ABSTRACT

SRINIVASA, GOPAL RANGANATHA. Abstraction-Based Static Analysis of Buffer Overruns in C Programs. (Under the direction of Associate Professor Dr. S. Purushothaman Iyer).

Bounds violations or *buffer overruns* have historically been a major source of defects in software systems, making bounds checking a key component in practical automatic verification methods. With the advent of the Internet, buffer overruns have been exploited by attackers to break into secure systems as well. Many security violations ranging from the 1988 Internet worm incident to the AnalogX Proxy server vulnerability have been attributed to buffer overruns. Programs written in the C language, which comprise most of the systems software available today, are particularly vulnerable because of the lack of array bounds checking in the C compiler, presence of pointers that can be used to write anywhere in memory, and the weak type system of the C language.

Many methods have been proposed to detect these errors. Runtime methods that detect buffer overruns suffer from significant overhead and incomplete coverage, while compile time methods could suffer from low accuracy and poor scalability.

In this thesis, we propose a new technique for bounds checking based on data abstraction that is more accurate, more scalable, and suffers from no runtime overhead. Enhancements have been made to C Wolf, a suite of model generation tools, to handle buffer overflow analysis.

Case studies on web2c, a publicly available software package, pico server, an open source web server, and on the wu-ftpd server are presented to demonstrate the practicality of the technique.

Abstraction-Based Static Analysis of Buffer Overruns in C Programs

by

Gopal Ranganatha Srinivasa

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Department of Computer Science

Raleigh

2003

Approved By:

Dr. Matthias F M Stallmann

Dr. Daniel C DuVarney

Dr. S. Purushothaman Iyer
Chair of Advisory Committee

Dr. Peng Ning

To my family

Biography

Gopal Srinivasa was born on 18th June, 1977 in Bangalore, India. He did his preliminary schooling at Daffodils English School and attended High School at the St. Josephs' Indian High School. He completed his undergraduate degree in Computer Science from the University Visvesvaraya College of Engineering, affiliated to Bangalore University, Bangalore in 1998 and worked for Siemens Communication Software, the 4th largest R&D unit of Siemens telecom, for three years. Gopal joined the Masters' program in Computer Science at NC State in August 2001. His research interests include Formal methods, Software Engineering and Object-Oriented Analysis and Design. His non-research interests are Cricket, Indian classical music, and Reading.

Acknowledgements

This thesis wouldn't have been possible without the support and guidance of my advisor, Dr. Purushothaman Iyer, who with a great deal of patience and kindness helped me through out the duration of this work. Many thanks are due to Dr. Dan DuVarney, for his help with ML, the C Wolf system, and almost everything else I asked him for. I am also grateful to Dr. Peng Ning and Dr. Matt Stallmann for their advice on some key aspects of this thesis.

I take this opportunity to thank the faculty and staff at NCSU, particularly in the Computer Science department, for their help, advice and support.

Finally, I would like to thank my family and friends, whose support and encouragement was my lifeline throughout the last two years.

Contents

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Buffer Overrun Vulnerabilities in C 4
- 1.2 Research Goals and Contributions 7
- 1.3 Conventions 7
- 1.4 Thesis Outline 7

2 Related Work 9
- 2.1 Runtime Bounds Analysis 9
- 2.2 Static Bounds Analysis 10
  - 2.2.1 Syntax-Based Analysis Techniques 11
  - 2.2.2 Constraint Solving Techniques 12
  - 2.2.3 Annotation-based Techniques 14
- 2.3 The Case for Abstract Interpretation 15

3 Principle 17
- 3.1 The Abstraction Map 18
- 3.2 Model Generation 20
- 3.3 Buffer overflow detection 21
- 3.4 Dynamically allocated arrays 24
- 3.5 Partial order analysis 26

4 Theory 35
- 4.1 The char and char array abstractions 35
- 4.2 The Dynamic Range Abstraction 36
- 4.3 Dynamic Range Conversions 37
- 4.4 Dynamic Range Operators 38
- 4.5 Array abstractions 38

5 Implementation **39**
5.1 Partial Order Table . . . 39
5.2 Dynamic Range Abstraction . . . 40
5.3 Design of the char array abstraction . . . 40
5.4 Structures . . . 43

6 Results **45**
6.1 Analysis of Web2c . . . 46
6.2 Analysis of the Pico Server . . . 48
6.3 Analysis of the FTP daemon . . . 49

7 Conclusions and Future Work **51**
7.1 Enhancements . . . 52
7.2 Summary . . . 53

Bibliography **54**

List of Figures

1.1 Possible stack layout after a function call 2
1.2 Overwriting the return address with an attacker's return address 3
1.3 Subverting a kernel process to get root access 3
1.4 Possible buffer overflow in strcpy as implemented in C 5
1.5 A security violation caused by a buffer overflow in a string operation . . . 6

2.1 Program for which metacompilation reports false positives 12
2.2 Code fragment for which metacompilation reports false negatives 13
2.3 Undetected buffer overflow while using BOON 13

3.1 Schematic diagram of the C Wolf system 17
3.2 Determining array sizes in C Wolf 19
3.3 For loop that results in redundant bounds checks 22
3.4 Model generated for the *for* loop with `int` abstraction 23
3.5 Model generated for the *for* loop with the dynamic range abstraction 25
3.6 Program fragment with a *complex* expression 29
3.7 Partial order table for program in Figure 3.6 29
3.8 Partial order table of Figure 3.7 after transitive closure 30
3.9 Partial order table of Figure 3.8 after the temporaries are removed 30
3.10 Function call that updates the partial order table 31
3.11 Partial order table for the code in Figure 3.10 before the function call . . . 32
3.12 Partial order table for the code in Figure 3.10 at the start of the function . 32
3.13 Partial order table for the code in Figure 3.10 before the function return . 33

4.1 Application of the abstraction and concretization functions 37

6.1 Possible buffer overflow caused by a string operation 47

List of Tables

3.1 C Wolf Abstractions 18
3.2 Relational operators and dynamic range variables 24
3.3 Rules for partial order arithmetic 27

4.1 Mathematical notation used in this thesis 35
4.2 Dynamic range operations 38

6.1 Results of some C Wolf runs 45
6.2 Comparison between cleanness checking and data abstraction 48
6.3 Buffer overflow errors detected in the Pico Server 49

Chapter 1

Introduction

Buffer overruns, also known as *bounds violations*, have been a major source of software errors and security holes for many years, making bounds checking an important aspect of practical software verification techniques. Buffer overruns are caused by unintentional accesses to memory outside of the storage allocated to hold the buffer contents. Typically, these errors are caused by programmer mistakes and insufficient language support to detect such mistakes. While earlier these errors were an annoyance, the advent of the Internet has metamorphosed buffer overruns from a nuisance into a major security threat. The Internet worm that infected thousands of computers in November 1998 used a buffer overrun vulnerability in the *fingerd* daemon to replicate itself [39]. Data extracted from CERT advisories show that buffer overruns account for 27% of the entries in one vulnerability database [37] and for 23% in another [22]. Even the *password disclosure* error in SQLBase 8.1.0 has been attributed to a buffer overflow [34]. A random sample of the vulnerabilities posted in the Security focus website [34] revealed that in four days, of the twenty security violations reported for UNIX systems, six were caused by buffer overruns. Many security holes in the popular wu-ftpd server, including some that even allowed attackers to start a shell on the server, have been traced to buffer overruns in the FTP daemon [34]. Even though the vulnerabilities were found on all major platforms and operating systems, they are commonly caused by the reliance on non-typesafe languages, such as C and C++.

It is important to understand how buffer overruns can compromise security. Typically, most systems store the return address of the calling function on the program stack

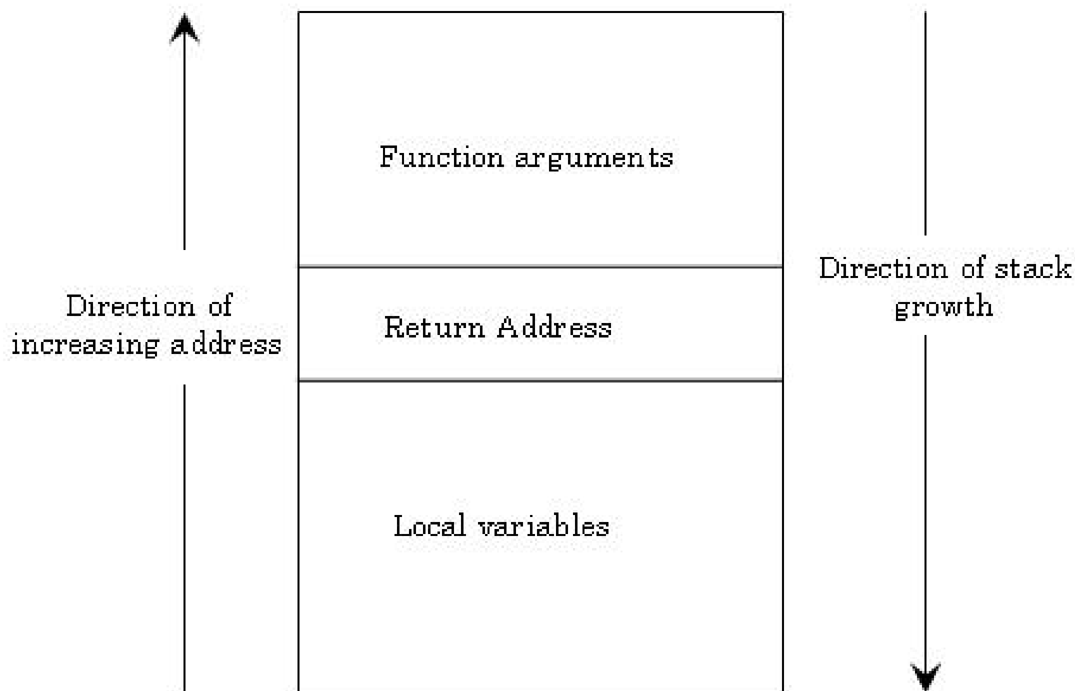

Figure 1.1: Possible stack layout after a function call

along with the local variables of the callee and the arguments passed to it (Figure 1.1).

