Removing Buffer Overflows In C Programs With Safe Library Replacement Transformation

by

Dusten James Doggett

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 14, 2013

Abstract

This work explores how buffer overflow vulnerabilities in C programs, specifically the ones that originate from the use of unsafe functions, can be fixed by using a source-to-source program transformation. I implemented a Safe Library Replacement transformation that replaces unsafe library functions with safe alternatives. The transformation improves the security of a system, which means that it does not preserve the original behavior of the program. It preserves good-path behavior, and modifies the behavior only on attack vectors. Implementing the transformation in C requires sophisticated static analyses that are typically unavailable in existing program transformation infrastructures for C. I used OpenRefactory/C, a framework for building correct and complex program transformations for C; I enhanced the infrastructure to support control flow and alias analysis. I tested the transformation on 1,778 test cases from the SAMATE reference dataset, and was able to remove the buffer overflow vulnerability from each case. I also applied the transformation on 181 instances of unsafe functions in three real C programs. The transformation replaced the function in 73% of the cases, and did not break the original program in any of the cases. A program transformation-based approach can integrate with a developer's coding activity, much like a refactoring, and allows a developer to fix library-related buffer overflow problems on demand.

Acknowledgments

I thank my fiancé, Britney, for being supportive and patient while I finish school, my parents, Geoffrey and Suzan, for believing in me and constantly encouraging me to keep working, and my brother, John Kyle, for being a great room-mate during my years as a graduate student. I could not have done this work without my family.

I thank my adviser, Dr. Munawar Hafiz, for guiding me and pushing me to do the best work that I could, and Dr. Jeffrey Overbey, for always being available to help me when I was stuck.

Finally, I thank my colleagues in the Software Analysis, Transformation, and Security lab for being excellent people to work with.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

SLR	Safe Library Replacement
SOPT	Security-Oriented Program Transformation
STR	Safe Type Replacement
AST	Abstract Syntax Tree
CRED	C Range Error Detector
CWE	Common Weakness Enumeration

Chapter 1

Introduction

Security is challenging to implement in a software system. It is even more challenging to add or upgrade security after the system has been implemented. This typically arises when developers or customers decide they want to add or upgrade security as a new feature, or when developers or attackers discover a new vulnerability.

The latter is more pressing because the vulnerability needs to be removed as quickly as possible to stop attacks. Security engineers are constantly trying to keep up with attackers. Great strides have been made in developing a body of knowledge on potential vulnerabilities and how to avoid them when implementing software systems. However, attackers have access to this knowledge too, and their ability to find new vulnerabilities is only limited by their creativity. Security engineers can employ their own creativity to create patches when new vulnerabilities are discovered, but they frequently find themselves lagging behind attackers.

One reason attackers have an advantage is that many types of vulnerabilities are applicable to many different software applications. This allows attackers to be systematic and use automated tools in their approach to finding vulnerabilities. Those vulnerabilities can then be exploited in the same way. Another advantage attackers have is that they do not need source code to find vulnerabilities. They can use their automated tools on executables and object code as well.

Security engineers, on the other hand, commonly respond to a new vulnerability by creating a specific patch for that vulnerability in an ad hoc manner. The patch often must be applied manually to the source code, and may leave other vulnerabilities of the same type untouched. Adding new security features is also often done in an ad hoc manner. Ad hoc patching approaches are laborious and time consuming. The difficulty of adding security to a software system after it has been developed has led security experts to believe that "security cannot be added on, it must be designed from the beginning." [2]

Redesigning entire software systems when a new vulnerability is found is impractical. Systematic and automated means of retrofitting security in a software system would have great value, and level the playing field between security engineers and attackers. A number of security retrofitting strategies, collectively called SECURITY-ORIENTED PROGRAM TRANSFORMATIONS, have been proposed by Hafiz [12].

1.1 Security On Demand

The idea of SECURITY-ORIENTED PROGRAM TRANSFORMATIONS is to add security to the source code of an existing software system systematically so that there are no vulnerabilities in the system for attackers to find. In other words, SECURITY-ORIENTED PROGRAM TRANSFORMATIONS provide 'Security On Demand'. Hafiz [12] describes thirty-seven general purpose SECURITY-ORIENTED PROGRAM TRANSFORMATIONS. Program transformations come in many varieties and have been around for a long time. Compilers transform high-level language code to a specific machine language code. Refactorings are source code transformations that change the structure of a program, but do not change its behavior [10]. SECURITY-ORIENTED PROGRAM TRANSFORMATIONS are similar to refactorings, but they are meant to improve security. They alter the behavior of the program to correct the program's response to certain attacks, but all other behavior is left unchanged.

This thesis focuses on implementing and evaluating one of the SECURITY-ORIENTED PROGRAM TRANSFORMATIONS identified by Hafiz, called SAFE LIBRARY REPLACEMENT transformation. Hafiz narrates the process by which SAFE LIBRARY REPLACEMENT transformation could be performed, and also provides a proof-of-concept Perl script that uses lexical analysis to try to perform SAFE LIBRARY REPLACEMENT transformation. The Perl script could not replace the unsafe function in all cases. This implementation will give better coverage of unsafe function replacements because it uses more sophisticated analysis techniques.

1.2 Safe Library Replacement Transformation

The Safe Library Replacement transformation focuses on the removal of a certain type of vulnerability called a buffer overflow.

Buffer overflows are one of the most common and well known security vulnerabilities, especially for C programs. A buffer is a block of contiguous memory. It is overflowed if memory beyond its boundaries is written to unintentionally. An attacker, through careful trial and error, can overwrite an important piece of memory if he is given access to a buffer that he can overflow. The attacker can, at minimum, crash the program. In the worst case he can cause the program to execute arbitrary code of his own design.

Buffer overflows are common in C programs because C does not have automatic bounds checking. Developers must ensure that buffer writes are in bounds. There are many ways buffer overflows can occur in a C program. The Safe Library Replacement transformation removes buffer overflows by eliminating uses of library functions that contain buffer overflow vulnerabilities, such as `strcpy`.

The Safe Library Replacement transformation replaces calls to unsafe library functions with calls to safe alternatives, such as `g_strlcpy` for `strcpy`. This is more difficult than a simple textual find and replace because safe library functions almost always require additional parameters to help them prevent buffer overflows. Table 2.1 shows a table of unsafe functions and their safe alternatives. One challenge to implementing the Safe Library Replacement transformation is correctly determining what the values of these new parameters should be, since the transformation must remove the buffer overflow vulnerability while preserving all other behavior.

1.3 Safe Type Replacement Transformation

In some cases the SAFE LIBRARY REPLACEMENT transformation cannot be done without risk of changing the program's behavior beyond removing the buffer overflow vulnerability. In such cases, another SECURITY-ORIENTED PROGRAM TRANSFORMATION from Hafiz [12] may work. It is called SAFE TYPE REPLACEMENT transformation.

The SAFE TYPE REPLACEMENT transformation removes buffer overflows caused by uses of character strings by replacing character strings with a safe struct type that contains the character string. The struct is shown in Figure 1.1. The struct type is safe because it keeps track of the number of bytes allocated to the character string, and the number of bytes in the character string that are in use. Every definition and use of the character string that is being replaced must be replaced with a function specific to the safe struct type. The replacing function has the same behavior as the replaced definition or use, but also keeps track of the buffer's allocation and length, and prevents overflows.

Figure 1.1: Safe type for character strings

```
struct stralloc {
    char *s; // pointer to the first byte of the stralloc's memory
    unsigned int len; // numbers of chars in the stralloc
    unsigned int a; // length (in bytes) of the stralloc's memory
}
```

The SAFE TYPE REPLACEMENT transformation does not directly target overflows due to unsafe library functions, but it will fix them incidentally if it replaces a character string that is used as the buffer being written to in an unsafe library function. An implementation and evaluation of the SAFE LIBRARY REPLACEMENT transformation is not included in this work, but it is planned for future work.

1.4 Thesis Statement And Contributions

Simple program transformations that replace vulnerable library functions can be safely implemented for C, and they can be effective to secure programs from buffer overflow vulnerabilities from those vulnerable functions. This thesis makes the following contributions:

- I implemented a Safe Library Replacement transformation for C programs using sophisticated static analysis. The transformation supports six unsafe library functions: `strcpy`, `strcat`, `sprintf`, `vsprintf`, `memcpy`, and `gets`.
- I implemented an alias analysis for C programs. Alias analysis is a highly non-trivial static analysis that was necessary to correctly implement Safe Library Replacement transformation.
- I evaluated the effectiveness of my Safe Library Replacement transformation implementation on two test corpuses: the Juliet Test Suite for C/C++ from the SAMATE Reference Dataset [32] and three real C programs, zlib, libpng, and gmp.
  - The Juliet Test Suite contains 61,387 test cases on a wide variety of bugs and weaknesses categorized as Common Weakness Enumerations (CWEs). I selected 1,778 cases that are relevant to the Safe Library Replacement transformation, and ran the transformation on all of them to determine the frequency with which it could correctly replace unsafe function calls.
  - I used the testing procedure described in [11] to test the Safe Library Replacement transformation on zlib, libpng, and gmp. The transformation was automatically run on every use of the unsafe library functions I targeted in the three C programs to determine the frequency with which the it could correctly replace them. Each test case was done independently starting with the original code.

The evaluation showed that my Safe Library Replacement transformation implementation could replace the unsafe function and remove the buffer overflow vulnerability while preserving all other behavior in all 1,778 test cases from the Juliet Test Suite. Of the 181 unsafe functions replaced in the C programs, 73% the transformation replaced correctly (the programs' internal test cases still passed), 27% the transformation decided not to replace, and in no cases did the transformation break the program by doing a replacement.

1.5 Thesis Outline

This work is organized as follows:

- Chapter 2 discusses the idea of replacing unsafe library functions with safe alternatives to remove buffer overflows.
- Chapter 3 discusses OpenRefactory/C, which is a framework for building source-level program analyses and transformations for C. I used it to implement Safe Library Replacement transformation.
- Chapter 4 discusses the algorithm I used in my implementation of the Safe Library Replacement transformation.
- Chapter 5 discusses my implementation of alias analysis for OpenRefactory/C.
- Chapter 6 discusses the evaluation of my Safe Library Replacement transformation implementation.
- Chapter 7 discusses work related to this one.
- Chapter 8 concludes.

Chapter 2
Using Safe Alternatives To Unsafe Library Functions To Remove Buffer Overflows

One of the root causes of buffer overflow vulnerabilities in C is the use of unsafe library functions. The vulnerability of some unsafe function has been recognized and safe alternatives have been written, but the use of unsafe functions is abundant in old C code and they continue to be used today. Table 2.1 shows a number of unsafe functions and some of their safe alternatives.

The `strcpy`, `strcat`, `sprintf`, `memcpy`, `gets`, and `getenv` functions are all vulnerable because they write to a buffer with no guarantee that the write will be within the buffer's bounds. Some of the safe alternatives attempt to resolve this by taking an additional parameter that limits how many bytes are filled in the destination buffer, such as `g_strlcpy`. Others take a dereference of the buffer pointer so that the buffer can be reallocated if more memory is needed, such as `astrcpy`.

