Automatically Fixing C Buffer Overflows Using Program Transformations

Alex Shaw
Auburn University
Auburn, AL, USA
ajs0031@auburn.edu

Dusten Doggett
Auburn University
Auburn, AL, USA
djd0001@auburn.edu

Munawar Hafiz
Auburn University
Auburn, AL, USA
munawar@auburn.edu

***Abstract*—Fixing C buffer overflows at source code level remains a manual activity, at best semi-automated. We present an automated approach to fix buffer overflows by describing two program transformations that automatically introduce two well-known security solutions to C source code. The transformations embrace the difficulties of correctly analyzing and modifying C source code considering pointers and aliasing. They are effective: they fixed all buffer overflows featured in 4,505 programs of NIST's SAMATE reference dataset, making the changes automatically on over 2.3 million lines of code (MLOC). They are also safe: we applied them to make hundreds of changes on four open source programs (1.7 MLOC) without breaking the programs. Automated transformations such as these can be used by developers during coding, and by maintainers to fix problems in legacy code. They can be applied on a case by case basis, or as a batch to fix the root causes behind buffer overflows, thereby improving the dependability of systems.**

I. Introduction

Buffer overflow still remains as a critical problem to fix in dependable systems. Many research works have explored how to detect buffer overflow, both statically [36], [38], [62], [63] and dynamically [11], [12]. Yet, buffer overflow vulnerabilities have not been eliminated. In 2012, National Vulnerability Database [59] published reports on 5,297 security vulnerabilities. Of them, 843 (15.91%) were overflows [50].

Existing prevention approaches do not fix overflows at source code level. Some dynamic approaches fix overflow by making the stack non-executable [49], checking array bounds [29], [35], or checking integrity of code pointers [12], [17], but all of these solutions are introduced transparently at run time. Moreover, these approaches suffer from performance overhead and the solutions can be circumvented [6], [64]. These approaches do not help developers produce better code. It is very easy to write C code with buffer overflows. Existing approaches at best detect or fix buffer overflows, but they cannot guide developers to understand and write secure code.

A more effective and didactic solution to fix buffer overflows is to introduce the solution at C source code level, thus teaching the developers as their problems are being fixed. CCured [8] and Cyclone [34] provides solutions to fix buffer overflows by using annotations that extend pure C, but these annotations have to be introduced manually in the source code. Other approaches [14], [40] to introduce fixes automatically at source code level are not safe to use, since they are supported by minimal or non-existent program analysis, and therefore often break the original program with the fix.

We describe a program transformation based approach—specifically two transformations that fix buffer overflows in C programs by automatically introducing two frequently prescribed security solutions. Experts have identified that some library functions in C are potentially unsafe and should not be used in practice. Our SAFE LIBRARY REPLACEMENT (SLR) transformation replaces unsafe library functions in a C program with safer alternatives. Experts also suggest that any pointer arithmetic should keep track of the buffer bounds at source code level. However, introducing this solution in legacy code requires a lot of manual effort. Our SAFE TYPE REPLACEMENT (STR) transformation replaces a character buffer with a safe data structure that keeps track of the length information which it uses to check buffer bounds during pointer operations. They are similar to refactorings [20], but they do not intend to preserve behavior. They are instead security-oriented program transformations [24], [25], that improve the security of systems by preserving expected behavior but fixing the two most prominent root causes behind buffer overflows: the use of *unsafe library functions* and *bad pointer operations*.

A transformation based approach has several advantages.

- Developers must make small, frequent changes in source code to fix buffer overflows manually. People are bad at repetitive tasks—computers are better.
- A program transformation based solution is more likely to be adopted, especially if the transformations make small changes [60]. Developers can use the transformations during development, similar to how refactorings are used, or use them to produce security patches during maintenance.
- The source-level program transformations would help developers better understand and be aware of the subtleties of buffer overflow vulnerabilities.
- Most importantly, the transformations can be applied on a case by case basis to fix a specific overflow, or as a batch to automatically fix root causes in large legacy systems, improving their dependability in the process.

Automatically transforming C programs to remove buffer overflows at source code level is challenging. Many of the safe function choices and the safe data structures need length information about buffers. It is hard to determine the length of a buffer using static analysis; pointers and aliasing further complicate the analysis. Program analyses that are applicable for compilers are not useful for program transformations, since the program transformations need to keep track of source code. For example, approaches to perform alias analysis for compilers transform source code to SSA form; this is not useful for program transformations since the results will be hard to map back to source code. None of the existing program transformation infrastructures for C support sophisticated static analyses at C source code level, e.g., Eclipse CDT only supports name binding, type analysis, and limited control flow analysis. Without data flow analysis, it is impossible to implement any non-trivial program transformation.

This paper describes the design decisions, the program analyses, and the mechanism to analyze and transform C code at source code level to fix buffer overflows. We implemented the transformations using OpenRefactory/C [27], an infrastructure for developing C program transformations. We validated the transformations by applying them automatically on benchmark programs and open source software. We demonstrated that the two program transformations are sufficient to fix all possible C buffer overflows originating from unsafe functions and bad pointer operations by automatically applying them to remove buffer overflows from 4,505 benchmark programs of NIST's SAMATE reference dataset [48] (Section IV-A). The program transformations ran on more than 2.3 million lines of code (MLOC). The SAMATE programs have a function showing normal behavior and another function showing problem behavior. In all cases, our transformations preserved normal behavior, and modified behavior resulting from overflows.

To demonstrate that our program transformations do not break existing code, we automatically applied the three transformations on all potential targets in four open source programs: libpng, zlib, GMP, and LibTIFF (Section IV-B). SLR was applied on all targeted unsafe functions, STR on all local character pointer variables. In total, SLR was applied on 317 function calls; it modified 259 function call expressions (average 81.7% in 4 programs). STR modified all 237 pointer variables that passed its preconditions (100%), overall 80.01% of all local variables. Even after these changes, the modified programs had minimal performance overhead (Section IV-C).

This paper makes the following contributions.

- It describes two source-to-source behavior-enhancing program transformations—the design decisions behind the transformations (Section II) and the mechanism of the transformations (Section III) including the program analyses to support the transformations (Section III-A).
- It demonstrates that complex, yet accurate, source-level C program transformations can be implemented as part of the refactoring catalog of popular IDEs.
- It shows that the two program transformations can prevent all types of buffer overflow originating from unsafe library functions and bad pointer operations (Section IV-A). It also tests the accuracy of the transformations by applying them automatically to make many small modifications to open source programs in a way that produces programs that maintain functionality (Section IV-B) and have minimal overhead (Section IV-C).

More details and results are available at our web page: https://munawarhafiz.com/research/overflow.

II. Program Transformations to Fix Overflows

We concentrate on the two most common root causes of buffer overflow. First, the C library includes several unsafe functions that may lead to buffer overflows. For example, `strcpy` (or `strcat`) copies (or concatenates) a source buffer to a destination buffer, but does not put a restriction that the copy (concatenation) should not overflow the destination buffer. Second, programmers often make mistakes in pointer operations, specifically pointer arithmetic; these mistakes lead to buffer overflows.

Avoiding unsafe functions and keeping track of buffer length are well-known solutions but are often not followed in legacy code. Empirical studies have shown that developers manually make these changes, but the manual process does not scale for large programs [25]. This paper describes how these prescribed solutions can be introduced as fixes at source code level using program transformations.

We introduce two program transformations. One replaces unsafe library functions in C programs with safe alternatives (Safe Library Replacement, Section II-A). The other replaces character pointers with safe data structures that keep track of buffer length and available memory information during pointer operations (Safe Type Replacement, Section II-B). Our transformations make structural changes to source code using sophisticated program analysis. They modify source code so that it conforms to a safe C programming style [56].

A. *Safe Library Replacement (SLR)*

You have a program that uses a library function that may cause data injection attacks if it receives an insufficiently validated input. You want to ensure that the program is not vulnerable to injection attacks.