In order to gain control over the system, an attacker can use buffers to traverse down the stack and overwrite the return address. This is accomplished by supplying a buffer with more data than the buffer is allocated to hold. The extra data overwrites memory beyond the end of the buffer, obliterating the contents of the other local variables and then the saved return address. This is called *stack smashing* [29], and it allows an attacker to replace the return address on the stack with the address of a malicious function that can compromise system security. Thus, when the callee finishes execution, it returns control to the malicious function, allowing the attacker to take control of the system. For example, the code shown in Figure 1.2 is susceptible to stack smashing, particularly since the data requested from the user is not sanity-checked before using it in the program. An attacker could gain control over the system by passing a string large enough to overflow the buffer `buff` and overwrite the saved return address.

The code in Figure 1.2 also illustrates the primary reason behind buffer overflows

```
void main ( )
{
    char buff[MAXLEN];

    char* ptr = get_data_from_network();
    copy_into(buff, ptr);
}
```

Figure 1.2: Overwriting the return address with an attacker's return address

- a combination of programmer error and lack of runtime bounds checking. In the example, while the programmer has failed to check the data supplied by the user, the language (in this case C), exacerbates the problem by not detecting the buffer overrun when it occurs.

Buffer overruns can be also be used to gain root access in operating systems like UNIX and Windows NT as they support root-privilege processes. A normal user can subvert these processes by causing a buffer overflow and gain root access to the system. Figure 1.3, taken from [2], shows a function in one such process. The function reads an integer from user space through the `copy_from_user` function and uses the integer as an index into an array of frames, without sanitizing it first. A malicious user can pass an illegal index to the function causing a buffer overflow. The process then aborts, exiting into the root shell, giving root access to the user.

```
if (!copy_from_user(&frame, arg, sizeof(int)))
    return -EFAULT;
ret = se401_newframe(se401, frame);
se401->frame[frame].grabstate=FRAME_UNUSED; /* unsanitized
                                                   array index */
return ret;
```

Figure 1.3: Subverting a kernel process to get root access

Buffer overruns cause other problems as well. Small operating systems like MS-DOS [13] and real-time operating systems like some versions of VxWorks afford little or no memory protection, allowing users to write into all available memory. In such systems, buffer overruns may result in overwriting operating system data structures, leading to un-

predictable behavior. In networked systems, buffer overruns can be used by malicious users to write to any file or to run any command on the compromised system.

1.1 Buffer Overrun Vulnerabilities in C

The C language [6, 20] is a popular language used for a variety of programming tasks - from the development of the UNIX operating system to the development of real-time software that controls flight-guidance systems. C is popular because of its efficiency, portability, closeness to the hardware and the presence of many features that give programmers more control over the system than most other languages. The efficiency and control however, come at the expense of type-safety and memory-safety, which is not ensured. Furthermore, the control provided is much more than most programmers can handle, as evidenced by all the memory-related programming errors, including buffer overflows, memory leaks and dangling pointers. To provide for efficient runtime behavior and a small memory footprint C provides minimal runtime support, which excludes checking for memory-related errors. This makes memory errors go undetected by the compiler and the runtime environment, allowing attackers to write into the program stack ,heap or static storage areas. For example, in a language like Pascal the code of Figure 1.2 would cause a runtime exception when a string longer than `MAXLEN` is encountered, whereas the runtime environment of C would let it through, allowing the attacker to take control of the system.

The treatment of strings in C allows for many more exploits. C does not provide users with a standard, bounded string data type. Programmers have to create strings by treating them as arrays of characters. These arrays, like arrays of other types in C, are not bounds checked making the program unsafe. Further problems are caused by the strings being bound not by the length of their buffers, but by a special *null* character. As a result, string manipulation functions in the standard C library manipulate strings by checking for this character, presenting many opportunities for buffer overruns when the arguments to these functions do not have the *null* character. For example, the C string library function `strcpy` copies the characters of a source string into a destination string. However, not only does it ignore to check that the destination string is large enough to hold the source string, it also assumes that the source string is terminated by the *null* character. Figure 1.4 represents the semantics of the `strcpy` function, the code borrowed from [6]. This code

```
char* strcpy (char*d, char*s)
{
  while ( (*d++)=(*s++) ) ;
  return d;
}
```

Figure 1.4: Possible buffer overflow in strcpy as implemented in C

poses the threat of multiple buffer overruns that may be caused either by the destination string being allocated lesser memory than the source or by the source string lacking the terminating *null* character. In fact, in the second case, it would be possible to completely overwrite the contents of the entire available memory if the operating system does not employ protection mechanisms.

Strings are therefore a primary culprit responsible for many of the buffer overruns in C programs. A study done on UNIX systems [26] traced nearly 30% of buffer overruns to errors in string manipulation functions. Even though strings and string manipulation functions are inherently unsafe, every practical C program has to use strings as strings are required to store user data, read from pipes, ports, and files, and to interact with the operating system. Even though some string functions like `strncat` and `strncpy` are safe, the return values of these functions can be non-null terminated, leading to overruns in subsequent sections of the program.

Another example of a security violation caused by a buffer overflow in a string manipulation function, taken from the RSA website [31], is shown in Figure 1.5. If the password passed to the program is larger than `MAXPWDLEN` characters, the `strcpy` function could overwrite the `passwd_ok` variable with one of the characters in the entered password, thus allowing the user access to the system even when the password is incorrect.

The behavior of many functions in the standard library also gives rise to unsafe conditions. Library functions like `strcpy` automatically append a *null* character to their return values, while others, like `strncat` do not. Functions like `gets` do not perform bounds checks on the buffers passed to them. Even innocuous functions like `printf` have been used by attackers to break into secure systems [38, 11]. The implementation of the GNU `malloc` function, which uses a doubly-linked list to hold the *freed* blocks, opens up

```
int main(int argc, char *argv[])
{
    char passwd_ok = 0;
    char passwd[MAXPWDLEN];

    strcpy(passwd, argv[1]);

    if ( check_passwd(passwd) == PASSWD_OK )
        passwd_ok = 1;

    if (passwd_ok) {
      ...
    }
}
```

Figure 1.5: A security violation caused by a buffer overflow in a string operation

new vulnerabilities. In order to conserve memory, the implementation allows two of these pointers to overlap user data. The pointers are erased if a block is freed twice, and when the doubly linked list is updated during the second free, an arbitrary memory location gets overwritten. This is called the *double free* attack [1]. A similar attack, called the *heap overflow* attack, can be used to change the pointers in the allocation list used by `malloc` [23]. The strategy used here is to overflow the allocated buffer so that the extra data overwrites the pointers of the next entry in the list. Another class of attacks, called the *integer overflow* attacks, utilize the weak-type system of the C language to cause buffer overflow. Such attacks are effective in programs that allow users to decide the size of the buffers. Even if the program sanitizes the values before allocating the buffers, it is possible to bypass these checks by choosing the right values. For example, if a program typecasts a size to a short before passing it to malloc, choosing 65536 as the size makes the system cast it to zero, resulting in the program thinking that the buffer is 64Kbytes in size while in reality, it is much smaller [4].

In summary, we note that programs written in C are more prone to buffer overflow vulnerabilities. These include vulnerabilities in stack and heap-based buffers, string manipulation functions and vulnerabilities caused by the weak type system of the language. We therefore observe that at the very least, any method that attempts to detect bounds viola-

tions in C programs must offer the ability to check statically allocated buffers, dynamically allocated buffers, string manipulation functions, and pointer arithmetic.

1.2 Research Goals and Contributions

The primary goal of this thesis is to develop scalable tools and techniques for detecting buffer overruns at compile time. We have enhanced an abstraction-based model generation toolset called C Wolf for this purpose and achieved the following:

- Developed a technique for detecting buffer overruns using data abstraction
- Enhanced the C Wolf toolset to detect buffer overruns using the above technique
- Added abstractions to minimize the increase in the number of states due to bounds analysis
- Performed experiments to validate the approach, and
- Compared our technique with the work of others and identified the trade-offs involved in each approach.

1.3 Conventions

Three basic fonts are used in this thesis. The first is the regular font, in which most of the text including this one is typeset. Computer programs, C expressions, and statements appear in the `teletype` font. Finally, *italics* are used to emphasize words, and are used when a term is introduced for the first time in the document.

Comments that explain important aspects of a figure are enclosed in `(* *)`.

Mathematical notations used in the document will be introduced in the relevant sections.

1.4 Thesis Outline

Chapter 2 outlines some of the related work and the ideas that led to our work,

Chapter 3 outlines some of the guiding principles of the thesis,

Chapter 4 explains the theory behind the abstractions used,

Chapter 5 describes the implementation of the ideas proposed,

Chapter 6 details some of the results, and

Chapter 7 presents the conclusions and suggestions for future work.

Chapter 2

Related Work

Bounds checking or Buffer overflow analysis has been widely researched because of its implications for enhancing program reliability and security. Most of the work done in this field can be categorized into:

- *Runtime analysis*
- *Static (compile-time) analysis*
- *Runtime analysis with compile-time support*

2.1 Runtime Bounds Analysis

Language support for runtime bounds checking has been prevalent since the days of Pascal and Ada, and is offered even by present-day languages like Java and Visual Basic. Since the C language lacks runtime support for bounds checking, bounds analysis for C programs is performed using external tools like the Compuware BoundsChecker [8], Rational Purify [30] and StackGuard [36]. BoundsChecker and Purify instrument the program data structures with special values and annotate the source code with assertion statements that check for bounds errors. StackGuard, on the other hand, instruments the program stack with a special random value called the canary, and can detect occurrences of stack smash attacks. Typically, programs that are instrumented with Purify and BoundsChecker are

tested with the instrumentations and released without the instrumented code, while the instrumentation code introduced by StackGuard is retained even in the release versions.