The `mktemp` and `tmpnam` functions are included to show that library functions can be unsafe for reasons other than buffer overflow vulnerability. These functions create names for temporary files that do not conflict with existing files. Attackers can predict the names that may be created and perform a denial-of-service attack by creating the files before `mktemp` or `tmpnam` can. This is called a time-of-check to time-of-use race condition vulnerability.

The `sprintf` function also has an additional format string vulnerability. A format string vulnerability is present when the format string of a formatting function, like `sprintf`, comes from unchecked user input. A malicious user can use a combination of the `%x` and `%n` format tokens to execute arbitrary code at some arbitrary address. While the idea of

Table 2.1: Some Unsafe Functions And Their Safe Alternatives

Unsafe functions	**Safe alternatives**

strcpy(3) - Copy string
`char *strcpy (char *dst,`
`const char *src);`
	g_strlcpy from glib library [29],
	astrcpy, astrn0cpy from libmib library [8],
	strcpy_s from ISO/IEC 24731 [18] and SafeCRT library [25],
	StringCchCopy, StringCchCopyN from StrSafe [28] library,
	safestr_copy and safestr_ncopy from Safestr library [27]
	`gsize g_strlcpy (gchar *dst, const gchar *src`
	`gsize dst_size);`
	`char *astrcpy (char **dst_address, const char *src);`

strcat(3), strncat(3) - Concatenate string
`char *strcat (char *dst,`
`const char *src);`
	g_strlcat from glib library [29],
	astrcat, astrn0cat from libmib library [8],
	strcat_s from ISO/IEC 24731 [18] and SafeCRT library [25],
	StringCchCat, StringCchCatN from StrSafe [28] library,
	safestr_concatenate from Safestr library [29]
	`gsize g_strlcat (gchar *dst, const gchar *src,`
	`gsize dst_size);`
	`char *astrcat (char **dst_address, const char *src);`

sprintf(3), snprintf(3) - Concatenate string
`char *sprintf (char *str,`
`const char *format, ...);`
	g_snprintf from glib library [29],
	asprintf from libmib library [8],
	sprintf_s from ISO/IEC 24731 [18] and SafeCRT library [25],
	`gint g_snprintf (gchar *string, gulong n,`
	`gchar const *format, ...);`
	`int asprintf (char **ppasz, const char *format, ...);`

memcpy(3) - Copy memory area
`void *memcpy (void *dst,`
`const void *src, size_t num);`
	memcpy_s from ISO/IEC 24731 [18]
	`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 [18], fgets from C99 [19],
	afgets from libmib library [8], gets_s from SafeCRT library [25]
	`char *gets_s (char *destination, size_t dest_size);`
	`char *fgets (char *dst, int dst_size, FILE *stream);`

getenv(3) - Get value of an environment variable
`char *getenv (const char *name);`
	getenv_s function [39]
	`errno_t getenv_s (size_t *return_value,`
	`char *dst, size_t dst_size, const char *name);`

mktemp(3), tmpnam(3) - Create a temporary file
`char *mktemp(char *template);`
`char *tmpnam(char *s);`
	mkstemp(3) and mkdtemp(3) from standard library
	`int mkstemp(char *template);`
	`char *mkdtemp(char *template);`

replacing unsafe library functions with safe alternatives can be applied to functions with any vulnerability, I focus on the functions with buffer overflow vulnerabilities in this work.

Before safe alternatives were available it was up to developers to prevent buffer overflows by using vulnerable functions carefully. This was done by ensuring that the buffer that might be overflowed would always have enough room before calling the unsafe function. Relying on developers inevitably results in occasional human error. Developers may forget to ensure enough space in the buffer, or they may not check its capacity correctly. A simple example of buffer overflow from the use of `strcpy` is shown in Figure 2.1.

Figure 2.1: Unsafe function buffer overflow example

```
 1   void foo() {
 2      char destination[50];
 3      destination[0] = ‘\0’;
 4
 5      char source[100];
 6      memset(source, ‘C’, 100-1);
 7      source[100-1] = ‘\0’;
 8
 9      strcpy(destination, source);
10   }
```

In the example, a destination buffer of fifty bytes is allocated and null terminated to make it an empty C-string on lines two and three. Next a source buffer of one-hundred bytes is created, filled with ninety-nine ‘C’ characters, and null terminated to make it a one-hundred byte C-string in lines five to seven. Finally, `strcpy` is given the `destination` and `source` buffers in line nine. `strcpy` will continue to copy bytes from `source` to `destination` until it finds the null terminator in `source`. `source`’s null terminator is at index one-hundred, so `strcpy` will copy one-hundred bytes, overflowing `destination`.

Developers continue to use unsafe library functions, despite the existence of safe alternatives. They may not be aware of the safe alternatives, may forget to use them, or choose not to use them. There also exists a great amount of C code written before safe alternatives were available. Manually updating old code to use safe alternatives is tedious, time consuming, and again prone to human error. A refactoring tool that automatically replaces unsafe library functions with correct uses of safe alternatives would be of great value. It could be used to retrofit old C code, and as a part of the development process.

Finding unsafe functions to replace in a C program can be done with a simple textual search. Replacing them can be difficult. The easiest step of every replacement is changing the function name. I do this with a simple textual search of the function call being replaced for the old name, and then a substring replacement of the old name with the new name. Most of the replacements also require the addition of one or more new parameters, as can be seen in Table 2.1. Determining, correctly, what these parameters should be is the major challenge to replacing unsafe library functions.

A critical requirement of the Safe Library Replacement transformation is that it remove the buffer overflow vulnerability without changing anything else about the program. If it alters program behavior in unexpected ways, it is of no use to developers. They will not use it. *Therefore, the replacement must be conservative. If the correct values of the new parameters cannot be determined with absolute certainty, the replacement cannot be done.* Uncertainty about the correct value of the new parameters can happen for many reasons. The most common case is that Safe Library Replacement transformation cannot accurately determine the length of the buffer that might be overflown.

Chapter 3

OpenRefactory/C

OpenRefactory/C is a plug-in for OpenRefactory, which is a framework for building source-level program analyses and transformations. It is written in Java. It is called Open-Refactory because it is designed to be extensible in the refactorings it can support and the languages it can refactor. OpenRefactory/C adds support for analysis and refactoring of C code. OpenRefactory/C is being actively developed by the Software Analysis, Transformation, & Security lab led Dr. Munawar Hafiz at Auburn University.

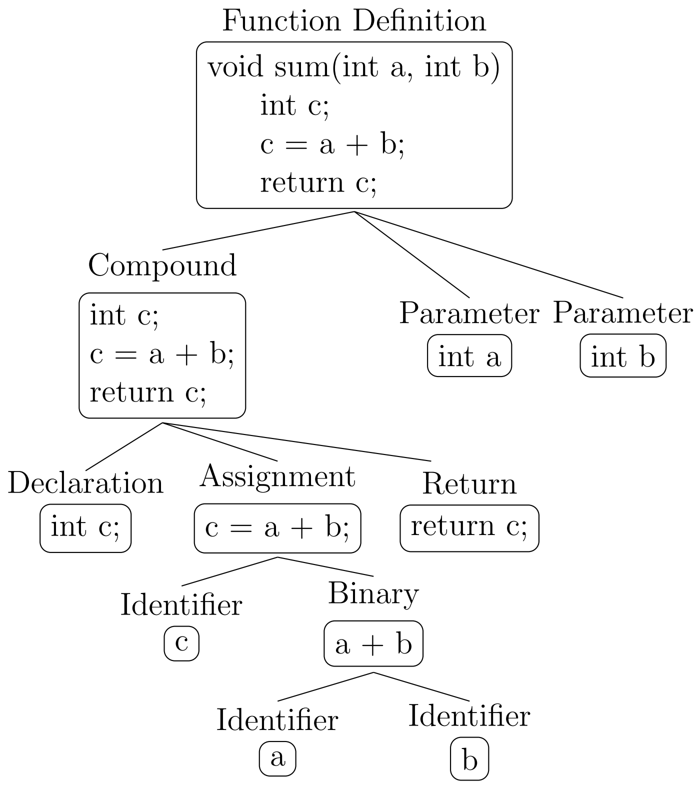
Modern languages like Java and C# have many IDEs that support maintenance and code evaluation with automated refactorings. C, despite its popularity, lacks IDEs with sophisticated static analyses and refactorings and static analyses. Without good static analysis support, it is impossible to create non-trivial program transformations like Safe Library Replacement and Safe Type Replacement. The goal of OpenRefactory/C is to fill this need.

The internal program representation of OpenRefactory/C is a rewritable abstract syntax tree (AST). Figure 3.1 shows a simplified example of a function definition's AST. All code analysis in OpenRefactory/C is performed by traversing the AST, and all refactoring is done by rewriting the AST.

Every syntactic construct in the C language has a corresponding AST node type that represents it in an AST. For example, the Function Definition node in Figure 3.1 represents the function definition in the code, and it contains a list of Parameter nodes and a Compound node which represent its parameters and body.

There is a complex type hierarchy of AST nodes in which there are some abstract node types that generalize certain families of node types. Two particularly important abstract

Figure 3.1: Simple example of a partial AST



types are the Expression node and Statement node. The Expression node abstracts basic C expressions like identifiers, binary expressions, and array access expressions. The Statement node abstracts more complex C structures like while loops and if/else statements.

The Safe Library Replacement transformation was implemented in OpenRefactory/C to take advantage of its powerful AST representation and several important static analyses. Two of these analyses are type and name binding. The type analysis returns the type of any Expression or Declaration node in the AST. The name binding analysis links all Identifier nodes for a certain variable to their Declaration node. The name binding can be used to find the Declaration node of any Identifier node, or all the Identifier nodes that refer to the same variable of a particular Identifier.

Other important static analyses used in Safe Library Replacement transformation are reaching definition and alias analyses. The reaching definition analysis returns a list of all the nodes inside a Function Definition that contain definitions. This, in turn, uses a control flow analysis to determine the order of execution of the nodes. The alias analysis will see use in many other OpenReafctory/C analyses and transformations, but it was originally implemented for this project. Implementing it was complex and critical enough to the SAFE LIBRARY REPLACEMENT transformation that it warrants a description in its own chapter. The ways these analyses are used by the SAFE LIBRARY REPLACEMENT transformation are discussed in detail in Chapter 4.

Chapter 4
Algorithm For Safe Library Replacement Transformation

The objective of the Safe Library Replacement transformation is to replace an unsafe function call with a safe alternative. My Safe Library Replacement transformation algorithm focuses on replacing functions that have buffer overflow vulnerability. The input to the transformation is a source code file and a selected function call to replace in that code. The output is a transformed source code file, or the original source code file if the transformation could not be done. The functions that the implementation can currently replace are `strcpy`, `strcat`, `sprintf`, `vsprintf`, `memcpy`, and `gets`. Figure 4.1 describes the algorithm.

The algorithm first checks if the function call is one of the functions it supports. It takes different steps for each supported function. If the algorithm does not support the selected function, it stops. If the function call is supported, then the algorithm takes steps to replace it with a safe alternative. The algorithm may stop during any one of these steps if it determines that it cannot do the replacement without risk of changing more of the program's behavior beyond removing the buffer overflow vulnerability.

There are only minor differences between the steps to replace each supported function. Determining the length of the buffer being written to is the first step to replacing every supported function. The steps for `strcpy`, `strcat`, `sprintf`, and `vsprintf` are the same, so they are combined. The cases for `memcpy` and `gets` have some additional steps in which additional lines of code may be added.