Replace unsafe functions with safe functions that are not vulnerable even if malicious data is injected.

1) Motivation: C library functions that do not perform extra checking during buffer operations are vulnerable to buffer overflow. Table I shows some of these unsafe functions. Safer alternatives (right column of Table I) have been introduced by researchers. Some alternatives force developers to explicitly mention the number of bytes that can be written to the buffer (functions in *glib* library); others use a different C string data type that can be resized at runtime (functions in *libmib* library). However, the unsafe functions are still used frequently. Moreover, there is a lot of legacy C code that contains these.

We surveyed the developers of the top ten most active projects of all time in SourceForge about their development approach. Six projects used C/C++, three used PHP and one used Java. In five out of six C/C++ projects, programmers initially used unsafe `strcpy` and `strcat` functions, but manually changed to safer C/C++ string libraries later.

Manual changes are error-prone. This method does not scale for large projects. For example, Ghostscript (about 800 KLOC) is a medium size program, but its programmers have replaced only a few of its unsafe functions (109 of 318, 34.3% of the `strcpy` functions; 6 of 172, 3.5% of `strcat`). We asked the developers the reason for this. They said that manually changing the functions are difficult; hence they made changes only to those functions that they thought were vulnerable. This is clearly a dangerous practice for dependable systems.

Unsafe Library Functions	**Safe Alternative Functions**

`strcpy`(3), `strncpy`(3) - Copy string
(3) indicates that documentation is available in Linux man section 3
`char *strcpy (char *dst, const char *src);`
`char *strncpy (char *dst, const char *src, size_t num);`
	`g_strlcpy` from glib library [45]
	`gsize g_strlcpy(gchar *dst, const gchar *src, gsize dst_size);`
	`astrcpy`, `astrn0cpy` from libmib library [19]
	`char *astrcpy(char **dst_address, const char *src);`
	`strcpy_s` from ISO/IEC 24731 [31] and SafeCRT library [41]
	`StringCchCopy`, `StringCchCopyN` from StrSafe [44] library
	`safestr_copy` and `safestr_ncopy` from Safestr library [43]

`memcpy`(3) - Copy memory area
`void *memcpy(void *dst, const void *src, size_t num);`
	`memcpy_s` from ISO/IEC 24731 [31]
	`errno_t memcpy_s(void *dst, size_t dst_size,`
	`const void *src, size_t num);`

`gets`(3) - Get input from stdin
`char *gets(char *dst);`
	`gets_s` from ISO/IEC 24731 [31], `fgets` from C99 [32]
	`char *gets_s(char *destination, size_t dest_size);`
	`char *fgets(char *dst, int dst_size, FILE *stream);`
	`afgets` from libmib library [19], `gets_s` from SafeCRT library [41]

`getenv(3) - Get value of an environment variable`
`char *getenv(char *dst);`
	`getenv_s` function [41]
	`errno_t getenv_s(size_t *return-value,`
	`char *dst, size_t dst_size, const char *name);`

`sprintf(3), snprintf(3) - Print string`
`char *sprintf(char *str,const char *format, ...);`
	`g_snprintf` from glib library [45]
	`asprintf` from libmib library [19]
	`sprintf_s` from ISO/IEC 24731 [31] and SafeCRT [41]
	`gint g_snprintf (gchar *string, gulong n,`
	`gchar const *format, ...);`
	`int asprintf (char *ppaz, const char *format, ...);`

TABLE I. SOME UNSAFE FUNCTIONS AND THEIR SAFER ALTERNATIVES

2) Precondition: A developer selects a function call expression and invokes the SAFE LIBRARY REPLACEMENT transformation. The following preconditions are checked:

- The function is one of the unsafe library functions supported by the transformation.
- The size of the buffer being written to can be calculated by applying control flow, data flow, and alias analysis. If the size of the buffer cannot be determined by our program analysis, the precondition will not be met.

3) Solution: Replace an unsafe function call with a safer alternative. SLR uses safer alternatives that check and truncate inputs to match the size of destination buffers, similar to *glib* functions. These functions are syntactically similar to the original functions. They do not require new data types like the *libmib* functions; therefore, the changes will be minimal per instance. The transformation has to analyze the program and calculate the exact size of the destination buffer to introduce as the extra parameter. This analysis is complicated by pointers and aliasing in C.

If the size of the buffer can be calculated, the function call is renamed with the new size parameter. Other additional steps may need to be taken for some function replacements, e.g., `fgets`, the replacement option for `gets`, requires a `FILE *` parameter (see Section III-B2 and Table I).

4) Example: Consider the following example, which demonstrates a buffer overflow from the use of `strcpy`:

```
1  char buf[10];
2  char src[100];
3  memset(src,'c', 50);
4  src[50] = '\0';
5  char *dst = buf;
6  strcpy(dst, src);
```

In the code, a one hundred byte buffer (`src`) and a ten byte buffer (`buf`) are instantiated. `src` is initialized to a fifty byte C string (line 3-4). A pointer (`dst`) is then set to the first byte of `buf` (line 5). The `strcpy` function is then called to copy the C string in `src` to the buffer pointed by `dst` (line 6). Fifty bytes will be written from where `dst` starts, which will overflow `buf`.

SLR first calculates the type of `dst`, which is a pointer. It will then find the most recent definition of `dst`, which is an assignment to `buf` (line 5). It will then recursively determine the size of `buf`, which results in array type (described in detail in Algorithm 1, Section III-B). The transformation then changes the name of the function call to `g_strlcpy` in Linux systems (or `strlcpy` in Mac systems), and adds the **`sizeof`** keyword on `buf` as the new parameter, as shown below.

```
...
6  g_strlcpy(dst, src, sizeof(buf));
```

The **`sizeof`** keyword on `buf` will return 10, which causes `g_strlcpy` to only write ten bytes starting from `dst`. The buffer overflow is removed.

B. Safe Type Replacement (STR)

You have a program in which character pointers are used; the pointers may be used in an unsafe manner, specifically in an arithmetic expression, which could overflow a buffer.

Change character pointers to a new data type which contains the length and the allocated size of the string it represents. Add explicit checks for buffer bounds before any buffer operation.

1) Motivation: The lack of bounds checking for **`char`** pointers in C may lead to overflows when they are used in certain expressions. Pointer arithmetic can be particularly dangerous, especially with the prevalence of logical flaws such as off-by-one errors. Standard C compilers cannot detect unsafe or incorrect usages of character pointers to prevent this error.

2) Precondition: A developer selects a **char** pointer or array and invokes the SAFE TYPE REPLACEMENT transformation. The following preconditions are checked:

- The variable is a **char** pointer or an array.
- The variable is locally declared. STR cannot be applied on a global variable, a function parameter, or a member of a user defined **struct**. We chose not to apply the transformation on these types of expressions to avoid making changes in external files.
- The variable is not used in an unsupported C library function (most common string functions in C library are supported).

3) Mechanism: Replace **char** pointer with a new data structure. For our implementation, all of the character pointers that are identified by the STR are changed to a new data structure called `stralloc`, which is defined as follows:

```
typedef struct stralloc {
    char* s;
    char* f;
    unsigned int len;
    unsigned int a;
}
```

Our implementation of `stralloc` is a modified version of the `stralloc` data structure from qmail [52]. The data structure stores a character pointer `s`, which is equivalent to the **char** pointer it replaces; the data structure stores additional information about the string represented by the **char** pointer. `f` is always set to point to the base of the original `s`; so it can be used for bounds checking even after `s` has changed. The variable `len` represents the length of the string, and `a` represents the number of bytes currently used. Upon initialization, the `stralloc` library appropriately allocates enough memory for the string being stored. The `stralloc` library also contains a number of functions designed to replace common C library functions.

STR replaces all expressions that use the target pointers and replaces the use instances following specific patterns. Table II gives some examples of some of the most common replacement patterns STR.