Runtime tools have the advantage of being extremely accurate, detecting bounds violations exactly when they occur. They are programmer-friendly, and require little or no effort on the part of the developers who use them. Usually, the tools are activated by means of a compiler switch that lets them instrument the program and include the relevant libraries in the executable.

The tools however, are not without their drawbacks. The primary problem with tools like Purify and BoundsChecker is the lack of complete code coverage. While Purify and BoundsChecker do detect all the errors on the paths that have been covered by testing, errors on paths that are not covered go undetected. Furthermore, since Purify and BoundsChecker are essentially testing tools, their performance is dependent on the data being used to test the software. Thus, there is no guarantee of complete code coverage, nor is there a guarantee that the program has been tested with all possible inputs. In essence, these tools suffer from all the drawbacks of conventional software testing techniques. Undetected buffer overflows therefore remain in the program to be exploited by an attacker. The problems mentioned above can be avoided by releasing the instrumented versions of the source, but that comes with a very large penalty in terms of runtime overhead making the technique unsuitable for most systems.

Another problem with runtime checking is the little scope available for recovery. StackGuard simply terminates the program when it detects a stack smash attack. Programs instrumented with Purify or BoundsChecker, if released with the instrumentations, terminate when the tool-inserted assertion statements fail. Clearly, this is not an option for operating system modules, servers, and software designed to run round-the-clock. In such systems, bounds analysis must be done earlier, preferably at compile time itself.

2.2 Static Bounds Analysis

Static analysis techniques, as distinct from runtime ones, attempt to find bounds violations at compile-time. They have the advantage of being able to scan the entire source for bounds errors, providing *complete coverage*, something that is not possible with runtime methods. Furthermore, since static analysis techniques detect buffer overflow errors at

compile time, they offer programmers the opportunity to insert checks that can handle overflow errors when they occur, thus avoiding the ungraceful termination problem endemic to runtime techniques.

Static analysis techniques that detect buffer overflows can be classified into:

- **Syntax-based analysis techniques.** These techniques typically operate by performing a syntax-based analysis of the program to detect buffer overruns. Syntax-based methods can be as simple as methods that simply report calls made to unsafe functions, or as complex as *metacompilation*, a technique that checks among other things, whether array subscript variables are sanitized before use [2].

- **Constraint solving techniques.** These techniques convert the bounds checking problem into a set of integer constraint problems for analysis. This category includes among others, the work on *Cleanness Checking* by Nurit Dor *et.al.* [12], David Wagner's work on developing BOON [39], and the work done by Radu Rugina and Martin Renard [32].

- **Annotation-based techniques.** These techniques use annotations supplied by the programmer to verify bounds safety. These include the *Splint* tool developed by David Evans *et.al.* [17] and *CQual*, developed by Foster *et.al.* [38]. Both use some form of programmer-annotations to check safety properties of C programs.

2.2.1 Syntax-Based Analysis Techniques

Syntax-based analysis attempt to detect bounds errors by taking advantage of some syntactic features of program semantics. For example, *metacompilation* performs bounds checking by ensuring that variables are checked against a lower and an upper bound before being used as an array subscript. Initially metacompilation places all the variables in the *tainted* state. If the compiler encounters a conditional where a variable is checked for an upper bound (i.e., checked if it is less than some value), the variable then moves to the *needs_lb* state. Similarly, a variable moves from the *tainted* state to the *needs_ub* state if it is checked against a lower bound. A variable that has been checked against both a lower and an upper bound is said to be in the *untainted* state. Array accesses that use *untainted* variables as the subscript are treated as safe accesses. All other array accesses are treated as being unsafe and are reported to the user.

```
#define N 3
int arr[N];
int i = 0 ;

for ( i = 0; i < N; i++ )
{
      arr[i] = 0;
}
```

Figure 2.1: Program for which metacompilation reports false positives

The process is repeated every time the subscript is assigned a value, as a new assignment rolls back the variable from any of the *untainted*, *needs_lb* or *needs_ub* states to the *tainted* state.

This technique, proposed by Dawson Engler, is simple, fast, and does not have any runtime overhead. A tool called *xgcc* developed using this principle, was successfully used to find many security problems in Linux and OpenBSD sources. However, the method has a potential problem in that there is no guarantee that the subscript has been checked against the right bounds. The method also does not take into account aliasing, nor does it handle difficult-to-find string manipulation errors. Further, many false alarms are also possible as some subscript variables might be within bounds, even if their bounds are not explicitly checked using conditionals. An example is given in Figure 2.1. Here, *xgcc* would report a bounds violation, stating that the lower bound is not checked, even though the lower bound for the array subscript is well within the array bounds. Also, a program like the one in Figure 2.2 would be reported as being safe, because the method is incapable of detecting that the subscript is being compared with the wrong bounds. Thus, metacompilation is not sound, though it scales well to large programs.

2.2.2 Constraint Solving Techniques

Constraint solving techniques typically convert the problem of analyzing buffer overruns into integer constraint problems that can be solved to verify bounds safety of programs.

The approach introduced by David Wagner *et al.* [39] mainly attacks the problem of buffer overruns in C strings. In this approach, strings are treated as instances of an

```
#define N 3
int arr[N];
int i;

if ( i <= N && i >= 0 )
      arr[i] = 0;
```

Figure 2.2: Code fragment for which metacompilation reports false negatives

abstract data type that has standard C functions like `strcpy` and `strstr` as its operations. Furthermore, the technique uses the fact that for the purpose of buffer overflow detection, only the length and the size of a string need to be known. Strings are therefore modeled as integer pairs: one element of the pair representing the current *length* of the buffer and the other representing the number of bytes allocated to it(*its size*). A system of integer equations is generated from the code, and the system solved to determine bounds violations. Wagner *et al.* also developed a tool called BOON [5] to check C programs for buffer overflow vulnerabilities. The tool found many buffer overflow errors in the UNIX `sendmail` program and the `nettools` package.

While this method is more accurate than metacompilation, it still suffers from many false positives and is also incapable of dealing with constructs like pointer aliasing. For example, the buffer overflow that occurs when a 13-byte string is copied into the 10-byte buffer t in the code of Figure 2.3 goes undetected by BOON. This happens because the analyzer is unable to detect that the pointer `p` aliases a location in the array `s`.

```
char s[20], *p, t[10];
strcpy (s, "Hello");
p = s + 5;
strcpy (p, "world!");
strcpy (t,s);
```

Figure 2.3: Undetected buffer overflow while using BOON

Cleanness Checking [12], is a constraint solving approach that improves on BOON by providing flow-sensitivity, pointer analysis and linear invariants to conservatively detect all buffer overruns in a program. Like Wagner's method, this method proposed by Nurit Dor

et al. [12], works by treating strings as an abstract data type. However, the method is more accurate and reports fewer false alarms. By performing pointer analysis, the technique is able to detect bounds errors in programs like the one in Figure 2.3. The analysis procedure can also be used to instrument the C program with `assert` statements that detect bounds violations at runtime, though the ability comes at the cost of a larger memory footprint and increased execution time. Cleanness Checking has been tested on sources that were part of the `web2c` software package with good results.

A serious drawback of this method, however, is the lack of inter-procedural analysis. Large programs are analyzed using user-provided pre- and post- conditions for every procedure. Errors in the pre- and post-conditions may severely affect the static analysis procedure. Furthermore, the technique is incapable of analyzing structures and is slow and unscalable, with the static analysis of a 403-line program taking nearly 310 seconds and 99.5MB of RAM [12]. Results reported in [12] also suggest that the runtime overhead remains significantly high for many applications.

2.2.3 Annotation-based Techniques

Annotation-based techniques rely on programmer-provided annotations to verify bounds safety of programs. Annotations are similar to pre- and post-conditions, describing the intentions and assumptions of a programmer [17, 25]. Annotations can be applied to variables, function parameters and return values, indicating their properties.

David Evans *et al.* have implemented a tool called Splint [35] for detecting buffer overruns using such annotations. The tool provides standard annotations to describe the properties of variables, parameters to functions and return values from functions. For example, the annotation `@notnull@` when applied to a variable v indicates that v cannot be null. Similarly, the `@null@` annotation applied to a variable v indicates that v can be null. The annotations `@requires` and `@ensures` reflect the assumptions and intents of a function, while the annotations `@maxSet` and `@minSet` bind the range of values that can be used to index an array. Standard C library functions are provided with the correct annotations by the developers of Splint, while programmers are expected to provide annotations for their functions.

The tool then uses a static analysis algorithm to find violations of the annotations. For example, the `strcpy` function is annotated with `@notnull@s1, @notnull@s2,`

`/*@requires maxSet(s1) >= maxRead(s2)@*/`
`/*@ensures maxRead(s1) == maxRead (s2) /\ result == s1@*/` where `maxRead` represents the length of the string. If any of the arguments passed to the function violate the `@requires` constraint, an error is reported. Similarly, if `s1` or `s2` are null, the tool reports a constraint violation.

While the tool found considerable success with wu-ftpd and BIND [17], the technique suffers from the inability to incorporate control-flow information into the bounds analysis procedure, and uses standard programming idioms and heuristics to analyze the same. Furthermore, annotation-based techniques are cumbersome and error-prone, as wrong annotations can lead to misleading analysis results.