The steps to replace `strcpy`, `strcat`, `sprintf`, and `vsprintf` (lines five to eight) are described in Section 4.1. The steps to replace `memcpy` (lines element to sixteen) are described in Section 4.2. The steps to replace `gets` (lines nineteen to twenty-three) are described in

Figure 4.1: Main Safe Library Replacement transformation algorithm

```
SafeLibraryReplacementTransformation(functionCall):
   - make replacementCall copy of original functionCall

   if (functionCall is strcpy, strcat, sprintf, or vsprintf)
      - determine length of buffer being written to
      - add new length parameter to replacementCall
      - change name of replacementCall
      - replace functionCall with replacementCall

   else if (functionCall is memcpy)
      - determine length of buffer being written to
      if (functionCall's length parameter is assignable)
         - add new assignment line above functionCall
      else
         - replace length parameter in replacementCall
         - replace functionCall with replacementCall

   else if (functionCall is gets)
      - determine length of buffer being written to
      - add new stream and length parameters to replacementCall
      - change name of replacementCall
      - add newLine removal code after functionCall
      - replace functionCall with replacementCall

   else
      - stop
```

Section 4.3. The algorithm for determining the length of the buffer being written to (used on lines five, element, and nineteen) is the most complex step by far, and is a common step in every replacement; it is described in Section 4.4.

4.1 Replacing `strcpy`, `strcat`, `sprintf`, and `vsprintf`

The safe alternatives I used for these functions are the glib functions `g_strlcpy`, `g_strlcat`, `g_snprintf`, and `g_vsnprintf`. Each safe alternative differs from its original only by its function name and the addition of a new length parameter that limits the number of bytes copied. As a result, the process to replace these functions is almost identical: determine the length of the buffer being written to, add that length as a new parameter, and change the name of the function. Table 4.1 shows a simple example of replacing `strcpy`.

Table 4.1: Simple `strcpy` replacement example

Before	**After**
`char destination[50];`	`char destination[50];`
`char source[100];`	`char source[100];`
`memset(source, ‘C’, 100-1);`	`memset(source, ‘C’, 100-1);`
`source[100-1] = ‘\0’;`	`source[100-1] = ‘\0’;`
`strcpy(destination, source);`	`g_strlcpy(destination, source,`
	`     sizeof(destination));`

The first step, line five in Figure 4.1, is to calculate the length of the buffer being written to. This step is described in Section 4.4. In the example, the length of `destination` can be calculated by the `sizeof` function, `sizeof(destination)`, because it is an array.

The next two steps, lines six and seven in Figure 4.1, are to add the length as a new parameter to the function call, and change the function call's name. This can be seen in the last line of the example's After column.

I chose to replace with the glib functions because they best match the behavior of the original functions. Some of the other safe alternatives, shown in Table 2.1, have different behavior than their unsafe originals, other than simply avoiding buffer overflows. The `astrcpy` function, for example, may alter the buffer being written to to avoid overflow. This could have unexpected consequences elsewhere in the program. The `g_strlcpy` function, on the other hand, has the exact same behavior as `strcpy`, except that it truncates the copying of any bytes beyond the limit it is given. This does not change the length or location of the buffer. If a copy is truncated, the program may behave in a way that is not useful, but it will not crash. Attackers will not be vindicated by crashing the program, and other users will be discouraged from bad input by getting useless results.

4.2 Replacing `memcpy`

Unlike the other supported functions, `memcpy` already takes a length parameter that specifies exactly how many bytes to copy. Its destination buffer can still be overflowed if the length parameter is larger than the length of the destination, however. This can happen because of developer error. ISO/IEC 24731 describes a safe alternative to memcpy that takes an additional length parameter, which limits how many bytes can be copied. Another way to 'replace' memcpy, without the use of a new library, is to add a comparison between the length of the buffer being written to and the given length parameter, so that the smaller of the two is always used as the length to copy. This can be done with a ternary expression of the form
`bufferLength > givenLength ? givenLength : bufferLength`. Table 4.2 shows a simple example of replacing `memcpy`.

The first step (line eleven in Figure 4.1) is to calculate the length of the buffer being written to. This step is described in Section 4.4. In the example, the length of `destination` can be calculated by the `sizeof` function, `sizeof(destination)`.

The next step (line twelve in Figure 4.1) is to determine if the length parameter can be assigned to. This is done because sometimes the length parameter to a `memcpy` call is later used to null terminate the buffer that was written to. This can be seen in the last line of the example. If the Safe Library Replacement transformation causes a different length to

Table 4.2: Simple `memcpy` replacement example

Before	**After**
`char destination[50];`	`char destination[50];`
`char source[100];`	`char source[100];`
`unsigned int len = 75;`	`unsigned int len = 75;`
`memcpy(destination, source, len);`	`len = sizeof(destination) > len ?`
`destination[len+1] = ‘\0’;`	`    len :  sizeof(destination);`
	`memcpy(destination, source, len);`
	`destination[len+1] = ‘\0’;`

be used in the `memcpy` call, then the null terminator might be in the wrong place. To fix this the algorithm adds an assignment to the length parameter before the `memcpy` call, if it can, so the actual length used in the `memcpy` call will be read in later lines. Otherwise, it directly replaces the length parameter.

If the length parameter can be assigned to, the next step (line thirteen in Figure 4.1) is to add an assignment to the length parameter before the `memcpy` call. The value assigned is the result of the ternary expression described before. The example shows this before the `memcpy` call in the After column. As a result of the addition of this line, the null terminator will be assigned at the right location in the last line of the After column. The `memcpy` call is left unchanged.

If the length parameter cannot be assigned to, the next step (line fifteen in Figure 4.1) is to directly change the length parameter to the ternary expression.

4.3 Replacing `gets`

I use `fgets`, another function from the standard libraries, to replace `gets`. These two functions differ more than the other supported functions do from their safe alternatives. The `fgets` function takes two additional parameters. One is a length to limit how many bytes are placed in the buffer being written to, and the other is a `FILE` pointer to read from. The `gets` function always reads from `stdin`, so `fgets` can match that by placing `stdin` as its

FILE pointer argument. As in the replacements described above, the length argument must be the length of the buffer being written to. Table 4.3 shows a simple example of replacing `gets`.

Table 4.3: Simple `gets` replacement example

Before	**After**
`char destination[50];`	`char destination[50];`
`gets(destination);`	`gets(destination, sizeof(destination),`
	`     stdin);`
	`char *check = strchr(destination, ‘\n’);`
	`if (check) {`
	`  *check = ‘\0’;`
	`}`

The first step (line nineteen in Figure 4.1) is to calculate the length of the buffer being written to. This step is described in Section 4.4. In the example, the length of `destination` can be calculated by the `sizeof` operator, `sizeof(destination)`.

The next two steps (lines twenty and twenty-one in Figure 4.1) are to add the new `FILE` and length parameters and change the name of the function call. This can be seen in the second line of the After column of the example.

The next step (line twenty-two in Figure 4.1) is to add a few new lines after the `gets` call. These are added because the behavior of `fgets` is not exactly the same as `gets`. The `fgets` function includes a terminating newline character, if there is one. The `gets` function never includes a terminating newline character. This problem can be resolved by adding a few lines to remove the terminating newline, if it is there. These can be seen in the After column of the example after the `fgets` call. The `check` pointer's name is selected by a utility that returns a name that doesn't conflict with any existing variables.

4.4 Determining Buffer Length

Calculating the length of the buffer being written to depends on how the buffer is given to the unsafe function call. Many different C expressions can be given as the buffer parameter. For example, the buffer can be a simple identifier like `ptr`, a pointer arithmetic expression like `ptr + 10`, or a function call that returns a pointer like `getBuffer()`. Figure 4.2 shows an outline of the buffer length algorithm.

Different steps are taken for each type of expression the buffer may be given as. The exact steps to calculate the buffer length for the expressions assignment, array access, binary, cast, identifier, element access, and prefix are described in the Subsections following this Section.

In every case, one of two functions is used in some way to determine the size of buffers: `sizeof` or `malloc_usable_size`. The C `sizeof` operator returns the size, in bytes, of any variable on the stack. The `malloc_usable_size` function return the length, in bytes, of a block of allocated heap memory.

The stack is an ordered block of memory for the program. When a function is called, memory for the parameters and local variables of that function is allocated on the stack. It is then deallocated when the function is complete. A C compiler determines how much memory to allocate and deallocate for each function. Therefore, the size of stack variables must be known before the program is compiled, and their size cannot be changed. The `sizeof` operator can be used to determine the size of any array or struct at any time during program execution. It cannot be used on pointers that point to a buffer, however, because it will return the size of the pointer variable itself, which depends on the size of addresses on the machine the C program is compiled on.

Determining the length of a buffer pointed to by a pointer requires some knowledge of what the pointer points to. If it points to an array or a struct, then the `sizeof` operator can be used. If it points to heap memory, then the `malloc_usable_size` function might be used. The heap is an unordered block of memory for the program. It is used when

Figure 4.2: Buffer length algorithm

```
BufferLength(buffer):

   if (buffer is assignment expression)
      - return buffer length of the right side of
        the assignment expression

   else if (buffer is array access expression)
      if (type of the array is array)
         - return a sizeof expression
      - stop

   else if (buffer is binary expression)
      if (binary op is + or -)
         - new op is + for -, or - for +
         - get buffer length of the left side of the
           binary expression
         - return left-side-length newOp right-side
      - stop

   else if (buffer is cast expression)
      - return buffer length of the expression
        being cast

   else if (buffer is identifier expression)
      if (type of the identifier is array)
         - return a sizeof expression
      if (type of the identifier is pointer)
         if (identifier is aliased)
            - stop
         else
            - get most-recent-definition of buffer
            if (most-recent-def is heap allocation assignment)
               - return malloc_usable_size expression
            else if (most-recent-def is assignment)
               - return buffer length of the right
                 side of the assignment
      - stop
```

```
else if (buffer is element access expression)
   if (type of element is array)
      - return a sizeof expression
   if (type of element is pointer)
      if (element’s struct is aliased)
         - stop
      else
         - get most-recent-definition of buffer
         if (element’s struct is defined after most-recent-def)
            - stop
         if (most-recent-def is heap allocation assignment)
            - return a malloc_usable_size expression
         else if (most-recent-def is assignment)
            - return buffer length of the right
              side of the assignment
   - stop

else if (buffer is prefix expression)
   if (prefix op is &)
      if (inner expression type is struct)
         - return a sizeof expression

else
   - stop
```

a program needs a block of memory whose size cannot be known at runtime. Allocation functions are called when a block of heap memory is needed. The allocation function is given the number of bytes needed, and returns a pointer to the first byte of an unallocated block of memory somewhere in the heap that is at least as large as the requested number of bytes. The `malloc_usable_size` function can be used to get the length, in bytes, of a block of allocated heap memory. However, it must be given a pointer to the first byte of the heap buffer to work correctly. Its behavior is undefined if given a pointer to any other location, even locations within the buffer.

4.4.1 Length Of Buffer From An Assignment Expression

The step to determine the length of a buffer from an assignment expression is line four in Figure 4.2. The return value of an assignment expression in C is the value of what is assigned. Therefore, if the buffer being written to is given to the unsafe function as an assignment expression, its size can be determined by recursively determining the size of whatever is on the right side of the assignment expression. A simple example is shown in Table 4.4.