4) Example: Consider this program from SAMATE reference dataset [48] having buffer over-read problem (CWE-126).

```
1  char* data;
2  char dest[100];
3  memset(dest, 'C', 100);
4  data[100] = dest[100];
```

Since `data` is not explicitly allocated, the program may access memory locations to which it should not have access, potentially causing buffer overflow. STR fixes this by transforming **char** * to a safe data structure.

```
1  stralloc *data, *dest;
2  stralloc ssss_data={0,0,0}, sss_dest = {0,0,0};
3  data = &ssss_data;
4  dest = &ssss_dest;
5  stralloc_memset(dest,'c',100);
6  stralloc_dereference_replace_by(data,
           stralloc_get_derefenced_char_at(dest,100),100,0);
```

Lines 1-4 show the declaration of the `stralloc` variables. Line 5 shows the initialization process. Finally, the assignment expression in Line 4 of the original program is replaced by another `stralloc` library function. The first argument specifies the `stralloc` pointer on the left hand side of the original expression. The second argument calls another `stralloc` library function to get the character represented by the right hand side of the assignment. The third argument specifies the index of the character to replace, and the last specifies that the size of the data pointer should not be changed. STR follows the replacement patterns (Table II) to automatically make changes.

TABLE II. TRANSFORMING COMMON EXPRESSIONS

Initial Code	Replacement Pattern
Declaration and Reference	
1. Identifier expression `buf`	No change necessary `buf`
2. Declaration statement `char* buf;`	Stralloc declaration statement `stralloc* buf;` `stralloc ssss_buf = {0,0,0};` `buf = &ssss_buf;`
3. Allocation of buffer `buf = malloc(1024)`	Assign/allocate member variables `buf->s = malloc(1024)` `buf->a = 1024`
Assignment Expression	
4. Assignment to null or (void*)0 `buf = null`	No change necessary `buf = null`
5. Assignment to other buffer `buf1 = buf2`	No change necessary `buf1 = buf2`
6. Assignment to string literal `buf = "text"`	Stralloc library function `stralloc_copybuf(buf,"text", strlen("text"))`
7. Assignment to cast expression `buf = (char*)(exp)`	Analyze rhs, replace with library function `stralloc_copybuf(buf,(char*) exp,sizeof((char*)exp)))`
Arithmetic and Binary Expressions	
8. Increment expression `buf++`	Stralloc library function `stralloc_increment_by(buf,1)`
9. Decrement expression `buf -= 3`	Stralloc library function `stralloc_decrement_by(buf,3)`
10. Binary expression `sizeof(buf) < 3`	Replace subexpressions `buf->a < 3`
Array Access and Dereference Expressions	
11. Array access expression `buf[1]`	Stralloc library function `stralloc_get_dereferenced_ char_at(buf,1)`
12. Assignment to an array element `buf[1] = 'b'`	Stralloc library function `stralloc_dereference_replace _by(buf,1,'b')`
13. Assigning one array element to another `buf1[0] = buf2[0]`	Stralloc library function `stralloc_dereference_replace _by(buf1,0,stralloc_get_ dereferenced_char_at(buf2,0))`
14. Dereference assignment statement `*(buf+4) = 'a'`	Stralloc library function `stralloc_dereference_replace _by(buf,4,'a')`
15. Dereferenced assignment to binary expression `*(buf+1) = 'a' + 'b'`	Stralloc library function `stralloc_dereference_replace _by(buf,1,'a'+'b')`
Argument in Function Call Expression	
16. Argument in C library function `strlen(buf)`	Function dependent `buf->len`
17. Argument in user defined function `foo(buf)`	Examine function - replace if safe `foo(buf->s)`
Conditional or Iteration Statement	
18. Conditional/Iteration statement `if(buf[0] == 'a')`	Examine and replace expression `if(stralloc_get_dereferenced _char_at(buf,0) == 'a')`

III. MECHANISM OF THE TRANSFORMATIONS

A. Program Analyses to Support the Transformations

We implemented the program transformations on OpenRefactory/C [27], a framework for building transformations for C programs. OpenRefactory/C supports name binding analysis,

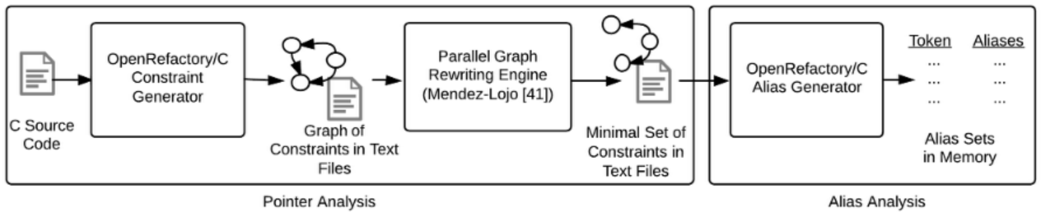
C Source Code
OpenRefactory/C Constraint Generator
Graph of Constraints in Text Files
Parallel Graph Rewriting Engine (Mendez-Lojo [41])
Minimal Set of Constraints in Text Files
Pointer Analysis
OpenRefactory/C Alias Generator
Token
Aliases
Alias Sets in Memory
Alias Analysis

Fig. 1. Components for Performing Pointer and Alias Analysis

type analysis, control flow analysis, and static call graph analysis of C programs. We extended OpenRefactory/C to add reaching definition analysis, points-to analysis, control and data dependence analysis, and alias analysis to support the program transformations. Here we describe the mechanism of points-to and alias analysis.

Figure 1 shows the main components of points-to analysis. Our analysis is based on the C++ pointer analysis algorithm by Hardekopf [28] but is performed at source code level. It is an intra-procedural, flow-insensitive, inclusion-based analysis. Our constraint generator component traverses the abstract syntax tree of a program and produces a graph that contains the variables in a C program as nodes; the edges between nodes indicate that one variable points to another variable. It does not perform shape analysis on arrays and structures; it describes them as aggregate nodes. Hardekopf's algorithm optimally rewrites the graph based on a few constraints. The graph rewriting is done using the parallel graph rewriting engine Galois [51] following Mendez-Lojo's [42] approach. Finally, the alias generator component topologically sorts the points-to graphs and calculates the alias sets. It is possible for an aggregate node in a points-to graph (e.g., a structure) to point to itself. These recursive cycles are irrelevant to aliases, so they are ignored. There should not be any other cycles in the points-to graph after the graph rewriting. The alias sets are stored in a hash map in memory for efficient access.

The alias set is used for the reaching definition analysis. Reaching definition and control and data dependence analysis algorithms follow traditional worklist based algorithms.

B. Mechanism of Safe Library Replacement Transformation

The common first step for SLR is to statically determine the size of the buffer written to. The size of a statically allocated buffer can be determined by the `sizeof` function. On the other hand, the size of a dynamically allocated buffer can be determined by the `malloc_usable_size` function. However, arbitrarily using a function is not allowed, e.g., applying the `malloc_usable_size` on a statically allocated buffer will result in a segmentation fault.