2.3 The Case for Abstract Interpretation

The preceding discussions indicate that a purely syntactic approach though scalable, is not effective in solving the problem of identifying potential buffer overflows. Constraint solving has the drawback that it is not scalable. Annotation-based methods are vulnerable to user errors and also lack sufficient information to handle many program constructs. Furthermore, the above methods lack sufficient information at compile-time to correctly identify many kinds of buffer overflows. Clearly, the runtime value of data needs to be taken into account while performing static analysis. It is here that *abstract interpretation* can play a major role. Abstract Interpretation is the process of executing the statements of a program in an *abstract domain*. A mapping of values from the *concrete domain* to the *abstract domain* is provided, as is a mapping of the operations. These mappings define the semantics of the program statements in the context of the abstract domain. Abstract Interpretation is used by model generation tools to generate models of programs. A model of the program thus obtained, can be checked for temporal and safety properties. Practical systems [3, 9, 15, 21] that use abstract interpretation to extract models from programs have been built. Since the process of abstract interpretation is sound, properties that hold in the abstract domain hold in the concrete domain as well(provided the abstractions are correct). Bounds safety is one such property. It is therefore possible to identify potential buffer overflows by reasoning with the model; an approach we take in this thesis.

Model generation using abstract interpretation is usually a difficult problem be-

cause of size of the models involved. Many techniques have been used to reduce model size, which include predicate abstraction [3, 21, 33] and data abstraction [10, 7, 14].

Predicate abstraction, first proposed by Graf and Saidi [33], provides a means of combining theorem proving and model checking techniques by automatically mapping an unbounded system (called the *concrete system*) onto a finite state system (called the *abstract system*), which can then be verified using model-checking tools. Predicate abstraction works by representing the values of a systems' variables as assignments to a set of boolean predicates. The resulting program can be model-checked using specializedtools that are based on predicate abstraction [3].

Data abstraction [7], on the other hand, works by mapping the values of the systems' variables onto a much smaller set of values based on how each variable is used in the system. A good example is the return value from the standard C string library function `strcmp`. The `strcmp` function takes two strings as arguments and returns the difference of the first differing characters in them [20]. However, in most situations, programmers are only interested in knowing if the return value is negative `(str1 < str2)`, positive `(str1 > str2)` or zero `(str1 == str2)`. The return value can therefore be abstracted to just three values, down from a possible 511 (assuming that the strings are in ASCII representation), effectively reducing the size of the state machine by 508 states. Techniques that use data abstraction to extract models from source code usually employ an abstract virtual machine (AVM) to interpret the given program in the abstract domain. The process requires that the user specify an *abstraction map* that maps values from the concrete domain to the abstract domain [14, 9].

Data abstraction has been used to generate models for programs written in the C and Java programming languages. The C Wolf system developed at NC State University generates models from C programs using this principle. The system has been used to extract models from many large systems, including the GNU i-protocol and BSD-FTP [15]. This thesis looks at applying the same techniques to detect buffer overruns. We enhance the C Wolf toolset with support for bounds checking. We also investigate methods for making the bounds checks more efficient, so as to cause minimal delay in the model generation process and minimal increase in model size, in order to ensure the scalability of the technique.

Chapter 3

Principle

C Wolf is a suite of tools that takes a C program as input and generates a model for the program. The model is a labeled transition system and can be in one of text, graphical, or CCS formats. Figure 3.1, reproduced from [14], shows a schematic diagram of C Wolf.

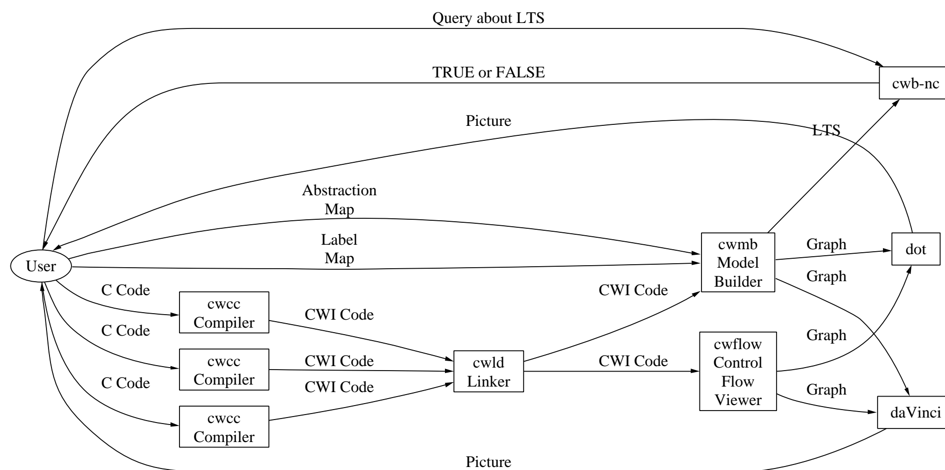

Figure 3.1: Schematic diagram of the C Wolf system

The C compiler, `cwcc`, takes a C file as input and writes out a binary *cwi* file that contains type information, variable and function tables, and intermediate code organized as basic blocks. The *cwi* linker, `cwld`, links various *cwi* files to generate a single, combined

cwi file for the entire program.

The model builder, `cwmb`, then reads the *.cwi* file and the user- specified *abstraction map* and *label-map* files to generate the model.The abstraction map contains the abstractions that are to be applied to the variables in the program; the label map specifies rules that are to be applied for labeling the transitions in the generated FSM.

3.1 The Abstraction Map

C Wolf allows its users to abstract integers, integer arrays and character arrays (strings). Abstractions are specified in an input file to the model builder, called the abstraction map or the *.am* file. Currently, C Wolf supports the abstractions shown in Table 3.1, replicated here from [14].

Syntax	**Description**
top	All values are abstracted to a single value $\top$
minmax	Values are abstracted to an upper and lower bound $\langle l, h \rangle$
mod(k)	Values are abstracted to the set of remainders modulo k
part$\langle a_1, a_2, ..., a_k \rangle$	Values are abstracted to a partition of the set of integers. The partition is $[-\infty \ldots a_1 - 1], [a_1 \ldots a_2 - 1], \ldots, [a_{k-1} \ldots a_k - 1], [a_k \ldots \infty]$
int	The value is either a precise integer or $\top$
τ array	The values is an array of values of abstracted type τ, where $\tau \in \{$int, bool, pointer, char, part$\langle a_1, a_2, \ldots, a_k \rangle\}$
free	The abstraction is chosen dynamically, and the variable assumes the abstraction of the latest assigned value
bool	The values are true, false and maybe

Table 3.1: C Wolf Abstractions

The default abstraction is `top`. Applying top to a variable simply abstracts away the variable from the program. This abstraction is used to abstract away those variables from the program whose values do not affect the properties of the model we wish to extract. The `mod` (k) abstraction maintains a set of remainders under division modulo k. For example, if a variable is abstracted using the abstraction `mod` (4) and the abstract value $\{1, 2\}$, the set of concrete values that the variable can take are $\{1 + 4 \cdot i \mid i \geq 0\} \bigcup \{2 + 4 \cdot i \mid i \geq 0\}$. The `minmax` abstraction maintains an upper and a lower bound on the concrete values that

a variable may take. The `int` abstraction tells the system to compute a precise value, or if it is not possible, set the variable to $\top$. Variables that use the `free` abstraction continuously take on the abstraction of the value that is being assigned to them. The `part` abstraction partitions the set of integers into a set of disjoint intervals. This abstraction is the workhorse abstraction, and can be used to track both individual values and ranges of values. For example, to track the values 4,7 and 8, we can specify a partition of the form `part (4,5,7,8,9)`. The concrete values are partitioned into the sets denoted by $[-\infty, 3], [4, 4], [5, 6], [7, 7], [8, 8], [9, \infty]$.

The τ `array` abstractions tell the system to treat the variable as a single-dimensional array with the elements having the τ abstraction. In the C source file, the array being abstracted must either be statically allocated or should be allocated using a heap allocation function like `malloc` or `calloc`. It is important to note that the system must be able to evaluate the arguments passed to the heap allocation function while building the model. For example, in the program segment of Figure 3.2, the variables `arr` and `pArr1` are abstracted as arrays of 20 integers while the variable `pArr2` is abstracted as an array of 30 integers. However, `pArr3` is abstracted to $\top$ as the model-builder cannot determine its size during abstract interpretation.

```
int     arr[20];
int     *pArr1;
int     *pArr2;
int     *pArr3;

void main ( )
{
    int v = 30;
    int k;

    scanf("%d",&k);
    pArr1 = (int *) malloc (20 * sizeof(int));
    pArr2 = (int *) malloc (v  * sizeof(int));
    pArr3 = (int *) malloc (k  * sizeof(int));
}
```

Figure 3.2: Determining array sizes in C Wolf

Pointers that alias array locations must be abstracted using the `free` abstraction. During abstract interpretation, if the model-builder is unable to determine the array being aliased, the pointer is abstracted to $\top$.

The abstractions to be applied to the program variables are specified in the abstraction map file. The abstraction map has three important declarations. The `file` declaration declares the beginning of a new scope, and all the declarations following it are assumed to originate from the source file mentioned in the `file` declaration. The `fun` declaration declares a new function scope, allowing the abstractions for local variables to be specified in it. Finally, the `var` declaration is used to specify abstractions for individual variables, using the syntax described in Table 3.1.

3.2 Model Generation

Given a set of C source files, an abstraction map and a label map, model generation is done in two steps. First, the sources have to be compiled using `cwcc` and linked using `cwld` to create a *cwi* file for the entire program. The *cwi* file is a high level intermediate code in which the program is organized into basic blocks and each statement in the program can be either an assignment to a variable/register, or a control-flow statement (conditional goto, unconditional goto, function call/return). Model generation is done by `cwmb` and is directed by the abstraction and the label map files.

Model generation is done by abstract interpretation - essentially, executing statements of the program in the context of the abstract domain. Two environments, a static environment and a local environment, store information about the bindings of the abstracted variables. Changes to the bindings in the environments trigger state changes. These state changes are tracked, and a model for the program generated from them. The process of abstract interpretation is semantically sound with respect to 2's complement arithmetic, as is the symbolic evaluation of each *cwi* instruction.

As the states are generated, transitions are inserted between the current state and the generated states. The transitions are labeled as per the rules specified in the label map file, or labeled τ, if none of the rules specified in the label map are triggered.

The output of the model generation process is always a file; the format can be human-readable text, a graph renderable by *dot* [16] or *daVinci* [18], or an automaton

compatible with the Concurrency WorkBench.