Table 4.4: Assignment replacement example

Will overflow	**Will not overflow**
`char buff[50];`	`char buff[50];`
`char source[100];`	`char source[100];`
`memset(source, 'C', 100-1);`	`memset(source, 'C', 100-1);`
`source[100-1] = '\0';`	`source[100-1] = '\0';`
`char *dest;`	`char *dest;`
`strcpy(dest = buff, source);`	`g_strlcpy(dest = buff, source,`
	`sizeof(buff));`

In the example, the buffer is given to `strcpy` in the Before column as an assignment of an array to a pointer. The length of the buffer can be determined by calculating the length of the right side of the assignment, which is `buff`. `buff` is an identifier, so its length is determined by the process described in Section 4.4.5 on identifiers. The size of an array can be determined by the `sizeof` function. The After column in the example shows `sizeof(buff)` added as the length parameter to `g_strlcpy`.

4.4.2 Length Of Buffer From An Array Access Expression

The steps to determine the length of a buffer from an array access expression are lines eight to ten in Figure 4.2. If the buffer is given as an array access expression, the buffer is inside an array. Determining the size of a buffer inside an array depends on the type of the array. If the buffer is an array within an array, then the size can be determined by using `sizeof` on the array access expression at the same index. Table 4.5 shows a simple example.

Table 4.5: Array inside array replacement example

Will overflow	**Will not overflow**
`char dest[3][50];`	`char dest[3][50];`
`char source[100];`	`char source[100];`
`memset(source, 'C', 100-1);`	`memset(source, 'C', 100-1);`
`source[100-1] = '\0';`	`source[100-1] = '\0';`
`strcpy(dest[1], source);`	`g_strlcpy(dest[1], source,`
	`sizeof(dest[1]));`

In the example, the buffer given to `strcpy` in the Before column is an array inside an array at index one. The length of that inner array can be calculated with the `sizeof` function at the same index. This can be seen added as the length parameter to `g_strlcpy` in the After column of the example.

Another possibility is that the buffer is a pointer within an array. In that case, it could be possible to use `malloc_usable_size` in the same way `sizeof` is used on array types, but only if the pointer at that index points to the first byte of a heap buffer. To decide if the pointer points to such a location, that pointer would need to be analysed in the same way identifiers are analysed in Section 4.4.5 below. The problem is that the exact index of every array access on the array containing the pointer would need to be known. Knowing the index of every array access on the pointer array would require difficult integer analysis that

OpenRefactory/C does not have. Therefore, if the algorithm finds that the buffer is an array access on a pointer array, it stops.

It is also possible to give an integer inside an array as a pointer as well. The value of the integer becomes the value of the address the pointer points to. The algorithm stops if it finds an array access on a integer array. It cannot be known from static analysis what a pointer points to if it is cast from an integer.

4.4.3 Length Of Buffer From A Binary Expression

The steps to determine the length of a buffer from a binary expression are lines thirteen to eighteen in Figure 4.2. Binary expressions are formed from C's binary operators, like `+` or `>`. Buffers usually appear in the form of a binary expression when some offset is being added to the pointer. In every case a buffer appeared as a binary expression in the code I evaluated Safe Library Replacement transformation on (see Chapter 6), the actual buffer was on the left of the expression, and some offset was added or subtracted on the right. As a result, I make an assumption that the left side of the binary expression is the buffer and the right side an offset. The length of the buffer is then determined by recursively determining the length of the left side of the binary expression, and adding or subtracting the offset to that result. The algorithm stops if it gets a binary expression with an operator other than `+` or `-`. Table 4.6 shows an example that exemplifies the use of recursion.

Table 4.6: Binary replacement example

Will overflow	**Will not overflow**
`char dest[50];`	`char dest[50];`
`char source[100];`	`char source[100];`
`memset(source, ‘C’, 100-1);`	`memset(source, ‘C’, 100-1);`
`source[100-1] = ‘\0’;`	`source[100-1] = ‘\0’;`
`strcpy((dest + offset) - 5, source);`	`g_strlcpy(dest + 5, source,`
	`(sizeof(dest) - offset) + 5);`

In the example, the buffer is given to `strcpy` in the Before column as an array, plus some offset, minus a constant. The outer binary expression has an inner binary expression on the left and a constant on the right. The algorithm first recursively determines the buffer length of the inner binary expression on the left. The inner binary expression has an identifier on the left and another identifier on the right. It again recursively determines the length of the left side. This is done by the identifier procedure in Section 4.4.5, and results in `sizeof(dest)`. The operator for the inner binary is swapped and then the right side subtracted from the length. The algorithm goes back up to the outer binary expression where the same thing is done. The operator is swapped and the right side is added to the length that came up from the left side recursive call. The final result can be seen added as the length parameter to `g_strlcpy` in the example's After column.

4.4.4 Length Of Buffer From A Cast Expression

The step to determine the length of a buffer from a cast expression is line twenty-one in Figure 4.2. Similar to assignment expressions from Section 4.4.1, determining the size of a cast expression can be done by recursively determining the size of the expression being cast. Table 4.7 shows a simple example.

Table 4.7: Cast replacement example

Will overflow	**Will not overflow**

`char dest[50];`
`char source[100];`
`memset(source, ‘C’, 100-1);`
`source[100-1] = ‘\0’;`
`strcpy((char *)dest, source);`
	`char dest[50];`
	`char source[100];`
	`memset(source, ‘C’, 100-1);`
	`source[100-1] = ‘\0’;`
	`g_strlcpy((char *)dest, source,`
	`     sizeof(dest));`

In the example, the buffer is given to `strcpy` as a cast of an array to a char pointer. To determine the length of the buffer, the cast is peeled off to reveal an identifier. The length of the identifier is determined by the procedure described in Section 4.4.5 on identifiers. The identifier is an array, so its length can be calculated by the `sizeof` expression. The addition of the length parameter can be seen in `g_strlcpy` in the After column.

4.4.5 Length Of Buffer From An Identifier Expression

The steps to determine the length of a buffer from an identifier expression are lines twenty-five to thirty-seven in Figure 4.2. An identifier expression is simply a variable name. Calculating the size of a buffer from an identifier depends of the type of the variable. If it is an array then `sizeof` can be used on the identifier. If it is a pointer, then `malloc_usable_size` may be used on it, but only if the pointer points to the first block of a heap buffer. It is difficult to determine what a pointer points to through static analysis alone, and in many cases it is impossible to be sure about anything. However, the capability to determine the size of at least some pointers is critical to the usefulness of Safe Library Replacement transformation. Pointers are frequently used with the unsafe functions I implemented, and a pointer often is the base case of the recursion of many of the other expression types. Table 4.8 shows a simple example.

Table 4.8: Pointer replacement example

Will overflow	**Will not overflow**
`char *dest = (char*)malloc(10);`	`char *dest = (char*)malloc(10);`
`char source[100];`	`char source[100];`
`memset(source, 'C', 100-1);`	`memset(source, 'C', 100-1);`
`source[100-1] = '\0';`	`source[100-1] = '\0';`
`strcpy(dest, source);`	`g_strlcpy(dest, source,`
	`    malloc_usable_size(dest));`

The algorithm takes a conservative approach by trying to decide if a pointer must *always* point to the start of a heap buffer, or if it *may not* point to the start of a heap buffer. It does this by first performing OpenRefactory/C's alias analysis on the pointer. If the pointer is aliased, the algorithm stops, because the pointer may be changed be through its aliases.

It may be possible to investigate all the aliases as well. In the example, the buffer being written to, `dest`, is not aliased.

If the pointer is not aliased, the algorithm continues by investigating the most recent definition of the pointer before its use in the unsafe function call. The most recent definition is found by OpenRefactory/C's reaching definition analysis. The simplest case is if the definition is an assignment to the result of a heap allocation function. In this case, the algorithm knows `malloc_usable_size` can be used safely. In the example, the most recent definition of `dest` is a declaration initialization to a call to `malloc`. `malloc` is a heap allocation function, so `malloc_usable_size` can be used. The addition of `malloc_usable_size` as the length parameter to `g_strlcpy` can be seen in the After column of the example.

If the most recent definition is an assignment to any other expression, the algorithm recursively determines the size of the assigned expression. Table 4.9 shows a simple example.

Table 4.9: Reassigned pointer replacement example

Will overflow	**Will not overflow**
`char buff[50];`	`char buff[50];`
`char source[100];`	`char source[100];`
`memset(source, ‘C’, 100-1);`	`memset(source, ‘C’, 100-1);`
`source[100-1] = ‘\0’;`	`source[100-1] = ‘\0’;`
`char *dest = (char*)buff;`	`char *dest = (char*)buff;`
`strcpy(dest, source);`	`g_strlcpy(dest, source,`
	`sizeof(buff));`

In the example, the most recent definition of `dest` is an assignment to a cast expression. The cast expression's length is recursively determined, as is described in Section 4.4.4 on casts. Inside the cast expression is another identifier, `buff`. Its length is recursively determined as described in Section 4.4.5. `buff` is an array, so its length can be calculated with the `sizeof` expression. The addition of this length can be seen in `g_strlcpy` in the After column.

OpenReafactory/C's reaching definition analysis only finds definitions within the scope of a function. It is possible that there not be a definition of the pointer in the local function before it is used in the unsafe function call. This can happen if the pointer comes in as a parameter, or if it is declared and never defined. If there is not a definition of the pointer before the unsafe function call, then the algorithm stops. If the pointer comes in as a parameter, it may be possible to look at all the function calls of that function in the C program to investigate what the pointer points to at those points. If all those pointers are similar enough to have the same length calculation done on them, then the unsafe function could be replaced.

4.4.6 Length Of Buffer From An Element Access Expression

The steps to determine the length of a buffer from an element access expression are lines forty-one to fifty-five in Figure 4.2. Element access expressions are accesses of an element in a struct. Their size is determined in the exact same way as identifiers from Section 4.4.5, except with two differences. One is that the alias analysis is done on the struct, not the element. The reason for this is that the alias analysis treats a struct and its elements like one object. If one of its elements is aliased, then the whole struct is considered aliased.

The other 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. If the entire struct is redefined, then the element is redefined as well. The algorithm will not replace if this is found. Table 4.10 shows a simple example of a successful replacement.

In the example, `destStruct.p` is a pointer. `destStruct` is not aliased or redefined. `destStruct.p`'s most recent definition is an assignment to a `malloc` call. As a result, `malloc_usable_size` can be used on `destStruct.p`. The addition of this length can be seen in `g_strlcpy` in the After column on the example.

Table 4.10: Element access replacement example

Will overflow	**Will not overflow**
`structType destStruct;`	`structType destStruct;`
`destStruct.p = (char*)malloc(10);`	`destStruct.p = (char*)malloc(10);`
`char source[100];`	`char source[100];`
`memset(source, ‘C’, 100-1);`	`memset(source, ‘C’, 100-1);`
`source[100-1] = ‘\0’;`	`source[100-1] = ‘\0’;`
`strcpy(destStruct.p, source);`	`g_strlcpy(destStruct.p, source,`
	`     malloc_usable_size(destStruct.p));`

4.4.7 Length Of Buffer From A Prefix Expression

The steps to determine the length of a buffer from a prefix expression are lines fifty-eight to sixty in Figure 4.2. A prefix expression is formed by one of C’s prefix operators. Most prefix expressions do not make sense as a buffer. There is one special case that was seen in the Safe Library Replacement transformation evaluation (see Chapter 6), however. If the prefix is the reference operator `&` and the variable referenced is a struct, then the size of the buffer can be calculated by calling `sizeof` on the struct. Table 4.11 shows a simple example.