The static analysis to determine the size of the buffer has to analyze different C expressions that can represent a buffer (e.g., the destination buffer parameter in `strcpy`). Algorithm 1 describes the mechanism. If the destination parameter contains an assignment expression, the size of the right hand side of the assignment expression is recursively calculated (Lines 2-4). If the parameter is an array access expression, the size of the array identifier is calculated using the `sizeof` function (Lines 5-7). If the parameter is a binary expression, it will have either addition or subtraction operation on a buffer with

Algorithm 1 Calculate length of a buffer expression

```
 1: function GETBUFFERLENGTH(B: expression for destination buffer)
 2:     if B is assignment expression then
 3:         B ← RHS(B)
 4:         return GETBUFFERLENGTH(B)
 5:     else if B is array access expression then
 6:         B ← GETARRAYIDENTIFIER(B)
 7:         return SIZEOF(B)
 8:     else if B is pointer arithmetic binary expression then
 9:         op ← GETOPERATOR(B)
10:         if op is + or - then
11:             newop ← + for -, or - for +
12:             numPart ← GETNUMERICPART(B)
13:             B ← GETBUFFERPART(B)
14:             return GETBUFFERLENGTH(B) newOp numPart
15:         return
16:     else if B is prefix expression then
17:         if GETPREFIXOPERATOR(B) is ++ then
18:             return GETBUFFERLENGTH(B) - 1
19:         else if GETPREFIXOPERATOR(B) is −− then
20:             return GETBUFFERLENGTH(B) + 1
21:     else if B is cast expression then
22:         return GETBUFFERLENGTH(GETEXPRESSION(B))
23:     else if B is identifier expression then
24:         if TYPE(B) is ArrayType then                    ▷ Type Analysis
25:             return SIZEOF(B)
26:         else if TYPE(B) is PointerType then             ▷ Type Analysis
27:             if ISALIASED(B) then                        ▷ Alias Analysis
28:                 return
29:             else
30:                 def ← definition reaching B        ▷ Reaching Definition
31:                 if def contains heap allocation then
32:                     return MALLOC_USABLE_SIZE(B)
33:                 else if def is assignment then
34:                     return GETBUFFERLENGTH(RHS(def))
35:     else if B is element access expression then
36:         if TYPE(B) is ArrayType then                    ▷ Type Analysis
37:             return SIZEOF(B)
38:         else if TYPE(B) is PointerType then             ▷ Type Analysis
39:             if ISALIASED(B) then                        ▷ Alias Analysis
40:                 return
41:             else
42:                 def ← definition reaching B        ▷ Reaching Definition
43:                 struct ← GETSTRUCT(B)
44:                 defStruct ← definition of struct reaching B
45:                 if defStruct is in the control flow path from def to B then
46:                     return                        ▷ Control Flow Analysis
47:                 else if def contains heap allocation then
48:                     return MALLOC_USABLE_SIZE(B)
49:                 else if def is assignment then
50:                     return GETBUFFERLENGTH(RHS(def))
```

some numeric value. The size is evaluated based on the size of the buffer, the kind of arithmetic operator, and the numeric value (Lines 10-15). If the parameter is a prefix expression, specifically an increment or decrement expression, the size of the parameter is computed by recursively computing the size of the buffer and making appropriate correction for the arithmetic (Lines 16-20). If the buffer is a cast expression, the size of the expression being cast is recursively computed (Lines 21-22).

Most commonly, the parameter is an identifier for a buffer. If the buffer is statically allocated, the **`sizeof`** function is used to calculate the size (Lines 24-25). If the buffer is a pointer and it is not aliased, the definition reaching the buffer is calculated (Lines 26-30). If a function call to `malloc` family is in that definition, the buffer is heap-allocated and its size is calculated using `malloc_usable_size` (Lines 31-32). For other assignment expressions, the algorithm recursively computes the size of the buffer on the right side of the assignment (Lines 33-34, following similar analysis in Lines 2-4).

Calculating the size of a buffer described by a structure element access expression is similar (Lines 35-50). The only difference is that the algorithm checks to see if the entire struct is redefined between the most recent definition of the element and the unsafe function call (Lines 42-45). Since we represent a structure as an aggregate object during alias analysis, the definition of the structure element reaching the destination buffer may be different from the definition of the entire structure. If the entire struct is redefined, then the element is redefined as well. The algorithm will not calculate size if this is found (Lines 45-46).

SLR replaces six unsafe functions: `strcpy`, `strcat`, `sprintf`, `vsprintf`, `memcpy`, and `gets`. The mechanism for replacing `strcpy`, `strcat`, `sprintf`, and `vsprintf` are similar, so they are described together; these are applicable to Linux programs. SLR for `memcpy` and `gets` applies to both Linux and Windows programs.

1) Replacing `strcpy`*,* `strcat`*,* `sprintf`*, and* `vsprintf`*:* We used safe alternatives from the *glib* library for these functions: `g_strcpy` replaces `strcpy`, `g_strlcat` replaces `strcat`, `g_snprintf` replaces `sprintf`, and `g_vsnprintf` replaces `vsprintf`. The replacement follows three steps:

(1) Determine the size of the buffer following Algorithm 1.
(2) Change the name of the function and add the size of the buffer as a new parameter to the function call.
(3) Add appropriate header or provide information of the new library at link time.

Consider the `strcat` function in libpng version 1.2.6, minigzip.c file, line 275. SLR is applied on the `strcat` function.

```
266  char outfile[MAX_NAME_LEN];
     ...
275  strcat(outfile, GZ_SUFFIX);
      |
      | (After SLR)
      V
266  char outfile[MAX_NAME_LEN];
     ...
275  g_strlcat(outfile, GZ_SUFFIX, sizeof(outfile));
```

In the example, SLR analyzes the type and determines that `outfile` is an array. The size of an array can be determined by **`sizeof`**; it becomes the third parameter of the safe library function (line 275). Information about the library (`-lglib-2.0`) is added to Makefile, so that it is available at link time.

2) Replacing `gets`*:* `gets` is replaced with another standard library function, `fgets`. `fgets` takes two additional parameters: a length parameter and a `FILE` pointer to a stream to read from. The `gets` function always reads from standard input, so SAFE LIBRARY REPLACEMENT adds `stdin` as the stream.

`fgets` has different semantics. It includes a terminating newline character in the data being read, if there is one; `gets` never includes it. To match their semantics, a few extra lines of code are added after the replaced function call to remove the terminating newline character.

Consider the `gets` function used in a benchmark program of the SAMATE reference dataset.

```
char dest[DEST_SIZE];
char *result;
result = gets(dest);
       |
       | (After SLR)
       V
char dest[DEST_SIZE];
char *result;
result = fgets(dest, sizeof(dest), stdin);
char *check = strchr(dest, '\n');
if (check) {
   *check = '\0';
}
```

Here, SLR determines that `dest` is an array by using the type analysis. It introduces `fgets` as the alternative; the size is calculated by **`sizeof`** and the stream pointer parameter is `stdin`. The last four lines remove the newline character.

3) Replacing `memcpy`*:* The `memcpy` function is different from the other unsafe functions because it already takes a length parameter that specifies exactly how many bytes to write to the destination buffer. There may still be buffer overflow if the length parameter specifies a size larger than the size of the destination buffer. The transformation follows three steps:

(1) Determine the size of the destination buffer (Algorithm 1).
(2) Add a check if the specified number of bytes to be copied is larger than the size of the destination buffer. The smaller of the two values is the number of bytes that will be actually copied during runtime. This is introduced as a ternary expression as the parameter to `memcpy`.
(3) Terminate the destination buffer with `NULL`.

Consider the following:

```
35  size_t numlen;
...
48  num = __GMP_ALLOCATE_FUNC_TYPE (numlen+1, char);
49  memcpy (num, str, numlen);
      |
      | (After SLR)
      V
35  size_t numlen;
...
48  num = __GMP_ALLOCATE_FUNC_TYPE (numlen+1, char);
49  numlen = malloc_usable_size(num) > numlen ? numlen :
     malloc_usable_size(num);
50  memcpy (num, str, numlen);
```

In the example, from gmp v4.3.2, file mpq/set_str.c, line 49, SLR uses `malloc_usable_size` to calculate the length of `num` because it is a pointer that was set to the result of a heap allocation function, in this case a macro `__GMP_ALLOCATE_FUNC_TYPE` (line 48). SLR adds a new assignment expression before the `memcpy` that sets `numlen` to the length of the buffer, if the buffer's length is smaller than the value of `numlen`. The ternary expression prevents buffer overflow at runtime.

