweight Static Checking*. First International Workshop on Automated Program Analysis, Testing and Verification. June 2000.
- [Evans00c] David Evans. *LCLint User's Guide, Version 2.5*. May 2000.
<http://lclint.cs.virginia.edu/guide/>
- [FBF99] Timothy Fraser, Lee Badger and Mark Feldman. *Hardening COTS Software with Generic Software Wrappers*. IEEE Symposium on Security and Privacy. May 1999.
- [GWTB96] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer. *A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker*. 6th USENIX Security Symposium. July 1996.
- [GH93] John V. Guttag and James J. Horning, editors, with Stephen J. Garland, Kevin D. Jones, Andrés Modet and Jennette M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag. 1993.
- [IS97] A. Ireland and J. Stark. *On the Automatic Discovery of Loop Invariants*. 4th NASA Langley Formal Methods Workshop. September 1997.
- [ISO99] ISO/IEC 9899 International Standard. *Programming Languages – C*. December 1999. Approved by ANSI May 2000.
- [LHSS00] David Larochelle, Yanlin Huang, Avneesh Saxena and Seejo Sebastine. *Static Detection of Buffer Overflows in C using LCLint*. Unpublished report available from the authors. May 2000.
- [Luckin01] Bob Luckin. Personal communication, April 2001.
- [Lundberg01] Gregory A Lundberg. Personal communication, April 2001.
- [McHugh84] John McHugh. *Towards the Generation of Efficient Code from Verified Programs*. Technical Report 40, Institute for Computing Science, University of Texas at Austin PhD Thesis, 1984.
- [Necula97] George C. Necula. *Proof-Carrying Code*. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1997.
- [NL96] George C. Necula and Peter Lee. *Safe Kernel Extensions Without Run-Time Checking*. 2nd Symposium on Operating Systems Design and Implementation, October 1996.
- [Orcero00] David Santo Orcero. *The Code Analyzer LCLint*. Linux Journal. May 2000.
- [Pethia00] Richard D. Pethia. *Bugs in Programs*. Keynote address at SIGSOFT Foundations of Software Engineering. November 2000.
- [PG00] Pramode C E and Gopakumar C E. *Static Checking of C programs with LCLint*. Linux Gazette Issue 51. March 2000.
- [RE89] Jon Rochlis and Mark Eichin. *With Microscope and Tweezers: the Worm from MIT's Perspective*. Communications of the ACM. June 1989.
- [Snow99] Brian Snow. *Future of Security*. Panel presentation at IEEE Security and Privacy. May 1999.
- [Spafford88] Eugene Spafford. *The Internet Worm Program: An Analysis*. Purdue Tech Report 832. 1988.
- [SI98] J. Stark and A. Ireland. *Invariant Discovery Via Failed Proof Attempts*. 8th International Workshop on Logic Based Program Synthesis and Transformation. June 1998.
- [Torvalds98] Linus Torvalds. Message archived in *Linux Weekly News*. August 1998.
<http://lwn.net/980806/a/linus-noexec.html>
- [VBKM00] John Viega, J.T. Bloch, Tadayoshi Kohno and Gary McGraw. *ITS4 : A Static Vulnerability Scanner for C and C++ Code*. Annual Computer Security Applications Conference. December 2000.
- [WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer and Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*. Network and Distributed System Security Symposium. February 2000.
- [Wagner00] David Wagner. *Static Analysis and Computer Security: New Techniques for Software*

Assurance. University of California, Berkeley, PhD Thesis, 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. *Efficient Software-Based Fault Isolation*. 14th ACM Symposium on Operating Systems Principles, 1993.

[Wegbreit75] Ben Wegbreit. *Property Extraction in Well-Founded Property Sets*. IEEE Transactions on Software Engineering, September 1975.

[WSJ01] The Wall Street Journal. *Researchers Find Software Flaw Giving Hackers Key to Web Sites*. January 30, 2001.

not guarantee that a null character is added; `strncat` appends `n` characters to the buffer and a null character. The ensures clauses reveal these differences clearly.

The full specifications for `malloc` and `calloc` also include `null` annotations on the result that indicate that they may return `NULL`. Existing LCLint checking detects dereferencing a potentially null pointer. As a result, the implicit actual postcondition for `malloc` is `maxSet(result) == size ∨ result == null`. LCLint does not support general disjunctions, but possibly `NULL` values can be handled straightforwardly.

A. Annotated Selected C Library Functions

```
char *strcpy (char *s1, char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead (s2)
  /\ result == s1@*/;

char *strncpy (char *s1, char *s2,
              size_t n)
/*@requires maxSet(s1) >= n - 1@*/
/*@ensures maxRead (s1) <= maxRead(s2)
  /\ maxRead (s1) <= (n - 1)
  /\ result == s1@*/;

char *strcat (char *s1, char *s2)
/*@requires maxSet(s1)
  >= (maxRead(s1)
    + maxRead(s2))@*/
/*@ensures
  maxRead(s1) == maxRead(s1)
    + maxRead(s2)
  /\ result == s1@*/;

strncat (char *s1, char *s2, int n)
/*@requires maxSet(s1)
  >= maxRead(s1) + n@*/
/*@ensures maxRead(result)
  >= maxRead(s1) + n@*/;

extern size_t strlen (char *s)
/*@ensures result == maxRead(s)@*/;

void *calloc (size_t nobj, size_t size)
/*@ensures maxSet(result) == nobj@*/;

void *malloc (size_t size)
/*@ensures maxSet(result) == size@*/;
```

These annotations were determined based on ISO C standard [ISO99]. Note that the semantics of `strncpy` and `strncat` are different – `strncpy` writes exactly `n` characters to the buffer but does