Table 4.11: Struct reference replacement example

Will overflow	**Will not overflow**
`structType destStruct;`	`structType destStruct;`
`char source[100];`	`char source[100];`
`memset(source, ‘C’, 100-1);`	`memset(source, ‘C’, 100-1);`
`source[100-1] = ‘\0’;`	`source[100-1] = ‘\0’;`
`strcpy(&(destStruct), source);`	`g_strlcpy(&(destStruct), source,`
	`     sizeof(destStruct));`

In the example, `destStruct` is a struct. A pointer to that struct is given to `strcpy` as the ‘buffer’. The length of that ‘buffer’ is simply the size of the struct. This can be calculated by using `sizeof` on the struct. The addition of this length to `g_strlcpy` can be seen in the After column of the example.

4.4.8 Length Of Buffers From All Other Expressions

The step for handling a buffer from any other expression is line sixty-three in Figure 4.2. Other expressions that might be given to the unsafe function as the buffer being written to are function calls, ternary expressions, postfix expressions, and constants. In the case of function calls, it may be possible to investigate what the function returns. In the case of ternary expressions, the result of its boolean expression must be known to know which branch of the ternary to investigate. The result of boolean expressions is difficult to determine through static analysis. In the case of postfix expressions, only the `++` and `--` operators make sense for a buffer. These could be handled similar to how binary expressions are handled in Section 4.4.3. I did not see postfix expressions as buffers in the SAFE LIBRARY REPLACEMENT transformation evaluation (see Chapter 6), however. In the case of constants, the constant will automatically be cast to a pointer. When an integer is cast to a pointer, it is impossible to know what it points to. If SAFE LIBRARY REPLACEMENT transformation finds any of these expressions as the destination buffer, it stops.

Chapter 5
Alias Analysis

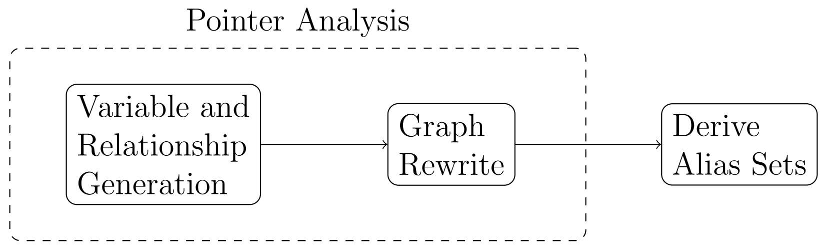
An alias analysis produces a set of variables and dereferences that may alias some variable. For example, if a pointer `p` points to an integer `i`, then the dereference of `p`, `*p`, is an alias of `i`. A variable or dereference aliases some other variable if a change to the value of that variable or dereference may also change the value of the variable it aliases. The Safe Library Replacement transformation needs an alias analysis to determine if a pointer it is analyzing is aliased (see Section 4.4.5). If a pointer is aliased, then it may be altered through its aliases. To accurately analyze what the pointer may point to, any aliases it may have must also be analyzed. Safe Library Replacement transformation stops when it finds a pointer that is aliased, instead of trying to check all the aliases. The transformation cannot be completed if there is any risk of the pointer changing in unexpected ways.

5.1 Algorithm For Alias Analysis

Figure 5.1 illustrates the three major steps in the alias analysis. The alias analysis is actually backed by a pointer analysis. A pointer analysis produces a graph of which variables in a C program may point to other variables in the C program. Alias sets can be derived from the pointer analysis results by propagating aliases down the edges of the graph. The alias analysis runs the pointer analysis and then generates alias sets from the results.

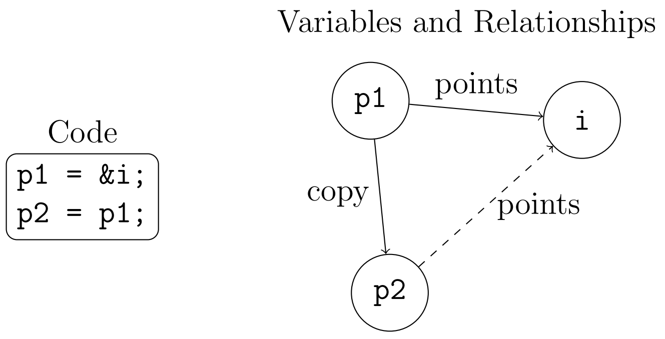
The pointer analysis algorithm has two major phases. The first phase is to record information about what variables are in a C program, and the relationships between them. Figure 5.2 shows an example of the variables and relationships produced from two lines of code. The algorithm makes a single pass through the code to be analysed, and records variables and relationships for each line of code. The recorded variables and relationships

Figure 5.1: High level alias analysis process



create a graph with the variables as nodes and the relationships as edges between them. In the example, the first line of code, `p1 = &i;`, adds the variables `p1` and `i` and a *points* relationship between them. The second line of code, `p2 = p1;`, adds the variable `p2` and a *copy* relationship between it and `p1`.

Figure 5.2: Abstract alias analysis example



```
p1 = &i;
p2 = p1;
```

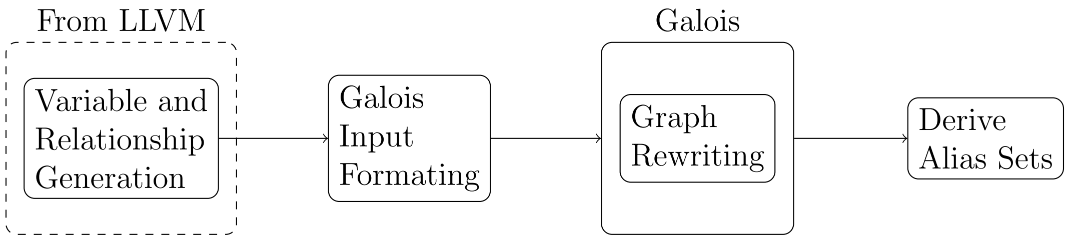
The second phase of the pointer analysis is to rewrite the graph to determine more accurate points-to information. One of the rewrite rules is that any node with a *copy* edge from another node should should have a *points* edge to all the nodes that the node it is copying has *points* edges to. The example shows the addition of such a *points* edge with a dotted arrow. After the entire graph is rewritten, the points-to graph can be read from the relationship graph by looking at the *points* edges.

The final step of the alias analysis algorithm is to derive alias sets from the points-to graph. To propagate aliases down the edges of the points-to graph efficiently, a topological sort is done on the graph. Then, in topological order, a dereference of each variable, and dereferences of any aliases that may have been pushed down earlier, is pushed down as an alias of all the variables it points to.

5.2 Alias Analysis Implementation For OpenRefactory/C

Figure 5.3 shows the parts of OpenRefactory/C's alias analysis implementation. When implementing alias analysis for OpenRefactory/C, I followed the example of a C++ pointer analysis implemented by Hardekopf [13] for the LLVM compiler infrastructure [21]. Hardekopf's implementation uses the algorithm described in Section 5.1. I reimplemented the variable and relationship generation phase from Hardekopf's implementation in Java for OpenRefactory/C.

Figure 5.3: Alias analysis implementation structure



Instead of reimplementing the graph rewriting phase, I directly integrated a Java implementation of that phase by Mendez-Lojo [26]. Mendez-Lojo's implementation is a parallelized reimplementation of the LLVM implementation's graph rewriting phase using the Galois framework [34], a framework for executing C++ or Java code in parallel. The Galois analysis takes the variables and relationships produced in the first phase as input, and does the graph rewriting phase faster than Hardekopf's implementation because it rewrites edges in the graph in parallel.

The following three Sections describe the variable and relationship generation phase, the Galois input formatting, and the alias set derivation from the Galois output.

5.3 Recording Variables And Their Relationships

The first step in the pointer analysis is to record information about what variables are in a C program and the relationships between them. Hardekopf's LLVM alias analysis does its analysis on LLVM's intermediate code representation. Figure 5.4 shows an example of a short C program on the left and its corresponding LLVM assembly code on the right. The LLVM pointer analysis records information about variables as *nodes*, and the relationships between them as *constraints*. Table 5.1 shows the *nodes* and *constraints* that would be generated from the example C program in Figure 5.4.

Figure 5.4: Alias analysis example C code
and corresponding LLVM assembly code

```
void main()
{
   int i;
   int *p1, *p2;
   p1 = &i;
   p2 = p2;
}
```

```
define void @main nounwind {
entry:
   %p2 = alloca i32*
   %p1 = alloca i32*
   %i = alloca i32
   %"alloca point" = bitcast i32 0 to i32
   store i32* %i, i32** %p1, align 4
   %0 = load i32** \%p1, align 4
   store i32* %0, i32** %p2, align 4
   br label %return

return:
   ret void
}
```

Table 5.1: Nodes and constraints generated from the example C program

Nodes	Constraints
value node:i	value node:i - address of - object node:i
object node:i	value node:p1 - address of - object node:p1
value node:p1	value node:p2 - address of - object node:p2
object node:p1	value node:p1 - store - value node:i
value node:p2	value node:null1 - load - value node:p1
object node:p2	value node:p2 - store - value node:null1
value node:null1	

There are two forms of *nodes*:

- *value nodes*
- *object nodes*

Value nodes loosely represent abstract pointers and are used only in the analysis. *Object nodes* represent the variables themselves, and are used to express the result of the analysis in a graph of *object nodes* pointing to other *object nodes*. A *value* and *object node* are always recorded for each variable in a C program, and those *nodes* remain associated with that variable. Additional 'null' *value nodes* may be added to help link relationships between *object nodes*, but there is always a one-to-one relationship between *object nodes* and variables. The example shows that a *value* and *object* node are produced for each of the three variables in the code. There is also an additional 'null' *value* node. It is produced for the LLVM variable `%0` on line 8 of the LLVM code.

There are five forms of *constraints*:

- *address-of constraint*
- *copy constraint*

- *load constraint*
- *store constraint*
- *get-element-pointer (gep) constraint.*

For the purpose of this thesis the exact meaning of each *constraint* type because their meaning is interpreted in the Galois implementation is unimportant. For more information see [13]. This stage of the analysis is only concerned with generating the correct *constraints* for each instruction in a C program. An *address-of constraint* is always recorded between the *value* and *object node* added for each variable in a C program. The example shows each of the three *address-of* constraints for each variable in the C code.

The *constraints* and *nodes* that a particular C instruction should produce cannot be determined by only reading the LLVM alias analysis. The LLVM alias analysis does its analysis on LLVM assembly code. Learning some of the LLVM assembly language and seeing how C instructions translate into LLVM assembly code was the only way to connect a C instruction with the *constraints* and *nodes* it should produce.