The second step is one of two options. If the existing length parameter is used in statements that are successors in control flow, the length parameter is explicitly assigned before the memcpy call. This may happen when the length parameter is used to null terminate the buffer. Otherwise, it is directly replaced with the ternary expression.

```
memcpy(dst, src, length)

---Option 1: to (length is used later)--->
length = dstLength > length ? length : dstLength
memcpy(dst, src, length)

---Option 2: to (length is not used)--->
memcpy(dst, src,
   dstLength > length ? length : dstLength)
```

C. Mechanism of Safe Type Replacement Transformation

Safe Type Replacement takes a source file and a **char** pointer or array selected by a user. It returns the transformed source file, or the original file if the preconditions are not met.

We use a modified version of the stralloc data structure and the library used in qmail [52] (Section II-B). Our implementation of STR is applicable for Linux programs, but Windows analogs can be implemented. The library contains functions designed to replace common C string operations and functions. Table II lists some of these functions.

The functions do not have one-to-one correspondence with C functions; Table II shows that some functions replace C expressions. Our implementation contains 18 functions. Among these are functions designed to initialize the stralloc variable, replace increment and decrement operations of C, find a substring or character at a given position in the string, and compare two strings contained by stralloc pointers. While these functions are relatively complex, a developer applying STR need not be concerned with the implementation as the transformation will automatically replace unsafe code with semantically equivalent safe code. The functions are named such that the developer can easily identify their purpose.

To maintain similar functionality as the **char** pointers, our implementation replaces the **char** pointers with stralloc pointers. Consider the following code:

```
char *src = "one", *dst = "two;
dst = src;
```

In this case, after both lines of code have executed, src and dst point to the same location in memory. In order to maintain this condition, it is not sufficient to use stralloc variables; we need stralloc pointers.

The following code is from zlib-1.2.5, minigzip.c file, transformed by applying STR.

```
300 char buf[1024];
301 char *infile;
...
319 infile = buf;
320 strcat(infile,".gz",strlen(".gz"));
     |
     | (After STR)
     V
300 stralloc *buf, *infile;
301 stralloc ssss_buf = {0,0,0}, ssss_infile = {0,0,0};
302 buf = &ssss_buf;
303 buf->a = 1024;
304 infile = &ssss_infile;
...
322 infile = buf;
323 stralloc_catbuf(infile,".gz",strlen(".gz"));
```

Two replacement patterns are demonstrated by this example. The first pattern is the assignment of one **char** pointer to another (Line 319 of original code). The new stralloc pointers should point to the same location in memory, so no change is needed (Line 321 of transformed code). Line 322 of transformed code shows a replacement pattern involving the strcat function (Line 320 of original). In this case, the function is simply replaced with a stralloc_catbuf function from the stralloc library.

For user defined functions, the stralloc library does not have analogs. In order to continue the transformation, we perform an inter-procedural analysis when a **char** pointer is used as an argument in a call to a user-defined function. It determines, at the call site, if the function call modifies the **char** pointer. In such a case, the transformation is not completed. The analysis is conservative; it may determine that a pointer is modified even when it is not. However, it is necessary to ensure that the transformation is safe and it does not change behavior. Empirical results in Section IV-B show that even with the conservative analysis, we were able to successfully transform over 80% of **char** pointers in real code.

The analysis is performed inter-procedurally, but the modification is limited to the function that contained the **char** pointer. This is to make the transformation usable. If the transformation suddenly makes changes to unrelated external functions, developers will find it hard to follow. Thus, if there is a possibility of a tainted buffer in the external function, we choose not to perform the transformation.

IV. Evaluation

To evaluate the usefulness of our program transformations, we answer the following research questions:

RQ1. Security. Are the program transformations effective in securing systems? More precisely, do they fix buffer overflow vulnerabilities originating from unsafe library functions and bad pointer operations?

RQ2. Correctness. Does a program transformation-based technique work? Do they break the original program? Do the program transformations scale to large programs?

RQ3. Performance. How is performance affected by the transformations? Do the safe libraries have additional overhead? How much overhead do the extra checks in safe data structure introduce?

Although the two transformations are similar to refactorings, we automated the process of applying the transformations as a batch in order to answer the three questions better. We applied the program transformations on all possible targets in real and benchmark programs. SLR was applied on all target unsafe functions; it replaced the unsafe functions. STR was applied on all **char** pointers; it replaced the **char** pointers with a safe data structure (stralloc pointers, Section III-C).

Our test corpus included benchmark programs and real software. To answer RQ1, we automatically applied the program transformations on benchmark programs of NIST's SAMATE reference dataset (Juliet Test Suite for C/C++) [48]. These programs represent many variants of buffer overflow vulnerabilities including the vulnerabilities originating from the use of unsafe library functions and bad pointer operations.

We also applied the program transformations on all possible targets on 4 open source software and demonstrated that changes made by the program transformations did not break the programs; this answers RQ2. In order to answer RQ3, we ran the original program and the modified program resulting from applying SLR and STR on all targets. This was done for 2 of the 4 open source programs.

A. *RQ1: Introduce Security Protection*

1) Securing Benchmark Programs: The SAMATE benchmark presents security errors in design, source code, binaries, etc. For C/C++ code, SAMATE's Juliet test suite version 1.2 provides 61,387 test programs for 118 different Common Weakness Enumerations (CWE). It is the most comprehensive benchmark available for C/C++ buffer overflow vulnerabilities.

TABLE III. CWEs DESCRIBING BUFFER OVERFLOWS

CWE	Transformations Applied		Total C Programs	KLOC	PP KLOC
	SLR	STR			
CWE 121: Stack Based Overflow	X	X	1,877	185.0	2,551.2
CWE 122: Heap Based Overflow	X	X	890	9.9	12.1
CWE 124: Buffer Underwrite		X	680	389.8	7,994.4
CWE 126: Buffer Overread		X	416	277.5	5,629.4
CWE 127: Buffer Underread		X	624	383.5	7,632.9
CWE 242: Use of Inherently Dangerous Function	X		18	2.8	23.9
			4,505	1248.5	23,843.9

KLOC: Lines of code / 1000; PP KLOC: Preprocessed KLOC

Table III lists 6 CWEs that describe buffer overflow vulnerabilities in their benchmarks. STR was applicable in 4,487 programs with 5 CWEs; it replaced **`char`** pointers in these programs. SLR was applicable to 1,758 programs across CWEs 121, 122, and 242; these CWEs use unsafe library functions that are fixed by SLR.

CWE 121 consists of 1,877 files which represent stack based buffer overflow. In these programs, a buffer is created and a certain amount of memory is allocated to the buffer. The buffer is then assigned a value that is too large for it to hold. STR modified all the programs replacing the assignments with functions from the `stralloc` library. The safe data structure will perform bounds checking. SLR was applied on a subset of these files that contain buffer overflow due to one of the unsafe functions targeted by SLR. It transformed 1,096 files.

CWE 122 consists of 890 files which represent heap based buffer overflow. Much like the programs from CWE 122, a buffer is created and allocated a certain amount of memory. The only difference is that in this case the memory is allocated on the heap rather than the stack. STR made similar changes. SLR was applicable to 644 files that contain buffer overflow due to one of the unsafe functions targeted by SLR.

CWE 124 consists of 680 files representing a buffer underwrite problem. In these files, a buffer is initially declared and allocated. The program then unsafely attempts to access a memory location before the start of the buffer. The modified programs replace the unsafe buffer with a `stralloc` pointer that prevents buffer underwrite by checking the bounds at runtime.

CWE 126 consists of 416 files with buffer overread problem. In these files, a buffer is initially declared and allocated. The program then attempts to access a memory location beyond the end of the allocated memory due to bad pointer arithmetic. The modified programs replace the buffer with a `stralloc` pointer which ensures that no unsafe arithmetic occurs.

CWE 127 consists of 624 files representing a buffer underread problem. In these files, like those in CWE 124, the program attempts to access a memory location before the start of an allocated buffer. The modified programs replace the buffer with a `stralloc` pointer to ensure that an unsafe memory location is not accessed.