3.3 Buffer overflow detection

While C Wolf allowed its users to abstract arrays, bounds checking was done if the subscript was abstracted using the integer abstraction or if it was abstracted using a carefully chosen `part` abstraction[1]. Since C Wolf was not originally intended to support bounds checking, it lacks the ability to perform bounds checking efficiently. In particular, to enable bounds checking, users had to manually specify that every variable used in the program as an array subscript was to be abstracted using an integer or a `part` abstraction. If the indices were abstracted using the integer abstraction, there was a strong possibility that a state space explosion would occur. On the other hand, indices abstracted using the `part` abstraction were only capable of handling fixed-size arrays. Also, the `part` abstraction was error-prone, because the partitions specified for an array are incompatible with arrays of a different size. As a result, if a subscript variable was abstracted using the `part(0,N)` abstraction, the variable could only be used to index arrays of size N.

In order to remove the above deficiencies, we have enhanced C Wolf to automatically abstract all variables that appear in array subscript expressions using the *Dynamic Range* abstraction, unless the variable has been abstracted using a *stronger* abstraction. We define a *stronger* abstraction as one that reflects more closely, the values of the concrete domain. Integer (`int`) and Interval (`part`) abstractions are treated as abstractions *stronger* than the dynamic range abstraction.

This step involves a static check on the block code, during which the basic blocks are checked for subscript expressions, and the variables in the expressions isolated. The (user-specified) abstraction of each variable is now obtained from the abstraction map; if the variable has a weaker abstraction than the dynamic range abstraction, it is abstracted using the dynamic range abstraction. When an array access involving a dynamic range variable is encountered during abstract interpretation, the index expression is evaluated using the semantics of the dynamic range abstraction (see Chapter 4 for more details). If the resulting abstract value has either the upper or the lower bound outside the range

[1] If the size of an array is N, we can choose a part abstraction of the form `part(0,N)` to abstract the indices of the array.

$[0 \ldots N]$, where N is the size of the array, the array access is flagged as unsafe and a message given to the user.

An interesting aspect of the bounds checking procedure is the optimizations that can be applied to it. Some of the popular methods of optimizing runtime bounds checks are those proposed by Rajiv Gupta [19] and those proposed by Micheal Wolfe and Priyadarshan Kolte [24]. Both methods use some form of compile-time optimization in the form of bounds-check elimination, bounds-check propagation, and partial redundancy elimination to reduce the overhead of bounds checking at runtime.

In C Wolf however, the size of the generated model is of greater importance than the speed of model generation (even though the time lag should be reasonable). In fact, the time taken by the model-builder is proportional to the number of states in the model. Hence, the optimizations we introduce focus less on speed and more on keeping the model size down to manageable proportions. Typically, bounds check optimization is used to eliminate bounds checks that are redundant because of some control structures. To illustrate the concept, we present an example in Figure 3.3.

```
#define N 3
int arr[N];
int i = 0 ;

for ( i = 0; i < N; i++ )
{
     arr[i] = 0;
}
```

Figure 3.3: For loop that results in redundant bounds checks

In the example of Figure 3.3, if the variable i is abstracted using the `int` or the `minmax` abstractions, the model-builder will perform N bounds-checks while interpreting the *for* loop. In reality, no checks need to be performed because the values that the subscript i can assume are limited to the range $[0 \ldots N-1]$ by the loop construct itself. Further, abstracting the array subscript using the integer or range abstraction results in the generation of thirteen states, of which six states are redundant (states 4 to 10 in Figure 3.4).

In order to minimize the number of redundant states and the frequency of the

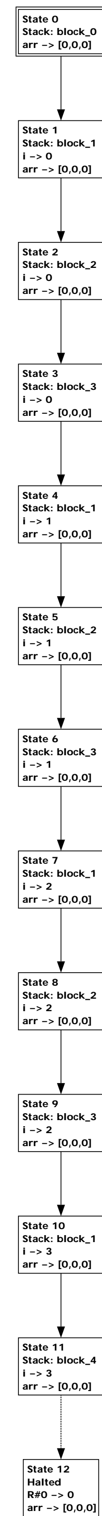

Figure 3.4: Model generated for the *for* loop with `int` abstraction

bounds check operations, we add a property to the dynamic range abstraction. Consider a variable v that is abstracted using the dynamic range abstraction. If v is compared with a variable/constant k and the result is *true*, we widen the range of v to an amount that is defined by the abstract value of k. By widening the range of v, we avoid the generation of the redundant states. This is possible because the information represented by the redundant states is now incorporated in the widened range. We also note that the relational operator used for the comparison should be an inequality operator and the variable k should not be abstracted using the dynamic range abstraction.

Specifically, in the previous case, the variable `i` has an initial value of $[0, 0]$. Execution of the statement `< (i, 3)` now sets `i` to the value $[0, 2]$, reflecting the range of values that the variable `i` can assume such that the condition $i < 3$ is still satisfied. When the index `i` is incremented inside the body of the loop, its value now becomes $[1, 3]$. The entry condition for the loop no longer holds and the loop terminates after just one iteration, needing just one bounds-check operation. The generated state diagram is also smaller, with just 7 states (Figure 3.5), illustrating the efficacy of our method. This compares favorably with 3 bounds checks operations and 13 generated states in the "normal" case.

Table 3.2 outlines the rules used when a dynamic range variable is compared with a variable whose abstract value is N.

Initial	**Operator**	**Result**
$[x, y]$	$<$	$[x, N-1]$
$[x, y]$	$\leq$	$[x, N]$
$[x, y]$	$\geq$	$[N, y]$
$[x, y]$	$>$	$[N+1, y]$

Table 3.2: Relational operators and dynamic range variables

3.4 Dynamically allocated arrays

As mentioned in [14], C Wolf only supports arrays allocated on static storage and pointers referring to such arrays. However, in most C programs arrays are allocated dynamically using heap allocation functions like `malloc` and `calloc`. No bounds analysis would be meaningful without support for such arrays. Therefore, we have added support

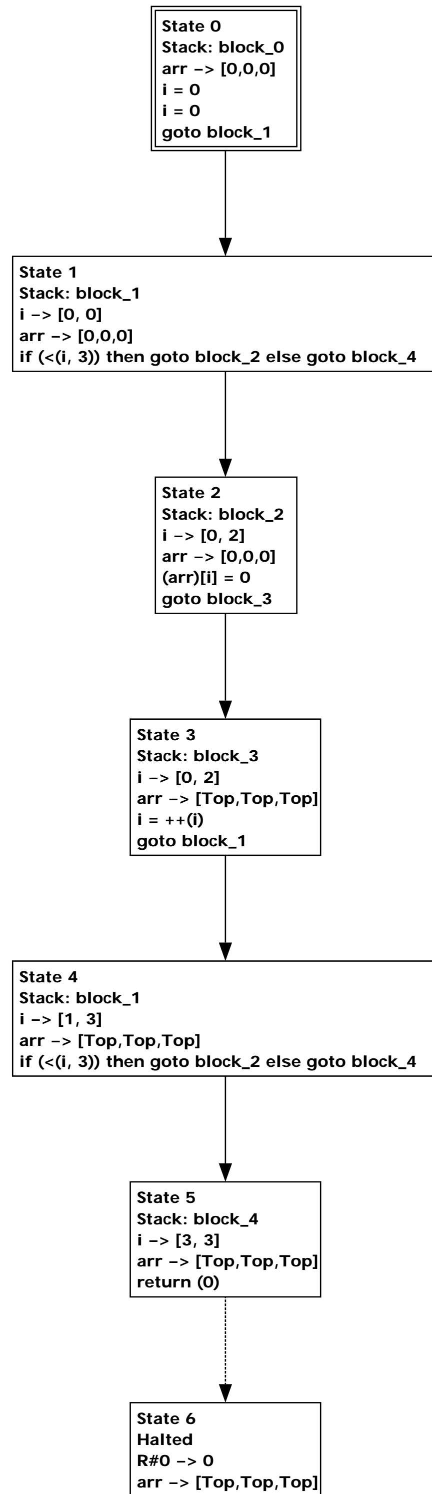

Figure 3.5: Model generated for the *for* loop with the dynamic range abstraction

for heap-based arrays by utilizing the existing mechanism for handling static arrays. Heap-allocated arrays are dealt with in a two step process:

First, we evaluate the parameter/expression passed to the `malloc` or `calloc` function to determine the total storage that needs to be allocated. Then, we find the abstract type of the pointer that refers to the allocated memory. Based on the type of the pointer we determine the actual size of the array. If the parameter passed to `malloc` or `calloc` evaluates to $\top$, or cannot be converted into an integral value, a storage of zero elements is allocated, and the variable is set to $\top$.

The "actual" array is stored in the global environment [14]. Subscript operations using this pointer (or any other pointer aliasing this one) directly work with the stored array. The pointer pointing to the array is represented by a pair with an integer representing the size of the array and an integer or character array representing the contents. Since C Wolf only deals with integral data types, such a representation is sufficient.

It must be noted that storing the array in the global environment reduces the scalability of C Wolf. However, since many string operations depend on the contents of the string, and programs that check for specific values within an array are common, we are constrained to store the contents of the array instead of abstracting them away.

Bounds checking for dynamically allocated arrays is done in the same manner as for statically allocated arrays.

3.5 Partial order analysis

C Wolf uses constraint analysis to improve the accuracy of evaluation of relational expressions [14]. To further enhance the accuracy of the process, we introduce a partial order analysis technique that stores additional information about the relationship between the variables of a program. Specifically, we intend to facilitate:

- the accurate deduction of ordering relationships between variables,
- the process of making inferences about the relationship between the value of an index and the bounds of the array being indexed, and
- the deduction of the relative sizes of any two arrays or in the case of pointers, the relative sizes of the arrays they point to.