The declarations in lines three and four of the C code correspond to the `alloca` instructions in lines three to five of the LLVM assembly. Identifiers in LLVM assembly are preceded by the % symbol. The LLVM alias analysis adds a *value node* for each identifier in the LLVM assembly. The `alloca` instruction causes the creation of an *object node* and the addition of an *address-of constraint* between the *object node* and the *value node* associated with the identifier the `alloca` is assigned to. If the example function had parameters, there would be `alloca` instructions for each at the beginning of the function definition in LLVM assembly. This relationship between C code and LLVM assembly is the reason why the OpenRefactory/C implementation of *node* and *constraint* generation adds a *value node*, *object node*, and *address-of constraint* for each local variable declaration, function definition parameter, and global variable declaration it finds in the C program.

The `bitcast` instruction in line six of the LLVM assembly is also associated with the declarations, but it does not add any alias information, so it is ignored.

The assignment in line five of the C code corresponds to the `store` instruction in line seven of the LLVM assembly code. The LLVM alias analysis adds a *store constraint* between the *value nodes* that correspond to the two identifiers used in the `store` instruction for each `store` instruction in the LLVM assembly. Any assignment of a reference to a pointer will always correspond to a `store` instruction in LLVM assembly. Therefore, the *node* and *constraint* generation in OpenRefactory/C adds a *store constraint* for every assignment of a reference to a pointer in the C program.

The assignment in line six of the C code corresponds to the `load` and `store` instructions in lines eight and nine of the LLVM assembly code. The LLVM alias analysis adds a *load constraint* between the *value node* of the identifier it is loading and the *value node* of the identifier the load is being assigned to for each `load` instruction. `load` instructions always cause the creation of a new LLVM assembly identifier, and therefore, a new 'null' *value node*. The assignment of one variable to another will always correspond to a `load` and `store` instruction in LLVM assembly. Therefore, for every assignment of one variable to another in the C program, the *node* and *constraint* generation in OpenRefactory/C adds a new 'null' *value node* for the new LLVM identifier, a *load constraint* between the *value node* of the variable being assigned and the new *value node*, and a *store constraint* for the `store` instruction. Assignment of non-pointers to non-pointers is specifically ignored.

The `return` instructions in lines ten, twelve, and thirteen of the LLVM assembly are there automatically, even though the C program does not have a return statement. Any actual return statements in the C program would result in similar LLVM instructions. Interprocedural analysis is done in the LLVM alias analysis, but it is omitted from the OpenRefactory/C alias analysis for now. The OpenRefactory/C alias analysis simply ignores function calls and return statements.

Aggregate variables like structs, arrays, and unions are treated like one object in the OpenReafactory/C *node* and *constraint* generation. Any time an element access on these types of variables is encountered, the element is treated as the entire structure. For example, in an assignment like `myStruct.myElement = myVar`, the element access `myStruct.myElement` would be associated with the *value* and *object nodes* for `myStruct` itself.

The *node* and *constraint* generation phase is complete when all global variables, function definitions, and their instructions have been processed.

Only a few of the many forms of instructions that can be written in C are shown in the example here. There are other forms of instructions that the OpenRefactory/C alias analysis generates *nodes* and *constraints* for, but many C instructions obviously have no bearing on aliases and can be ignored without consideration. There are some instructions that the LLVM alias analysis generates *nodes* and *constraints* for, but that OpenRefactory/C does not. Greater coverage of more obscure instructions will be added to OpenRefactory/C's alias analysis in the future, such as support for inter-procedural analysis.

5.4 Galois Pointer Analysis Input

The next stage of the alias analysis is to format the *nodes* and *constraints* for input to the Galois pointer analysis. The Galois pointer analysis reads from two files for its input, a file for *nodes* and a file for *constraints*. Each file has a specific format, shown in Table 5.2.

The node file starts with the total number of *nodes* the file contains. The next line gives the id of the last *object node* in the file. The next line gives the id of the last *node* with its flag set to true. Every node in the LLVM alias analysis has a boolean flag associated with it. It was not mentioned before because I do not currently understand the purpose of the flag. It is not set to true in any of the *node* generation code I translated to OpenRefactory/C from the LLVM alias analysis. OpenReafactory/C currently only generates *nodes* whose flag is false. As a result, the last flaged *node* id in the node file is currently always 0.

Table 5.2: Galois pointer analysis node and constraint file format

Node file format	Constraint file format
# of nodes	# of constraints
id of last object node	0,2,1,0,0
id of last flagged node	id,dst,src,type,offset
0,0,0	...
1,1,0	
2,0,0	
id,val/obj,flag	
...	

Each subsequent line in the node file specifies a single *node*. Each *node* line starts with the *node's* id, then 0 for *value nodes* or 1 for *object nodes*, and finally 0 for a false flag or 1 for a true flag. The *nodes* must always appear in order of their ids. There are always three special *nodes* at the top of the file. The first is a blank *node*. The second is an *object node* that represents an 'unknown' variable. This *node* is used for instructions like integer to pointer casts, where it is not known what the pointer may point to. The third is a *value node* that always points to the 'unknown' *object node*. After the first three special *nodes*, there may follow any number of *value* and *object nodes*. However, the *object nodes* must all come before the *value nodes*. I do not understand why this is the case, but the Galois alias analysis will not run otherwise.

The constraint file also starts with the total number of *constraints* the file contains. Each subsequent line in the constraint file specifies a single *constraint*. Each *constraint* line starts with the *constraint's* id, followed by the ids of the destination and source *nodes*, followed by the type of the *constraint*, and finally an offset. The type is given by 0 for *address-of*, 1 for *copy*, 2 for *load*, 3 for *store*, and 4 for *gep*. Like the flag field in the *nodes*, I do not clearly understand the purpose of the offset. It is probably used when dealing with aggregate variables like structs and arrays, but OpenRefactory/C's alias analysis treats these structures as a whole, so the offset is always zero. The *constraints* must always appear in order of their ids. There is always a special constraint at the top of the file. It adds an *address-of* constraint from the special 'unknown' *value node* to the special 'unknown' *object node*.

Table 5.3 shows the *node* and *constraint* files that would be produced for the example *nodes* and *constraints* from Table 5.1. Notice that the *nodes* have been rearranged so that all the *object nodes* come first. The last flagged node value is set to 0 because there are no flagged nodes in the file.

Table 5.3: Example Galois pointer analysis node and constraint file

Node file	Constraint file
10	7
5	0,2,1,0,0
0	1,3,6,0,0 (i:val - address of - i:obj)
0,0,0	2,4,7,0,0 (p1:val - address of - p1:obj)
1,1,0	3,5,8,0,0 (p2:val - address of - p2:obj)
2,0,0	4,6,7,3,0 (p1:val - store - i:val)
3,1,0 (i:obj)	5,7,9,2,0 (null1:val - load - p1:val)
4,1,0 (p1:obj)	6,9,8,3,0 (p2:val - store - null1:val)
5,1,0 (p2:obj)	
6,0,0 (i:val)	
7,0,0 (p1:val)	
8,0,0 (p2:val)	
9,0,0 (nul1:val)	

Normally the filenames of the node and constraint files are given as command line arguments to the Galois alias analysis, and Galois reads the files itself. OpenRefactory/C directly calls Galois' main method and, instead of giving the two filenames, it gives the entire contents of the two files as arguments, formatted correctly from the *nodes* and *constraints* generated in the first stage. The Galois method that parses the files was modified slightly so that it directly parses the two arguments, instead of using them to open files.

5.5 Galois Pointer Analysis Output

The final stage of the alias analysis is to get the result of the analysis from Galois and associate it with the correct variables in the C program. The points-to results for the running example are shown in Table 5.4.

Table 5.4: Example pointer analysis results

Object node	Points-to list
i - 3	[]
p1 - 4	[i - 3]
p2 - 5	[i - 3]

The Galois pointer analysis thinks only in id numbers, so an association between the ids set in the Galois pointer analysis input and the actual variables they correspond to is kept in memory. The results are taken from the internal representation of *object nodes* in the Galois alias analysis. Each internal Galois *object node* will have a, possibly empty, list of ids that that *object node* points to. The ids are translated back to variables using the stored association, and a hash map is used to record the points-to list of each *object node*. Reverse points-to information can be derived from the points-to graph by following edges backwards.

The final step of the alias analysis is to derive alias sets from the points-to results. The alias results for the running example are shown in Table 5.5.

Alias information is kept as a list of aliases. I consider an *alias* a combination of a variable and a number of dereferences. For example, if pointer `p` points to integer `i`, then

Table 5.5: Example alias analysis results

Object	Alias set
i	[i, *p1, *p2]
p1	[p1]
p2	[p2]

`i` has aliases {`i` and `*p`}, where `i` is an *alias* with zero dereferences and `*p` is an *alias* with one dereference. To propagate alias information down the points-to graph efficiently a topological sort is done on the graph. Then, in topological order, a dereference of each variable, and dereferences of any aliases that may have been pushed down earlier, is pushed down as an alias of all the variables it points to. It is possible for an aggregate variable like a stuct or array 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. The alias sets are stored in a hash map like the points-to sets for efficient access.

Chapter 6
Evaluation Of Safe Library Replacement Transformation Implementation

The implementation of the SAFE LIBRARY REPLACEMENT transformation in OpenRefactory/C was evaluated on two test corpuses. The first was the Juliet Test Suite for C/C++ from the SAMATE Reference Dataset [32]. The second was comprised of three real open source C programs: zlib, libpng, and gmp. In each test case there were three possible results: pass-complete, did-not-replace, and fail. A case passes-complete if it replaces the unsafe function, removes the buffer overflow vulnerability, and does not change any other behavior. A case did-not-replace when the SAFE LIBRARY REPLACEMENT transformation determines that it cannot safely do the replacement. A case fails if the SAFE LIBRARY REPLACEMENT transformation does the replacement but causes errors elsewhere in the program.

6.1 SAMATE Test Suite

The Juliet Test Suite contains 61,387 test cases on a wide variety of bugs and weaknesses categorized as Common Weakness Enumerations (CWEs). To identify the cases relevant to the SAFE LIBRARY REPLACEMENT transformation, all test cases that do not contain any of the unsafe functions that SAFE LIBRARY REPLACEMENT transformation supports were removed. That substantially reduced list of cases was then manually investigated to see which cases contained a buffer overflow from one of SAFE LIBRARY REPLACEMENT transformation's six supported functions. This resulted in a list of 1,778 test cases from CWEs: 121–Stack Based Buffer Overflow, 122–Heap Based Buffer Overflow, and 242–Use of Inherently Dangerous Function.

The test procedure for each case was to run the Safe Library Replacement transformation on the unsafe function that caused a buffer overflow, and then compile and run the program to see if it was corrected. *All 1,778 test cases passed-complete.*

6.2 Real C Programs

The Safe Library Replacement transformation implementation was also evaluated on three real C programs: zlib-1.2.5, libpng-1.2.6, and gmp-4.3.2. The test procedure for an individual use of a supported unsafe function was to run Safe Library Replacement transformation on that function, then compile the program, then run the internal tests cases for the program to see if it had been broken. This test procedure was performed on every use of the six supported unsafe functions in the three C programs. Table 6.1 shows the results.

73% of the supported functions in the real-world C programs could be replaced without breaking the program. None of the test cases broke the program by replacing the function. The Safe Library Replacement transformation implementation could not replace the function with certainty in all the other cases. The overwhelming majority of the could-not-replace cases were uses of `memcpy`. The reason is that `memcpy` is used in a much wider variety of ways than the other functions. The following Section discusses each reason Safe Library Replacement transformation could not replace a function, and gives an example from the C programs for each.