CWE 242 consists of 18 files representing the use of an inherently dangerous function—`gets`. This is because `gets` copies input from standard input without any bounds. This allows users to easily overflow the buffer `gets` writes to. SLR was applied on `gets` in each of these files to replace it with a safer alternative that limits the number of bytes that can be written to the buffer.

Programs in SAMATE have a good function and a bad function. The good function uses a **`char`** buffer to perform some string operation and prints the buffer. The bad function attempts to do the same, but produces either a segmentation fault or incorrect output. After applying SLR and STR, the vulnerability was fixed in bad functions in all test programs.

2) Securing Real Programs: Among the open source software that we used to test the correctness of our transformations (Section IV-B), LibTIFF version 3.8.2 had a reported buffer overflow vulnerability with a known exploit [5]. SLR could be applied to fix the vulnerability.

The vulnerability is in the file tools/tiff2pdf.c, function t2p_write_pdf_string, line 3671.

```
3664  char buffer[5];
3667  len=strlen(pdfstr);
3669  for (i=0;i<len;i++){
3670    if((pdfstr[i]&0x80) || (pdfstr[i]==127)
          || (pdfstr[i]<32)){
3671      sprintf(buffer, "\\%.3o", pdfstr[i]);
3673    }
```

There is a vulnerability because the condition in line 3670 will be true if a character in `pdfstr[i]` has an unsigned numerical value of 128 or higher (the most significant bit is 1). This is because of the `pdfstr[i]&0x80` term. If the program runs the `sprintf` call on line 3671 when the value in `pdfstr[i]` is greater than 128, the value will be sign extended to an integer, which will produce more digits than expected and overrun the buffer. The value is signed extended to an integer because the format string calls for an integer. This vulnerability can exploited to cause a denial of service attack by trying to convert a TIFF file that has UTF-8 characters in its DocumentTag to PDF using the tiff2pdf tool.

SLR will remove the vulnerability by replacing the `sprintf` with `g_snprintf` and adding a third parameter of **`sizeof`**`(buffer)` which will allow only five bytes to be written to the buffer. This does not fix the program's behavior when UTF-8 characters are in a TIFF file's DocumentTag, but it does remove the possibility of buffer overflow. The program will work normally for TIFF files that do not have an attack vector, but will fail to generate the PDF file for the input with the attack vector. This modifies what was previously acceptable by the program to be unacceptable now, but such changes are beneficial [53].

B. RQ2: Correctness of Transformations

Our program transformations modify program behavior to fix a problem, but should not break normal behavior. We tested the program transformations on 4 open source programs: libpng-1.2.6, zlib-1.2.5, GMP-4.3.2, and LibTIFF-4.0.1. The transformations were each applied to more than 900,000 lines of code. In order to perform the test, we needed to preprocess the programs. The transformations ran automatically on 1.7 million lines of preprocessed code in 645 files (Table IV).

TABLE IV. TEST PROGRAMS

Software	# of C Files	KLOC	PP KLOC
zlib-1.2.5	29	54.0	89.3
libpng-1.2.6	16	112.2	167.0
GMP-4.3.2	528	457.0	1,097.7
libTIFF-4.0.1	72	281.5	385.0
	645	904.7	1,739.0

KLOC: Lines of code / 1000; PP KLOC: Preprocessed KLOC

SLR was applied on six unsafe functions used in the programs: `strcpy`, `strcat`, `sprintf`, `vsprintf`, `memcpy`, and `gets`. There were 317 instances of these functions. Of these 317 candidates, 259 were replaced correctly (81.7%). The other 58 candidates were not replaced because they failed SLR's preconditions. There were no cases where a replacement caused a compilation error. We also ran the test suite of the software (`make test`) each time SLR was applied. All the test cases passed. Table V summarizes the results.

TABLE V. RUNNING SLR ON TEST PROGRAMS

Software	# Unsafe Functions	# Transformed	% Transformed
zlib-1.2.5	41	17	41.46%
libpng-1.2.6	79	64	81.01%
GMP-4.3.2	61	52	85.26%
libTIFF-4.0.1	136	126	92.64%
	317	259	81.70%

Figure 2 shows the different kinds of unsafe functions and the percentage of the functions that has been changed. Of the 6 target functions, `gets` is not shown in the figure since it was not used in the open source software we tested. SLR mostly failed to transform instances of `memcpy` in code; it only transformed 72 of 115 instances (62.6%). This is because `memcpy` is is not limited to `char` buffers only. The goal of SLR is to replace unsafe string (`char` buffer) functions.

In every case where SLR's precondition failed, the buffer was of pointer type. We found four different reasons:

(1) In most of the cases, the definition of a buffer reaching its use does not contain an explicit heap allocation function (e.g., `malloc`). This may happen if the buffer is allocated by another function or is passed as a parameter from different call sites. In such cases, using `malloc_usable_size` is not safe.

(2) In one case, the buffer pointer was a part of a `struct` that was aliased. In fact, one other member of the `struct` was aliased in this case, not the entire struct. Our alias analysis treats a `struct` as an aggregate object (Section III-A).. SLR determines that it is safe not to transform the code. Our alias analysis can be made more precise, but that adds to the runtime overhead of the transformations. In practice, this was happening in only one case and could be ignored.

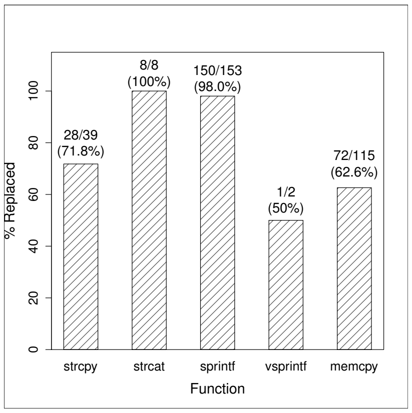

Fig. 2. Changes in Unsafe Functions by SLR

(3) In one other case, the buffer pointer was part of an array of buffers. We could not handle it because we do not support shape analysis on arrays.

(4) There was a single case where the definition of the buffer was the result of a ternary expression which contained heap allocation in both branches. This is an easy structural fix. We ignored it because it happened only once in 300 tests.

STR was applied to all `char` pointers in local scope, i.e., declared within a function. There were 296 candidates in the test programs. Of these 296 candidates, 59 were used in a potentially unsafe manner in user defined functions. We perform an inter-procedural analysis of pointers used in user defined functions. SAFE TYPE REPLACEMENT replaces the pointer when the it is not written to inside the user defined funciton. If it is, a detailed message is printed to the log explaining to the user exactly why the transformation could not be completed. The remaining 237 `char` pointers passed the preconditions of STR (shown as 'Buffers Replaced' in Table VI); these were locally declared and used. STR replaced all 237 `char` pointers (100%). Table VI summarizes the results.

TABLE VI. RUNNING STR ON TEST PROGRAMS

Software	Buffers Identified [C1]	Buffers Replaced [C2]	Buffers That Did Not Pass Precondition [C3]	% Of Buffers Replaced [C2/C1]	% Of Buffers That Passed Precondition Replaced [C2/(C1-C3)]
zlib-1.2.5	10	5	4	60.00%	100.00%
libpng-1.2.6	19	19	0	100.00%	100.00%
GMP-4.3.2	105	70	35	66.87%	100.00%
libTIFF-4.0.1	132	142	29	87.66%	100.00%
	296	237	59	80.01%	100.00%

For all our transformations, we ran `make test` to run the test suite for the programs. The results were the same for before and after programs. Our transformations did not break or change the functionality when executing unit tests.