The relationships between the variables in the program are stored in the form of a graph, called the partial order table. Relationships between variables are determined by comparing their values, while the relationships between arrays and variables are determined by comparing the abstract value of the variable with the size of the array. Relationships between two arrays are determined by comparing their sizes. Two partial order tables are used throughout the analysis, one which stores the <= relationships, and the other which stores the >= relationships. Both the tables are complementary and are actually the inverse of each other. If two variables have both a <= relationship as well as a >= relationship, they are equal. An equivalence table is also used to split the set of program variables into partitions of equal elements. The tables are modified when assignment statements, conditional goto statements, call statements and return statements are encountered.

When an assignment statement is encountered, the current relationships of the variable on the *lhs* of the statement are discarded. The relationships between the assignee variable and those on the *rhs* of the assignment statement are then determined and stored. If the *rhs* of the assignment statement has an arithmetic expression, the relationships are determined by the rules specified in Table 3.3. Currently, logical and relational expressions that constitute the *rhs* of an assignment statement are ignored.

Operation ->	**Range**	**r = x + y**	**r = x * y**	**r = x - y**	**r = x / y**	**r = x mod y**
Values						
	$\lvert x \rvert < \lvert y \rvert$	x <= r, y <=r	x <=r, y <=r	r <= x, r <=y	r <= x, r <= y	r <= x, r <= y
x > 0 y > 0	$\lvert x \rvert = \lvert y \rvert$					
	$\lvert x \rvert > \lvert y \rvert$			r <= x	r <= x	
x > 0 y = 0		x <= r, y <=r	r <= x, r <=y	r <= x, y <= r	Undefined	Undefined
	$\lvert x \rvert = \lvert y \rvert$	r <= x, y <= r	r <=x, r <= y	x <= r, y <= r	r <= x, y <= r	r <= x, y <= r
x > 0 y < 0	$\lvert x \rvert < \lvert y \rvert$				r <= x, y <= r	
	$\lvert x \rvert > \lvert y \rvert$				r <= x	
x = 0 y > 0		x <= r, r <= y	r <= x, r <=y	r <= x, r <=y	r <= x, r <= y	r <= x, r <= y
x = 0 y = 0		x <=r, y <= r	r <= x, r <=y	r <= x, r <=y	Undefined	Undefined
x = 0 y < 0		r <= x, y <= r	r <=x, r <= y	x <= r, y <= r	r <= x, y <= r	r <= x, y <= r
	$\lvert x \rvert = \lvert y \rvert$	x <= r, r <= y	r <=x, r <= y	r <= x, r <= y	x <= r, r <= y	x <= r, r <= y
x < 0 y > 0	$\lvert x \rvert < \lvert y \rvert$					
	$\lvert x \rvert > \lvert y \rvert$					
x < 0 y = 0		r <= x, r <= y	x <= r, r <= y	r <= x, r <= y	Undefined	Undefined
	$\lvert x \rvert = \lvert y \rvert$	r <= x, r <= y	x <= r, y <= r	x <= r, y <= r	x <= r, y <= r	x <= r, y <= r
x < 0 y < 0	$\lvert x \rvert < \lvert y \rvert$					
	$\lvert x \rvert > \lvert y \rvert$			x <= r		

Table 3.3: Rules for partial order arithmetic

Table 3.3 specifies the rules for updating the partial order tables when a single binary expression is encountered. Assignment statements that have complex expressions on the *rhs* are handled in a two-step process:[2]

First, when a subexpression that constitutes the complex expression is evaluated, a temporary symbol representing the value of the subexpression is introduced in the partial order tables. The relationships of the result of the subexpression with the terms that constitute the subexpression are determined by the rules of Table 3.3. These relations constitute the relationships of the temporary symbol in the partial order tables. For example, if the subexpression encountered is `a + b`, with `a` > 0, `b` > 0, and the temporary symbol representing the value of the expression is `t`, then the entries for `t` in the partial order tables will indicate that `t` is greater than both `a` and `b`. The process is repeated until the entire expression is evaluated, at the end of which, the partial order tables contain a temporary symbol that represents the value of the complex expression. When the value of the expression is assigned to the variable on the *lhs*, the relationships of the final temporary symbol replace those of the assignee variable in the partial order tables.

At this point, the partial order relationships of the assignee variable with the final temporary symbol is known. In order to determine the relationships of the assignee with the variables that constitute the *rhs*, we perform a transitive closure operation on the partial order tables. Since the $\leq$ and $\geq$ operators are transitive, the closure operation successfully determines the required relationships. The temporary symbols, having served their purpose are discarded from the tables. It is important to note that if the evaluation of any of the subexpressions results in an overflow, the process of determining partial order relations is stopped and the ordering relations of the concerned variables are discarded from the tables. Figure 3.6 shows a program with a complex expression, Figure 3.7 shows the resulting partial order table after the expression is evaluated, Figure 3.8 shows the table after the transitive closure is performed, and finally, Figure 3.9 shows the table after the temporary symbols are discarded.

When a conditional goto statement is encountered, the result of the evaluation of the conditional expression is used to update the partial order tables. However, only those expressions that evaluate to *true* or *false* are considered for this purpose. For example, if a conditional expression like `minX >= minY` evaluates to *true*, then the partial order tables

[2]We define a complex expression to be an expression that has more than one binary operator connecting its terms. For example, `a + b + c` is a complex expression.

```
void main ( void )
{
    int a = 6, b = 5, c = 2;
    int k;

    k = a + b * c ;
}
```

Figure 3.6: Program fragment with a *complex* expression

```
The >= table

  0  -> 0
  a  -> 0 | a | b | c
  b  -> 0 | b | c
  c  -> 0 | c
  k  -> T2|
  T1 -> 0 | b | c       (* T1 = b * c  *)
  T2 -> 0 | a | T1 | k  (* T2 = a + T1; k = T2 *)

(*
    Note: (1) In the table, a -> b => b >= a
          (2) The <= table is the exact inverse of
               the >= table.
          (3) T1 and T2 are temporary variables that
               represent the values b * c  and a + T1
               respectively.
*)
```

Figure 3.7: Partial order table for program in Figure 3.6

```
The >= table

  0  -> 0
  a  -> 0 | a | b | c
  b  -> 0 | b | c
  c  -> 0 | c
  k  -> T2| T1| k  | a | b | c | 0
  T1 -> 0 | b | c
  T2 -> 0 | a | T1 | k | b | c
```

Figure 3.8: Partial order table of Figure 3.7 after transitive closure

```
The >= table

  0  -> 0
  a  -> 0 | a | b | c
  b  -> 0 | b | c
  c  -> 0 | c
  k  -> k | a | b | c | 0
```

Figure 3.9: Partial order table of Figure 3.8 after the temporaries are removed

are updated with entries indicating that $minX \geq minY$.

Calls to `malloc` update the table by treating the size of the allocated space as the value of the array and recalculating the relationships of the other variables with the array variable.

In order to keep the tables consistent across procedure calls, C Wolf uses a simple interprocedural analysis technique. First, the arguments passed to the function are evaluated and the partial order tables updated using the procedure described earlier in this chapter. As a result, the abstract value of the expression that is passed to each formal parameter is represented by a unique temporary symbol in the partial order tables. Next, the relationships of the temporary symbols are copied onto those of the corresponding formal parameters. Finally, a transitive closure is performed on the tables and the relationships between the formal parameters obtained. As an example, for the program fragment of Figure 3.10, the original partial order table is shown in Figure 3.11 and the partial order table at the start of the function `func` shown in Figure 3.12.

```
void func( int p, int q, int r )
{
    p = q + r;
    r = q * p;
    return r;
}

void main ( void )
{
    int a = 1, b = 5, c = 7;
    a = func (a - b, b + c, c + a);
}
```

Figure 3.10: Function call that updates the partial order table

In Figure 3.12, it is important to note how the relationships between the formal and actual parameters, and the relationships between the formal parameters themselves have been deduced. Since `p = a - b`, `a > 0`, `b > 0` and `b >= a`, we know that `p` is negative, and `0 >= p`. Therefore, `p` is also less than `a` and `b`. Similarly, because `r = c + a`, `c > 0` and `a > 0`, we know that `r >= c` and `r >= a`. After the table has been

```
 The >= partial order table

  0 -> 0
  a -> a | 0         (* a >= 0, a *)
  b -> b | a | 0     (* b >= b, a, 0  *)
  c -> c | b | a | 0 (* c >= c, b, a, 0 *)
  p ->
  q ->
  r ->

The <= partial order table is the exact inverse of the >= table
and is not shown here. Typically if the >= table has a path from
p to q, the <= table has a path from q to p and vice versa.
```

Figure 3.11: Partial order table for the code in Figure 3.10 before the function call

```
 The >= partial order table

 0 -> 0 | p                (* p is negative *)
 a -> a | 0 | p            (* a >= p because p = a - b && (a, b >= 0,
                            and b >= a)  *)
 b -> b | a | 0
 c -> c | b | a | 0
 p -> p |
 q -> q | b | c | 0 | p    (* q >= 0, 0 >= p, => q >= p *)
 r -> r | c | a | 0 | p    (* r >= a, a >= p, => r >= p *)
```

Figure 3.12: Partial order table for the code in Figure 3.10 at the start of the function

transitively closed, we deduce that `r` $>=$ `p`, accounting for the `p` entry in the `>=` list of `r`.

Return values are handled in a similar fashion. The return expression is evaluated and its relationships with the other variables determined using transitive closure. When the return value is assigned to a variable, the relationships of the temporary symbol representing the return value are copied onto those of the variable and transitive closure performed to update the tables. For example, for the program fragment shown in Figure 3.10, Figure 3.13 shows the partial order table just before the function returns.

```
The >= partial order table

  0 -> 0                          (* p = q + r, and is >= 0 *)
  a -> a | 0 | p                  (* a is not updated yet   *)
  b -> b | a | 0
  c -> c | b | a | 0
  p -> p | q | 0                  (* because p = q + r, r = q * p *)
  q -> q | b | c | 0
  r -> r | c | a | b | q | p | 0 (* transitivity, r = q * p *)
R#0 -> r | c | a | 0 | q | p | 0 (* Return value of the function = r *)
```

Figure 3.13: Partial order table for the code in Figure 3.10 before the function return

To handle recursive functions C Wolf performs another step. Before updating the relationships of the formal parameters C Wolf stores the "current" relationships by means of a simple mechanism. The relationships of variable i are stored as that of the symbol $i + k \cdot N$ where N is the number of variables in the environment, and k is determined by the formula $\lceil (L/N) \rceil$ where L is the largest identifier in the partial order table.