The `gets` function is not used in any of the C programs we tested on. The reason is that they are all libraries, and therefore have no need for user input. Testing on programs with user interaction will be done in the future to better test the replacement of `gets`.

6.2.1 Reasons For Not Replacing A Function

Every reason Safe Library Replacement transformation decided it could not correctly replace the function was the result of investigating a pointer that points to the buffer being written to.

Table 6.1: Safe Library Replacement transformation evaluation results for real C programs

Function	Uses	Pass Complete	Cannot Replace	Fail (Break Program)
		zlib		
strcpy	11	5	6	0
strcat	4	4	0	0
sprintf	1	1	0	0
memcpy	25	7	18	0
TOTAL	41	17 **(41%)**	24	0
		libpng		
strcpy	6	6	0	0
sprintf	18	18	0	0
memcpy	55	40	15	0
TOTAL	79	64 **(81%)**	15	0
		gmp		
strcpy	6	6	0	0
strcat	3	3	0	0
sprintf	21	20	1	0
vsprintf	2	1	1	0
memcpy	29	22	7	0
TOTAL	61	52 **(85%)**	9	0
		function totals		
strcpy	23	17	6	0
strcat	7	7	0	0
sprintf	40	39	1	0
vsprintf	2	1	1	0
memcpy	109	69	40	0
TOTAL	**181**	**133 (73%)**	**48**	**0**

Functions not in the table were not used in the program(s).

Reason 1: Local buffer pointer is not set equal to a heap allocation function.

Twenty-two unsafe function uses were not replaced for this reason. It is the most common. In this case, the destination buffer in the unsafe function call is a local pointer that is not set to the result of a heap allocation function between the point that it is declared and where it is used in the unsafe function call. In some cases the pointer is part of a local struct, but the reasoning is the same. It is not certain that the pointer points to the start of a block of heap memory, so the `malloc_usable_size` function cannot safely be used on it.

The only strategy I am aware of that might work to resolve this case is to keep track of the size of whatever the pointer points to. Safe Type Replacement transformation may be appropriate for attempting to remove the buffer overflow vulnerability in these cases.

Figure 6.1 shows an example of this case from zlib. The `memcpy` has a char pointer as its destination buffer that is declared locally. The destination pointer is never set equal to the result of a heap allocation function, which invalidates the use of `malloc_usable_size`.

Figure 6.1: No-replace example from zlib: infback.c line 338

```
int ZEXPORT inflateBack(strm, in, in_desc, out, out_desc)
...
{
   ...
   unsigned char *next;
   unsigned char *put;
   unsigned copy;
   ...  // put is not assigned to a heap allocation function in these lines
   memcpy(put, next, copy);
   ...
}
```

Reason 2: Parameter buffer pointer is not locally set equal to a heap allocation function.

Nineteen unsafe function uses were not replaced for this reason. In this case, the buffer being written to in the unsafe function call is a pointer parameter to the function definition in which the unsafe function call is found. The pointer is not set to the result of a heap allocation function between the the start of the function and where it is used in the unsafe function call. Sometimes the pointer is part of a struct parameter, but the reasoning is the same. It is not certain that the pointer points to the start of a block of heap memory, so the `malloc_usable_size` function cannot safely be used on it.

It might be possible to investigate how the buffer is defined before it is given to function calls of the function in which it is used unsafely. That would require that every use of the function be investigated. It is unlikely that `malloc_usable_size` could be used on the pointer in every case.

Using Safe Type Replacement transformation to keep track of the size of the buffer would also be difficult. Safe Type Replacement transformation would have to replace the type of the parameter to the function definition, which would require replacing the type of every variable that is used as a parameter to that function. This case is difficult to overcome.

Figure 6.2 shows an example of this case from zlib. The `strcpy` has a char pointer as its destination buffer that comes in as a parameter to the `test_compress` function. The destination pointer is never set equal to the result of a heap allocation function, which invalidates the use of `malloc_usable_size`.

Figure 6.2: No-replace example from zlib: example.c line 69

```
void test_compress(compr, comprLen, uncompr, uncomprLen)
   Byte *compr, *uncompr;
   uLong comprLen, uncomprLen;
{
   int err;
   uLong len = (uLong)strlen(hello)+1;

   err = compress(compr, &comprLen, (const Bytef*)hello, len);
   CHECK_ERR(err, "compress");

   strcpy((char*)uncompr, "garbage");
   ...
}
```

Reason 3: Local buffer pointer is set equal to the result of a ternary expression which uses one of two different heap allocation functions.

Only one unsafe function use was not replaced for this reason. In this case, the buffer pointer being written to was set equal to the result of a ternary expression before its use in the unsafe function call. In both the ternary expression's 'then' and 'else' parts a heap allocation function was used. The pointer is set to the result of a heap allocation function in either case, so `malloc_usable_size` could have safely been used on it. It is unusual to use a ternary expression to allocate heap memory, however. This case could be handled simply by checking if both results of the ternary expression are heap allocation functions.

Figure 6.3 shows this case from gmp. The `sprintf` takes a local char pointer as its destination buffer. The buffer is assigned to the result of a macro a few lines before it is given to `sprintf`. When the macro is expanded it is a ternary expression that results in a char pointer from one of two gmp specific heap allocation functions, depending on the result of another function call.

Figure 6.3: No-replace example from gmp: test/trace.c line 257

```
void
mpn_tracea_file (const char *filename,
                 const mp_ptr *a, int count, mp_size_t size)
{
   char  *s;
   ...
   s = (char *) TMP_ALLOC (strlen (filename) + 50);

   for (i = 0; i < count; i++)
   {
      sprintf (s, "%s%d", filename, i);
      mpn_trace_file (s, a[i], size);
   }
   ...
}
```

Reason 4: Buffer pointer is given as an array access on a double pointer.

Three unsafe function uses were not replaced for this reason. In this case, the buffer being written to is given to the unsafe function as an array access dereference at some offset `i` on a double pointer. To follow what is done to the pointer at offset `i`, Safe Library Replacement transformation must also be able to follow what is done to the double pointer and the exact indices of any dereference on it. Following the double pointer could be done in the same way a single pointer is followed, but determining the exact index of any dereference is a challenge that would not resolve many cases.

Figure 6.4 shows this case from libpng. The pointer pointing to the destination buffer given to the `memcpy` comes from dereferencing a double char pointer at some offset using an array access expression. For example, if the double pointer is `pp`, then the destination buffer is given by `pp[someOffset]`. The exact value of `someOffset` is not known, so `malloc_usable_size` cannot be used safely.

Figure 6.4: No-replace example from libpng: pngwutil.c line 252

```
static int /* PRIVATE */
png_text_compress(png_structp png_ptr,
        png_charp text, png_size_t text_len, int compression,
        compression_state *comp)
{
   ...
   png_memcpy(comp->output_ptr[comp->num_output_ptr], png_ptr->zbuf,
              png_ptr->zbuf_size);
   ...
}
```

Reason 5: Buffer pointer is aliased.

Only one unsafe function use was not replaced for this reason. In this case, the buffer pointer being written to was part of a struct that the alias analysis determined had aliases. In cases where the buffer is aliased, it might be possible to follow all the aliases in the same way the buffer pointer is followed to investigate how the buffer is altered through the aliases.

Figure 6.5 shows this case from libpng. The destination buffer pointer is part of struct that comes in as a parameter to the `png_set_text_2` function. The pointer element is actually assigned to a heap allocation function before the `memcpy` such that `malloc_usable_size` could safely be used on it. However, later in the function an element of the struct that contains the buffer pointer is aliased. The OpenRefactory/C alias analysis considers the whole struct aliased and is also flow-insensitive. The imprecision of the alias analysis is what causes this case to not be replaced. OpenRefactory/C's alias analysis may be improved in the future.

Figure 6.5: No-replace example from libpng: pngset.c line 731

```
int /* PRIVATE */
png_set_text_2(png_structp png_ptr, png_infop info_ptr, png_textp text_ptr,
   int num_text)
{
   info_ptr->text = (png_textp)png_malloc_warn(png_ptr,
            (png_uint_32)(info_ptr->max_text * png_sizeof (png_text)));
   ...
   png_memcpy(info_ptr->text, old_text, (png_size_t)(old_max *
              png_sizeof(png_text)));
   ...
   png_textp textp = &(info_ptr->text[info_ptr->num_text]);
   ...
}
```

Reason 6: Buffer pointer is set equal to a heap allocation function, but it is part of a struct whose pointer is given to a function call.

Only one unsafe function use was not replaced for this reason. In this case, the buffer pointer being written to is part of a struct. The pointer element is set equal to the result of a heap allocation function, but before it is used in the unsafe function a pointer to the struct is given to a function call. The function may change the elements inside the struct. Therefore, the `malloc_usable_size` function cannot safely be used. It may be possible to look into the function that the struct is given to to see if the pointer element is changed.

Figure 6.6 shows this case from libpng. The destination buffer pointer is part of a struct that comes in as a parameter to the `png_push_save_buffer` function. The pointer element is assigned to a heap allocation function, but before it is given to the `memcpy`, its struct is given to a function. The function may change the pointer element, so it is unsafe to use `malloc_usable_size` on the pointer after that point.

Figure 6.6: No-replace example from libpng: pngpread.c line 632

```
void /* PRIVATE */
png_push_save_buffer(png_structp png_ptr)
{
   ...
   png_ptr->save_buffer = (png_bytep)png_malloc(png_ptr,
         (png_uint_32)new_max);
   ...
   png_free(png_ptr, old_buffer);
   ...
   png_memcpy(png_ptr->save_buffer + png_ptr->save_buffer_size,
         png_ptr->current_buffer_ptr, png_ptr->current_buffer_size);
   ...
}
```

Reason 7: Element access expressions did not match.

Only one unsafe function use was not replaced for this reason. In this case, the buffer pointer being written to is part of a struct, and it is set equal to the result of a heap allocation function such that `malloc_usable_size` can be used at the unsafe function call. However, when the pointer is accessed for use in the unsafe function call its struct is surrounded by parentheses. For example, `(myStruct).myPointer`. When it is set to the allocation function the struct does not have parentheses. The Safe Library Replacement transformation implementation uses string matching to follow element access expressions, so it does not recognize the allocation function assignment. This case could be fixed by improving the matching of element access expressions.

Figure 6.7 shows this case from gmp. The destination buffer is given as a pointer element of a struct with some pointer arithmetic done on it. Safe Library Replacement transformation can handle the pointer arithmetic, and the pointer element is set to the result of a heap allocation function such that `malloc_usable_size` could be used on it, but Safe Library Replacement transformation does not recognize the heap allocation assignment because the element access do not string match. When the macro `GMP_ASPRINTF_T_NEED` is expanded, it assigns `(d)->buff` to the result of a heap allocation function, but uses parentheses in the assignment.