C. RQ3: Effect on Performance

Our program transformations fix buffer overflows, but they may have performance overhead. We measured overhead using the open source programs in our corpus. We modified the programs after applying SLR and STR on all possible

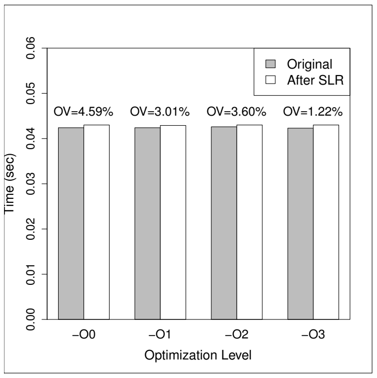

Fig. 3. libpng Perf. After SLR
OV=Percentage Overhead

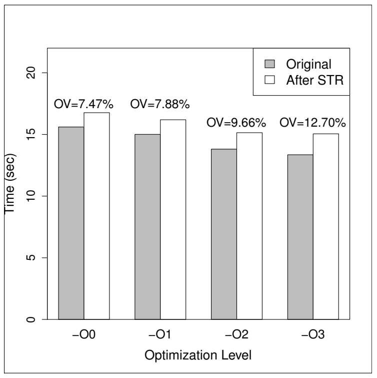

Fig. 4. LibTIFF Perf. After STR
OV=Percentage Overhead

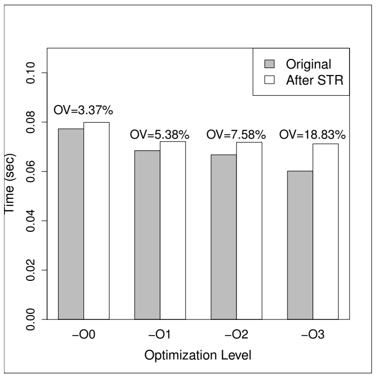

Fig. 5. zlib Perf. After STR
OV=Percentage Overhead

targets. Each program (before and after) was compiled with no optimization and then enabling optimization flags (-O1, -O2 and -O3). We ran the test suite accompanying the programs 100 times and averaged the runtimes to collect the performance numbers. Our test results show that neither the STR transformation nor the SLR transformation significantly altered the runtime of the programs tested. We discuss the runtime results of three programs.

Performance Overhead of SLR. SLR was tested on libpng and LibTIFF, since these had a lot of changes (Table V). Figure 3 shows the performance numbers for libpng. There is no noticeable performance change after the transformation. This makes sense because SLR replaces functions with safer alternatives whose behavior matches the original function as closely as possible. The authors of `strlcpy` family of functions also reported minimal overhead [45].

For libpng, the transformed version took 4.59% longer to run than the original when compiled with flag -O0, 3.01% longer when compiled with -O1, 3.60% longer when compiled with -O2, and 1.22% longer when compiled with -O3. The tests were run on a ThinkPad laptop with an Intel Core i5-3210M 2.50GHz x 4 CPU, 3.6GB of RAM, and Ubuntu 12.04 LTS operating system. We used gcc-4.2 to compile.

Performance Overhead of STR. STR performance was tested on three software, libpng, zlib, and LibTIFF. libpng and zlib had few changes, but they were executed inside hot loops. On the other hand, LibTIFF had a lot of changes (Table VI). Figures 4 and 5 show the performance numbers for LibTIFF and zlib. The overhead from applying the the STR transformations is relatively low.

After applying STR to LibTIFF, the -O3 optimized versions had approximately 12% overhead. The highest overhead was for zlib's -O3 optimized version (18.83%). Interestingly, the transformed version was not optimized by the compiler beyond -O0. This makes sense intuitively as our transformation changed only in a few places. After the compiler optimizes with -O1, there is very little left for it to do for -O2 and -O3 optimization levels.

For STR, the programs were tested on a MacBook machine with a 2 GHz Intel Core 2 Duo Processor and 4GB 1067 MHz DDR3 memory running Mac OS X Lion 10.7.4. We used llvm-gcc 4.2.1 to compile the programs.

V. Related Work

This section compares our transformation based approach with the approaches of detecting and fixing buffer overflow vulnerabilities. First we compare our general approach; this is followed by a discussion of the two transformations.

A. General Approach

Most of the research works on buffer overflow vulnerabilities and static analysis concentrate on detecting buffer overflows [10], [15], [18], [36], [38], [62], [63], [65]. On the other hand, dynamic analysis approaches mostly concentrate on preventing buffer overflow vulnerabilities, except for a few detection approaches [46], [55], [57]. The static approaches range from very simple lexical analyzer [62] to tools performing integer analysis to approximate the pointer arithmetic in C source code [63] to applying abstract interpretation to prove the absence of runtime errors [10]. Most of the early approaches suffer from a high rate of false positives. But, approaches that have used symbolic analysis to detect buffer overflow [2], [21], [22], [36], [54], [63], [65] have reported better results. Our program transformations, on the other hand, attempt to fix buffer overflows by replacing all instances of unsafe library functions and **`char`** pointers.

Cowan and colleagues [13] identified four basic approaches to defend against buffer overflow vulnerabilities: (1) writing correct code; (2) making the stack segment non-executable; (3) checking array bounds; and (4) checking the integrity of code pointer. The program transformations in this paper fall in the first category; these assist developers in writing secure code as they are coding, similar to refactorings. A few hardware based approaches have explored non-executable stacks, e.g., Linux Openwall project [49], but it requires the OS kernel to be patched. Other approaches have split the control and data stack [66], but it is hard to calculate the function return address. Besides, these approaches focus on stack-based buffer overflow only. Most dynamic analysis solutions fall in the third or the fourth categories. Array bounds checking approaches [29], [30], [35], [37] provide better protection, but these have high performance overhead. Some code pointer integrity checking approaches, such as StackGuard [12], PointGuard [11], and ProPolice [17], use markers (canaries) to check for buffer integrity; others store the copy of the return address in a safe place, e.g., Stack Shield [61] and GMM [39]. However, these tools provide a partial solution [64] that can be circumvented [6]. The performance overhead is lower but still significant.

B. Safe Library Replacement Transformation

Bartaloo and colleagues' [3] libsafe library and the GMM library [39] are dynamically loaded libraries that replace unsafe library functions. These use the `LD_PRELOAD` feature to dynamically load the libraries. Once preloaded by a vulnerable process, the libraries intercept all C library functions, and allows functions to execute only if the arguments respect their bounds. These approaches work as binary patches and the application does not have to be recompiled. The SAFE LIBRARY REPLACEMENT transformation allows the developer to add similar protection at source code level; they have the additional advantage of allowing a developer to understand and learn how to write secure code. They are similar to refactorings [20], but they are security-oriented program transformations [24], [25] that push refactorings beyond behavior preservation. Most importantly, they introduce actual solutions to buffer overflow as opposed to patches that only cover potential overflow vulnerabilities in source code, but do not actually fix them.

The SAFE LIBRARY REPLACEMENT transformation is not limited to preventing buffer overflows. The REPLACE ARITHMETIC OPERATOR transformation [7] replaces an arithmetic operation in a C program with a call to a safe library function that correctly handles integer overflow and underflow. SQL injection attacks can be prevented by replacing all instances of string concatenation based SQL queries with SQL `PreparedStatement` [4], [16], [58]

C. Safe Type Replacement Transformation

SAFE TYPE REPLACEMENT replaces an unsafe datatype with a safe data structure [23], [26]. Another approach to replace unsafe types is explored by the Gemini tool [14]. Interestingly, Gemini is the only program transformation approach to introduce protection against buffer overflow. Gemini, written in TXL [9], transforms all stack-allocated buffers in a C program to heap-allocated buffers, because exploiting a heap overflow is more difficult. But it does not remove the actual vulnerabilities. Some approaches that explored new C dialects, most notably CCured [8] and Cyclone [34], combined static analysis and runtime checks: these checked source code for safety and added runtime checks where safety cannot be guaranteed statically. Both require manual intervention in the form of annotations and source code changes. Both are C dialects; on the contrary, SAFE TYPE REPLACEMENT introduces an automated protection that is also pure C.

Many options of safe library and safe data structure are available. Table I lists some safe library options. Some of these safe functions trim the resultant destination buffer to fit its size [32], [45], while other functions dynamically resize the destination buffer [33], [43]. Some examples of safe data structures are `stralloc` data structure used in qmail [26], `sm_str_s` data structure used in MeTA1 [1], and the data structure described by Narayanan [47].