While returning from such functions, the partial order tables are restored to their original state by simply copying the stored relationships back to those of the variables.

Partial order tables are consulted when the model-builder is unable to determine if the result of a conditional expression is *true* or *false*. Usually, this occurs because of the abstract values involved. For example, in the expression `a > b`, if `a` has an abstract value $[1, 3]$ and `b` has an abstract value $[2, 4]$, and both `a` and `b` are abstracted using the `minmax` abstraction, the expression can be either *true* or *false*, because the ranges overlap. In such situations, the model-builder performs a lookup on the partial order tables to see

if it can determine the result of the conditional based on the relationships stored in them. Branching now occurs based on the entries in the tables. The tables are also used to compare the lengths of arrays and to determine whether the range of an array index is within the bounds of the array being indexed.

In conclusion, we note that buffer overflow detection in C Wolf was cumbersome and error-prone earlier. We have made many enhancements to C Wolf to support bounds checking and to make the bounds checking process efficient and less tedious for the user. We have introduced a new abstraction, called the dynamic range abstraction that is automatically used to abstract array indices. We have modified the semantics of the dynamic range abstraction to minimize the increase in model size. Keeping in mind the prevalence of heap-allocated arrays, we have provided support for heap- allocation functions `calloc` and `malloc`, giving C Wolf the ability to abstract these arrays. We have also offered a small string API to abstract the behavior of the most common string functions, and provided support to abstract strings as arrays of characters. Finally, we have improved the constraint analysis procedure of C Wolf by using partial order analysis techniques. This also gives the model-builder greater information to determine the relative sizes of arrays and the relationship between the value of an index and the size of the array being indexed.

Chapter 4

Theory

In this chapter, we present some of the principles behind the abstractions presented in this work. We begin by introducing in Table 4.1, the mathematical notations used in this chapter.

Notation	**Representation**
$\mathbb{Z}$	The set of Integers
$\mathbb{C}$	The set of Characters = $\{0 \ldots 255\}$
α_A	The abstraction function for domain A
γ_A	The concretization function for domain A
$\sqcup(S)$	Least upper bound of set S
$\sqcap(S)$	Greatest lower bound of set S
$x \sqcup y$	Least upper bound of x and y
$x \sqcap y$	Greatest lower bound of x and y

Table 4.1: Mathematical notation used in this thesis

4.1 The char and char array abstractions

The character abstraction is used to model the `char` datatype in C. This is simply provided as a basic building block for the character array abstraction. It is worth noting that even though the integer abstraction would cover characters we define a character abstraction

in order to reduce the concrete domain from the set of all integers to the set $\{0 \ldots 255\}$. This also reduces the size of the model generated, as any value outside the above specified range is automatically abstracted to $\top$.

As with the `integer` abstraction [14], the abstraction and the concretization functions for characters are trivial and are given as:

$$\gamma_C(c) = \begin{cases} \{c\}, \text{ if c} \in \{0 \ldots 255\} \\ \{minint \ldots -1\} \cup \{256 \ldots maxint\}, \text{ otherwise.} \end{cases}$$

$$\alpha_C(a) = \begin{cases} a, \text{ if a} \in \{0 \ldots 255\} \\ \top, \text{ otherwise} \end{cases}$$

where α_C and γ_C are the abstraction and concretization functions respectively. The abstract domain of the `char` abstraction is the set $\{0 \ldots 255\} \cup \top$.

The standard arithmetic operators, $+, -, \div, *$ as well as the relational operators $\leq, \geq, =, <, >$ are supported for the abstraction. The semantics of the operators are the same as that of the integer abstraction with the added restriction that any value not in the set $\{0 \ldots 255\}$ is abstracted to $\top$.

4.2 The Dynamic Range Abstraction

The *Dynamic Range* abstraction is used to reduce the state size of the model when array indices are automatically abstracted. The *Dynamic Range* abstraction is similar to the *Range* abstraction which abstracts concrete values to an upper and a lower bound [14], with some additional constraints on the operation of relational operators.

The concrete domain for the *Dynamic Range* abstraction is the set of integers, $\mathbb{Z}$, while the abstract domain can be specified as the set $\{\top\} \cup \{\langle x, y \rangle \in \mathbb{Z} \times \mathbb{Z} | x \leq y\}$. The abstraction function is given by the formula:

$$\alpha_D(S) = \begin{cases} \langle \sqcup S, \sqcap S \rangle \text{ if S} \neq \phi \\ \top, \text{ otherwise} \end{cases}$$

The use of the abstraction and concretization functions for the `char` and *Dynamic Range* abstractions is given in Figure 4.1.

$$\alpha_D(8) = \langle 8, 8\rangle \quad \gamma_D(\alpha_D(8)) = \{8\}$$

$$\alpha_C(48) = \{48\} \quad \gamma_C(\alpha_C(48)) = \{48\}$$

$$\alpha_C(257) = \top \quad \gamma_C(\alpha_C(257)) = \{minint, minint + 1, \ldots, 0, 1, 2, \ldots, maxint\}$$

Figure 4.1: Application of the abstraction and concretization functions

4.3 Dynamic Range Conversions

The result of converting a value to the *Dynamic Range* abstraction is a pair, specifying an upper and lower limit on the possible values. In a manner similar to that of the *Range* abstraction [14], we specify two utility functions for each domain $\mathcal{D}$: A function $\lceil x \rceil_{\mathcal{D}}$ that computes an upper bound on x, and a function $\lfloor x \rfloor_{\mathcal{D}}$ that computes a lower bound. $\lceil x \rceil_{\mathcal{D}}$ always returns a value greater than or equal to $\sqcup\gamma_D(x)$ and $\lfloor x \rfloor_{\mathcal{D}}$ returns a value less than or equal to $\sqcap\gamma_D(x)$. These bounds will be as tight as possible. The formal definitions of these functions was given in [14] as follows:

$$\lceil x \rceil_U = maxint$$
$$\lfloor x \rfloor_U = minint$$
$$\lceil \langle x, y\rangle \rceil_R = y$$
$$\lfloor \langle x, y\rangle \rfloor_R = x$$
$$\lceil S \rceil_{M_k} = \begin{cases} maxint \text{ if } \sqcup S \geq 0 \\ \sqcup S \text{ if } \sqcup S < 0 \end{cases}$$
$$\lfloor S \rfloor_{M_k} = \begin{cases} \sqcap S \text{ if } \sqcup S \geq 0 \\ minint \text{ if } \sqcup S < 0 \end{cases}$$
$$\lceil T \rceil_{I_S} = \sqcup\{y \mid \langle x, y\rangle \in T\}$$
$$\lfloor T \rfloor_{I_S} = \sqcap\{x \mid \langle x, y\rangle \in T\}$$

M_k denotes the *mod(k)* abstraction and I_S denotes the *part* abstraction, where S is the partitioning set specified by the user.

Operation	**Result**
$\langle x, y\rangle + \langle p, q\rangle$	$\langle x+p, y+q\rangle$
$\langle x, y\rangle - \langle p, q\rangle$	$\langle min(x-p, y-q), max(x-p, y-q)\rangle$
$\langle x, y\rangle * \langle p, q\rangle$	$\langle x*p, y*q\rangle$
$\langle x, y\rangle / \langle p, q\rangle$	$\langle min(x/p, y/q), max(x/p, y/q)\rangle$
$\langle x, y\rangle ++$	$\langle x+1, y+1\rangle$
$\langle x, y\rangle --$	$\langle x-1, y-1\rangle$
$neg\langle x, y\rangle$	$\langle y, x\rangle$
$pos\langle x, y\rangle$	$\langle x, y\rangle$
$\langle x, y\rangle \text{ mod } \langle p, q\rangle$	$\langle min(x \text{ mod } p, y \text{ mod } q), max(x \text{ mod } p, y \text{ mod } q)\rangle$
$\langle x, y\rangle < \langle p, q\rangle$	$true,$ if $y < p; false$ if $x \geq q;$ otherwise, $maybe.$
$\langle x, y\rangle > \langle p, q\rangle$	$true,$ if $x > q; false$ if $y \leq p;$ otherwise, $maybe.$
$\langle x, y\rangle == \langle p, q\rangle$	$true,$ if $x = p$ and $y = q; false$ otherwise.
$\langle x, y\rangle \leq \langle p, q\rangle$	$true,$ if $y \leq p; false$ if $x > q;$ otherwise, $maybe.$
$\langle x, y\rangle \geq \langle p, q\rangle$	$true,$ if $y \geq p; false$ if$x < q;$ otherwise, $maybe.$

Table 4.2: Dynamic range operations

4.4 Dynamic Range Operators

The semantics of the arithmetic and relational operators for the *Dynamic Range* abstraction are presented in Table 4.2.

Since the *Dynamic Range* abstraction supports range-widening, the relational operators that operate on a *Dynamic Range*-abstracted variable also bind the variable to a new value, the new value being defined by the rules given in Table 3.2.

4.5 Array abstractions

The `char array` abstraction is designed to work within the current array and pointer abstraction mechanism. The semantics for abstracting character arrays is the same as those of integer arrays.

Chapter 5

Implementation

This chapter describes some details of the implementation. An overview of the important data structures, algorithms, and the C Wolf string API are presented here.