Figure 6.7: No-replace example from gmp: printf/asprntffuns.c line 45

```
int
__gmp_asprintf_memory (struct gmp_asprintf_t *d, const char *str, size_t len)
{
  GMP_ASPRINTF_T_NEED (d, len);
  memcpy (d->buf + d->size, str, len);
  d->size += len;
  return len;
}
```

Chapter 7
Related Work

Most of the research on C buffer overflows concentrates on detection of buffer overflows. Safe Library Replacement transformation is different from buffer overflow detection in that it does not try to determine if buffer overflow will happen when an unsafe library function is used. Instead, Safe Library Replacement transformation simply removes the possibility of buffer overflows from the use of unsafe library functions by replacing them with safe alternatives. It is up to developers to remove buffer overflows when they are found by a buffer overflow detection tool. Buffer overflow detection tools fall into two main categories: static and dynamic. There are many static analysis tools [7, 22, 23, 38, 40, 42] and dynamic analysis tools [14, 15, 24, 30, 35] for C buffer overflow detection available.

Splint [7] is a lightweight, extensible static analysis tool for C that leverages annotated libraries and annotations added to source code by developers to find potential vulnerabilities. In the case of finding potential buffer overflows, a very simple approach is to issue a warning for every use of an unsafe library function. Unsafe functions are often used safely, however, so warning for all of them results in many false positives. To increase accuracy, annotations are added. For example, consider a use of `strcpy` with parameters `dst` and `src`. The safety requirement that `dst` be larger than `src` can be added by annotating the library's declaration of `strcpy` with a *requires* clause: `/*@requires maxSet(dst) >= maxRead(src) @*/`. `maxSet(b)` is the largest integer `i` such that `b[i]` can safely be an lvalue. `maxRead(b)` is the largest integer `i` such that `b[i]` can safely be used as an rvalue. If Splint cannot determine that the *requires* clause is true for a use of `strcpy`, it issues a warning.

Wagner et al. [40] describe an integer range analysis technique for detecting buffer overflows. They model buffers by two integer values: bytes allocated to the buffer (allocation size) and bytes currently in use (length). Detection of buffer overflows is done by checking, for each buffer, whether the inferred maximum length of the buffer is smaller than the inferred maximum allocation size. The implementation of the technique has two phases: generation of integer range constraints and solving the constraint system.

Archer [42] is a static analysis tool that uses path-sensitive, inter-procedural symbolic analysis to to detect memory access errors. It derives and propagates memory bounds and variable values by keeping track of constant relations, like `x = 4`, and symbolic constraint with unknown values, like `j < x < 16`. It uses a custom constraint solver to evaluate the values used in an operation given known constraints at every potentially dangerous memory access. Archer propagates constraints across procedure boundaries to make the analysis inter-procedural.

Purify [14] is a dynamic memory access error detection tool. It inserts function calls before every load and store in the object code produced by the compiler . The calls maintain a bit table that contains a two-bit state code for every byte of memory used by the program and checks every memory read and write for access errors. The three possible states for each byte are *unallocated*, *allocated but uninitialized*, and *allocated and initialized*. One example of when a memory access error occurs is when a byte that is unallocated is written to.

CRED (C Range Error Detector) [35] is a dynamic buffer overrun detector. It maintains a table of referent-objects that represent buffers and contain the base address and size of that buffer. Pointers in the program are associated with a referent-object if they point to somewhere inside the object. Any arithmetic done on an in-bounds pointer (a pointer associated with a referent-object) is checked with that pointer's referent-object to see if the resulting pointer will also be in-bounds. CRED is implemented as an extension of the GNU C compiler. The front end of the compiler is modified so that all object creation, address manipulation, and dereference operations are replaced with routines from their checking library.

STOBO (Systematic Testing Of Buffer Overflows) [15] is another dynamic buffer overflow detector. It detects buffer overflows by tracking the size of memory buffers and checking certain conditions before functions with buffer overflow vulnerability are called. The tool does this by implementing source code with additional functions. A function call is added for each buffer declaration and every heap memory management function. Also, a wrapper function is added in place of every function that may cause buffer overflow, like `strcpy`. The added functions keep track of the size of buffers and check for potential overflow from unsafe functions. A warning is issued if a buffer overflow is detected.

StackGuard [4] and PointGuard [5] are compiler extensions that dynamically prevent attacks by buffer overflows. StackGuard detects changes to the return address of a function before the function returns. It does this by placing a canary word next to the return address on the stack. If the canary word is changed, its assumes the return address has been changed too. PointGuard protects against buffer overflows by encrypting pointer values, except while they are in a register. Attackers can only access encrypted pointers because registers are not addressable via computed addresses. The encryption protects against attacks because attackers cannot corrupt a pointer such that it decrypts to a predictable value.

Hafiz [12] implemented a simple proof-of-concept Perl script to perform Safe Library Replacement transformation. The script uses simple string pattern matching to try to replace unsafe functions. The script has a list of patterns and a replacement procedure for each pattern. It looks for those patterns in uses of unsafe functions, and if it finds a pattern that it recognizes, it performs the corresponding replacement procedure. There are cases where a use of an unsafe function will not match any patterns. In such a case, the script cannot replace that function. The Safe Library Replacement transformation implemented in this work provides better replacement coverage and is less likely to break the original code than the Perl script because it uses more sophisticated static analyses and an AST code representation.

Dahn and Mancoridis [6] describe a different type of transformation to secure programs against buffer overflows. Their transformation changes a program so that all buffers are allocated on the heap. The motivation behind this is that performing a successful buffer overflow attack on a stack buffer is easier than a heap buffer. Overflowing a stack buffer can overwrite the return address and change the control flow of the program. Performing an attack of this severity is harder on heap buffers. The transformation is done using a language specifically designed for program transformations called TXL. TXL must be given a grammar for the input text so it can parse it into a parse tree similar to OpenRefactory/C's AST. It is also given a set of transformation rules that look for sub-trees in the parse tree that match some pattern, and replace that pattern with something else. The pattern the transformation looks for is a declaration of an arrays inside a function. These declarations are replaced by declarations of heap buffers when they are found.

Pointer analysis is a heavily researched topic. It is necessary in many program analyses. Precise pointer analysis is an NP-hard problem [20], so any practical analysis must trade precision for efficiency. The most precise algorithms are flow-sensitive and context-sensitive, which means they account for control flow dependencies and semantics of function calls. Much study has been devoted to flow-sensitive and/or context-sensitive algorithms [17, 31, 37, 41, 43], but none have been efficient enough to scale to large programs.

The most precise flow-insensitive and context-insensitive pointer analysis algorithm is currently Andersen's algorithm [1]. As a result, it was initially one of the least efficient. Steensgaard's flow-insensitive and context-insensitive algorithm [36] has near-linear time efficiency. Many improvements have been made to the efficiency of Andersen's algorithm since its publication, however, through the work of Hardekopf [13], Mendez-Lojo [26], and others [16, 9, 33, 3]. These improvements to its efficiency have given newer implementations of Andersen's algorithm one of the best balances of efficiency and precision.

Chapter 8
Conclusions and Future Work

I implemented the Safe Library Replacement transformation originally proposed by Hafiz [12], and evaluated its performance on a large test suite and three real C programs. The goal of the transformation is to remove buffer overflow vulnerabilities from the use of unsafe library functions without changing any other part of the program's behavior. It does this by replacing unsafe library functions with safe alternatives, and prevents unintended changes to behavior by not doing the transformation if it cannot guarantee the correctness of the new parameters to the safe function.

The evaluation of my Safe Library Replacement transformation implementation showed good results. It was able to safely replace the unsafe function and remove the buffer overflow in all 1,778 test cases from the Juliet Test Suite. It was able to safely replace 73% of the 181 unsafe functions found in the real C programs. The remainder of the unsafe functions were not replaced due to the Safe Library Replacement transformation implementation's decision that replacing it might break the test code. In no cases did the Safe Library Replacement transformation implementation break the test code by incorrectly replacing an unsafe function.

Safe Library Replacement transformation's ability to automatically replace, on average, 73% of unsafe functions in an entire C program has great value. The transformation can be run on legacy code that needs updating or greater security, and it can be run during the development process to ensure that no unsafe functions were accidentally used.

I also implemented an alias analysis as a part of the Safe Library Replacement transformation project. Alias analysis has uses in many other refactorings and transformations. It will continue to be used as an important static analysis in OpenRefactory/C.

The Safe Library Replacement transformation implementation shows promise for even better coverage in the future. One important future goal is to add support for more unsafe functions. Given that Safe Library Replacement transformation's algorithm for determining buffer size already works well, it would be easy to implement replacements for other functions that contain buffer overflow vulnerability. In addition, replacing functions that are unsafe for other reasons may be explored. Another important future goal for the Safe Library Replacement transformation is to enhance its capability to replace unsafe functions in a wider variety of uses. Chapter 6 lists all the reasons the transformation could not replace a function. Many of them have the potential to be resolved.

There are some cases, however, where it is impractical to expect the Safe Library Replacement transformation to work. In such cases, another Security-Oriented Program Transformation described by Hafiz [12], called Safe Type Replacement, may work better. Safe Type Replacement was briefly described in Section 1.3. Another of our future goals is to implement and evaluate Safe Type Replacement. Having both Safe Library Replacement and Safe Type Replacement as development tools would give developers significant capability to eliminate a large amount of buffer overflow vulnerabilities from their code.

Bibliography

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] James P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, Oct 1972.

[3] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[4] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX, editor, *Seventh USENIX Security Symposium proceedings: conference proceedings: San Antonio, Texas, January 26–29, 1998.* USENIX, 1998.

[5] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104. USENIX, August 2003.

[6] Christopher Dahn and Spiros Mancoridis. Using program transformation to secure c programs against buffer overflows. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 323–, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

[9] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

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

[11] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *European Conference on Object-Oriented Programming (ECOOP 2013), pages TO APPEAR, Montpellier, France*, 2013.

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

[13] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136, 1992.

[15] Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. In *NDSS*. The Internet Society, 2003.

[16] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[17] Michael Hind, Michael Burke, Paul Carini, and Jong deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21, 1999.

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

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

[20] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[21] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[22] Wei Le and Mary Lou Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 272–282, New York, NY, USA, 2008. ACM.

[23] Lian Li, Cristina Cifuentes, and Nathan Keynes. Practical and effective symbolic analysis for buffer overflow detection. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 317–326, New York, NY, USA, 2010. ACM.

[24] Davide Libenzi. Guarded memory move (GMM), February 10 2004.

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

[26] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443, 2010.

[27] Matt Messier and John Viega. Safe C string library v1.0.3. `http://www.zork.org/safestr/safestr.html`.

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

[29] Todd Miller and Theo de Raadt. `strlcpy` and `strlcat` — Consistent, safe, string copy and concatenation. In *1999 Usenix Annual Technical Conference, Monterey, California, USA*, 1999.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44:245–258, June 2009.

[31] ErikM. Nystrom, Hong-Seok Kim, and Wen-meiW. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 165–180. Springer Berlin Heidelberg, 2004.

[32] National Institute of Standards and Technology (NIST). Juliet Test Suite for C/C++ version 1.2. `http://samate.nist.gov/SRD/testsuite.php`, May 2013.

[33] David J. Pearce. Efficient field-sensitive pointer analysis for c. In *In ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, page 2008, 2004.

[34] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[35] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[36] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[37] TeckBok Tok, SamuelZ. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2006.

[38] John Viega, J. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference*. ACM, 2000.

[39] Visual C++ Library. Runtime library reference: getenv_s.

[40] David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*. The Internet Society, 2000.

[41] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference*

on Programming language design and implementation, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[42] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28:327–336, September 2003.

[43] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 145–157, New York, NY, USA, 2004. ACM.