VI. FUTURE WORKS AND CONCLUSION

Why do we need another research work on buffer overflow? Despite the extensive research on detecting and fixing buffer overflows, there continues to be a lot of overflow vulnerabilities that are reported even in mature software. Studies have also revealed that the detection tools are not used in practice and the overhead of manually fixing the detected vulnerabilities is too much. Our program transformation based approach naturally fits into the programming chore, much like the refactoring tools. Therefore, these tools have a better chance to be adopted. We provide the power tools missing from developers' toolkit.

We made several design decisions while implementing the transformations, e.g., which library to use as alternatives, or which safe data type to use. The choices are all supported by latest research; but we plan to validate these decisions with usability studies in future.

Our transformation based approach improves security, but has a more significant impact on improving dependability since the transformations can be applied as a batch to fix the root cause behind buffer overflows. The transformations are supported by sophisticated analyses and empirical data that they do not break original programs. They are not silver bullets, but have nearly the same effect on fixing the targeted root causes.

VII. ACKNOWLEDGEMENTS

We thank Paul Adamczyk, Ralph Johnson, Jeffrey Overbey, and the anonymous reviewers. This work was supported by the NSF grants CCF-1217271 and CCF-1340124.

REFERENCES

[1] C. Aßmann. MeTA1 README. Technical report, *Sendmail Inc.*, Jan 2007.

[2] D. Babic and A. Hu. Calysto: scalable and precise extended static checking. In *ICSE '08*, pages 211–220, New York, NY, USA, 2008. ACM.

[3] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack-smashing attacks. In *2000 USENIX Annual Technical Conference: San Diego, CA, USA*, 2000.

[4] P. Bisht, A. P. Sistla, and V. Venkatakrishnan. Automatically preparing safe SQL queries. In *FC '10*, pages 272–288, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Bugtraq ID 18331. LibTIFF tiff2pdf remote buffer overflow vulnerability.
http://www.securityfocus.com/bid/18331, 2006.

[6] Bulba and Kil3r. Bypassing StackGuard and Stack Shield. *Phrack Magazine*, 10(56):File 5, 2000.

[7] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *ICSE '13*, pages 792–801, Piscataway, NJ, USA, 2013. IEEE Press.

[8] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *PLDI '03*, pages 232–244, New York, NY, USA, 2003. ACM.

[9] J. R. Cordy. Source transformation, analysis and generation in TXL. In *PEPM '06*, pages 1–11. ACM, 2006.

[10] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRéE analyzer. In *PLDI '05*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *USENIX SECURITY Symposium '03*, pages 91–104. USENIX, Aug. 2003.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX SECURITY Symposium '98*. USENIX, 1998.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DISCEX '00)*. IEEE Computer Society Press, Jan. 2000.

[14] C. Dahn and S. Mancoridis. Using program transformation to secure C programs against buffer overflows. In *WCRE '03*, page 323, Washington DC, USA, 2003. IEEE Comp. Society.

[15] N. Dor, M. Rodeh, and S. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI '03*, pages 155–167. ACM, 2003.

[16] F. Dysart and M. Sherriff. In *ISSRE '08*.

[17] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.research.ibm.com/trl/projects/security/ssp/, 2000.

[18] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19:42–51, January 2002.

[19] F. Cavalier III. Libmib allocated string functions. http://www.mibsoftware.com/libmib/astring/.

[20] M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.

[21] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03*, pages 345–354. ACM, 2003.

[22] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE '06*, pages 232–241, New York, NY, USA, 2006. ACM.

[23] M. Hafiz. Security architecture of Mail Transfer Agents. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[24] M. Hafiz. *Security On Demand*. PhD thesis, University of Illinois at Urbana-Champaign, 2010.

[25] M. Hafiz, P. Adamczyk, and R. Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *ESSoS '09*, Feb 2009.

[26] M. Hafiz and R. Johnson. Evolution of the MTA architecture: The impact of security. *Software—Practice and Experience*, 38(15):1569–1599, Dec 2008.

[27] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. OpenRefactory/C: An infrastructure for building correct and complex C transformations. In *WRT '13*, 2013.

[28] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*, pages 290–299, New York, NY, USA, 2007. ACM.

[29] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX '92*, pages 125–136, 1992.

[30] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *NDSS*. The Internet Society, 2003.

[31] International Organization for Standardization. *ISO/IEC 24731: Specification For Secure C Library Functions*. 2004.

[32] International Organization for Standardization. *ISO/IEC 9899:TC3: Programming Languages — C*. Sep 2007.

[33] ISO/IEC 14882. C++ std::string.

[34] T. Jim, J. G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX ATEC '02*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[35] R. Jones and P. Kelly. Bounds checking for C. http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html, July 1995.

[36] W. Le and M. L. Soffa. Marple: A demand-driven path-sensitive buffer overflow detector. In *FSE-16*, pages 272–282, New York, NY, USA, 2008. ACM.

[37] K. Lhee and S. Chapin. Type-assisted dynamic buffer overflow detection. In *USENIX 2002*, pages 81–88, 2002.

[38] L. Li, C. Cifuentes, and N. Keynes. Practical and effective symbolic analysis for buffer overflow detection. In *FSE '10*, pages 317–326, New York, NY, USA, 2010. ACM.

[39] D. Libenzi. Guarded memory move (GMM), Feb. 10 2004.

[40] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *ICSE '13*, pages 282–291, 2013.

[41] M. Lovell. Repel attacks on your code with the Visual Studio 2005 safe C and C++ libraries. *MSDN Magazine*, May 2005.

[42] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA '10*, pages 428–443. ACM, 2010.

[43] M. Messier and J. Viega. Safe C string library v1.0.3. http://www.zork.org/safestr/safestr.html.

[44] Microsoft Developer Network. Using the Strsafe.h functions.

[45] T. Miller and T. de Raadt. `strlcpy` and `strlcat` — Consistent, safe, string copy and concatenation. In *USENIX '99*, 1999.

[46] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *PLDI '09*, pages 245–258, New York, NY, USA, 2009. ACM.

[47] A. Narayanan. Design of a safe string library for C. *Software—Practice and Experience*, 24(6):565–578, 1994.

[48] National Institute of Standards and Technology (NIST). SAMATE - Software Assurance Metrics and Tool Evaluation, 2012.

[49] Openwall Project. Linux kernel patch from the Openwall project. http://www.openwall.com/linux/.

[50] S. Özkan. CVE Details: The ultimate security vulnerability datasource—Vulnerabilities By Type. http://www.cvedetails.com/vulnerabilities-by-types.php.

[51] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of parallelism in algorithms. In *PLDI '11*, pages 12–25. ACM, 2011.

[52] qmail home page. Maintained by Daniel Julius Bernstein. http://cr.yp.to/qmail.html.

[53] M. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *OOPSLA '05*, pages 21–30. ACM, 2005.

[54] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI '00*, pages 182–195, New York, NY, USA, 2000. ACM.

[55] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *DSSS 11*, pages 159–169, 2004.

[56] R. Seacord. *The CERT C secure coding standard*. Addison-Wesley, 2009.

[57] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX 2005, year = 2005, pages = 17-30,*.

[58] S. Thomas, L. Williams, and T. Xie. On automated prepared statement generation to remove SQL injection vulnerabilities. *Inf. Softw. Technol.*, 51(3):589–598, Mar. 2009.

[59] US-CERT. National vulnerability database version 2.2.

[60] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE '12*, pages 233–243. IEEE, 2012.

[61] Vendicator. Stack Shield: A stack smashing technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/, 2000.

[62] J. Viega, J. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC '00*. ACM, 2000.

[63] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*. The Internet Society, 2000.

[64] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[65] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28:327–336, September 2003.

[66] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks. In *EASY '02*, 2002.