Since the thesis was an enhancement to an existing toolset, most of the data structures used in this implementation were already present in the toolset [14]. Minor modifications were done to many data structures in the model builder - notably the AbstractValue, AbstractArrayValue and AbstractIntValue structures, which were modified to handle characters, char arrays and dynamic range integers.

5.1 Partial Order Table

An interesting structure is the "graph" used to store the partial-order-tables. The graph is implemented using an adjacency list structure, and stores variable identifiers for efficient access and manipulation. The graph is defined as a *list* of vertices or nodes, each node containing a variable identifier and a list of neighbors, which are also *nodes*. In ML, the declarations are:

```
datatype node =
  Node of {
    identifier : int,
    neighbours : node list ref (*related vertices*)
  }
```

```
datatype graph = (* The graph *)
  Graph of {
    vertices    : node list ref  (* Nodes in the graph *)
  }
```

In addition to the usual graph manipulation functions like adding vertices and neighbours, and query functions like getting the neighbors of a node, the API also allows the user to check if a path exists between two nodes and construct transitive closure of a given graph.

The partial order table merely serves as a wrapper for the graph, and most of its functions use functions of the graph API to accomplish their operations. In addition to the wrapper functions, the partial order table structure offers functions to clear a variable completely from the table and to delete temporary symbols.

Since the graph is sparse, the partial order table is stored using the adjacency list representation. The implementation uses DFS to compute the transitive closure of the tables and the time taken is in the order of $\mathbb{O}(n(n+m))$.

5.2 Dynamic Range Abstraction

The next important data structure is the `IntDynamicRange` structure. Essentially, the `IntDynamicRange` type is a pair of integers, denoting the upper and lower bounds of the abstraction. The functions supported are

1. Arithmetic functions, $+$, $-$, $*$, $/$, mod, inc, dec
2. Comparison functions, $>$, $=$, $<=$, $>=$, $<$, $\neq$ and
3. functions to hash, sort, input and output Dynamic Range integers.

5.3 Design of the char array abstraction

Since C Wolf supports both arrays and pointer to arrays, the model-builder converts array values (represented by the AbstractArrayValue structure) into pointers to arrays (represented by the ArrayPtr structure). Since the string functions manipulate entire arrays

(as opposed to ArrayPtrs that only access a single element in an array), it is necessary to convert the ArrayPtr back to a character array. This also allows the intermixing of string constants and string variables in the source. ArrayPtrs are converted into a pair, the first element being a character array and the second, an index pointing somewhere in the array. A string constant is converted into a character array and a zero index. If the strings passed to the C Wolf String API do not follow pointer/array safety rules, then an ArrayBoundsViolatedException is thrown, indicating the variable that was at fault, the line in the source file where the erroneous call was made, and the type of the error.

The string "API" supported by C Wolf is a subset of the string library in C and includes the following functions:

1. `int strlen ( const char* str )` Returns the length of the string pointed to by `str`. Return $\top$ if the length of `str` cannot be determined.

   Throws ArrayBoundsViolatedException if `str` does not have a trailing *null* character. This is easily determined by checking if the *null* character is not present within the allocated size of the string. Also throws the ArrayBoundsViolatedException if `str` evaluates to $\top$.

2. `char* strcpy ( char* dst, const char* src )` Copies the contents of string `src` into string `dst`.

   Throws ArrayBoundsViolatedException if:

   - length (`src`) >= size(`dst`)
   - `src` $= \top$ or `dst` $= \top$
   - `src` does not have a trailing *null* character.

   Note: length is the length of the string as returned by the *strlen* function and *size* is the number of bytes allocated to the string.

3. `char* strcat ( char* dst, const char* src )` Concatenates the contents of `src` with that of `dst` and places the resulting string in `dst`.

   Throws ArrayBoundsViolatedException if:

   - length (`dst`) + length (`src`) >= size(`dst`)
   - `src` or `dst` evaluates to $\top$

- `src` or `dst` do not have a trailing *null* character.

4. `int strcmp ( const char* str1, const char* str2 )` Compares `str1` and `str2` and returns:

   - 0 if `str1` and `str2` have the same contents
   - -1 if `str1` $<$ `str2`
   - $+1$ if `str1` $>$ `str2`

   The $<$ and $>$ relations are defined by the ordering of the characters in the ASCII system.

   Throws ArrayBoundsViolatedException if either string does not have a trailing *null* character. Also throws the ArrayBoundsViolatedException if either string evaluates to $\top$.

5. `int strncmp ( const char* str1, const char* str2, int N )` Compares the first `N` characters of `str1` and `str2` and returns:

   - 0 if `str1` and `str2` have the same characters
   - -1 if `str1[i]` $<$ `str2[i]` for some `i` such that 0 <= i < N && $\forall j \mid 0 <= j < i, str1[j] = str2[j]$
   - 1 if `str1[i]` $>$ `str2[i]` for some `i` such that 0 <= i < N && $\forall j \mid 0 <= j < i, str1[j] = str2[j]$

6. `char* strncpy ( char* dst, const char* src, int N )` Copies the first `N` characters of the string `src` into the string `dst`.

   Throws ArrayBoundsViolatedException if:

   - min (length (`src`), `N`) >= size(`dst`)
   - `src` $= \top$ or `dst` $= \top$
   - `src` or `dst` do not have a trailing *null* character.

   Note: length is the length of the string as returned by the `strlen` function and *size* is the number of bytes allocated to the string.

7. `char* strncat ( char* dst, const char* src, int N )` Concatenates the first `N` characters of `src` with those of `dst` and places the resulting string in `dst`. If `N` >= length(`src`), then all the characters of `src` are copied into `dst`.

   Throws ArrayBoundsViolatedException if:

   - length (`dst`) + min (length (`src`), `N`) >= size(`dst`)
   - `src` or `dst` evaluate to $\top$
   - `src` or `dst` do not have a trailing *null* character.

8. `char* strdup (const char* src)` Returns a copy of the string `src`

   Throws ArrayBoundsViolatedException if:

   - `src` evaluates to $\top$
   - `src` does not have a trailing *null* character.

5.4 Structures

Earlier versions of C Wolf ignored structure variables, abstracting them away to $\top$. Most practical programs however, use structure variables, and hence a need was felt to support structures in C Wolf. The present implementation allows individual structure members to be abstracted. In order to abstract an entire structure, it is necessary to individually specify abstractions for each of its members. The abstractions that can be applied to the members are limited to the known abstractions of C Wolf, specified in Table 3.1. Furthermore, the current implementation only supports global, static structure members and does not take into account pointers to structures, nor does it handle aliasing or structure assignment. Even with these restrictions, support for global structures helps C Wolf handle a wider range of programs. Structure members can also be specified in the *watch* rules of the label map file, allowing transitions to be labeled based on the state of these members.

In order to abstract structure members, the abstraction map has been enhanced to allow variable specifications of the form `VAR ⟨objectname⟩.(⟨(membername⟩)`$^+$ `: ⟨abstract type⟩.`

A specification of the form `var p.q.s : int;` informs the model builder that

the member `s` of the member `q` of the variable `p` has to be abstracted using the `int` abstraction.

Since structure members do not find a presence in the Variables table of the *cwi* file, it is necessary to append the members specified in the abstraction map into the Variables table. Therefore, when the abstraction map is read by the model builder, we check if the expression specified in the abstraction map is a valid structure expression, and add it to the Variables table. The Variables table is annotated with a list of identifiers that have been assigned to structure members. Each entry in the list is a pair containing the member expression (as in `p.q.s`) and the identifier assigned to the it.

While evaluating expressions, structure expressions (known as *FieldExpr*s in C Wolf) are treated as variables, and the appropriate operations carried out on them using the procedure used for other variables [14].

Support for structures will be enhanced to allow pointers, structure variables, and copying structures in the future versions of C Wolf.

Chapter 6

Results

We tested the implementation on a variety of small programs. Every time, given the right abstractions, C Wolf was able to detect buffer overruns. The results of some of the executions are documented in Table 6.1.

File	**Lines of code**	**Number of overflows present**	**Number of overflows detected**	**Number of states**
malloc.c	14	1	1	8
for.c	18	3	3	13
strings.c	13	1	1	7
loop.c	13	1	1	7
strflow.c	27	3	3	

Table 6.1: Results of some C Wolf runs

We found that in general, C Wolf was able to detect buffer overflows when the buffers and the related variables were correctly abstracted. Overflows caused by indices that are out of range, overflows caused by invalid parameters to string functions, and overflows caused by simple pointer arithmetic were all detected by the toolset. Pointer aliasing too did not pose any trouble as long as the pointer and the array being pointed to were correctly abstracted.

The time taken by the model-builder to generate the model while performing

bounds analysis is usually 10-15% higher than the time taken to generate the model otherwise. However, the absolute time taken for programs that have large buffers (> 8KBytes) is significant, and runs into a few hundred seconds of CPU time. We found that this is caused by the implementation of arrays in Standard ML. The process can be speeded up by using intelligent data storage techniques or by porting to compilers that are more efficient. Porting C Wolf to more efficient compilers like OCaml is currently being investigated.

6.1 Analysis of Web2c

In order to compare Cleanness Checking with Abstract Interpretation, we used the same package, `web2c`, which was used by Dor *et al.* [12] to analyze the performance of C Wolf. The package consists of three main programs: `agrep`, a `grep` application used for pattern matching, `fixoutput`, a tool to check the output of a lexical analyzer, and `web2c`, a program used for converting TeX and other related WEB programs to C.

While C Wolf was able to detect the same number of buffer overruns as Cleanness Checking did for the `fixoutput` program, the behavior for the `agrep` program was mixed. C Wolf detected five *possible* errors in the main function of the agrep tool, but also issued nine false alarms in the process. In contrast, Dor *et al.*, are silent about these overflows. The errors were caused by calls made to string manipulation functions like `strcpy` where one of the arguments passed to the functions was an unsanitized command line argument (Figure 6.1). At the same time however, C Wolf was unable to detect an overflow in the `m_short` function that was detected by Cleanness Checking. The error was caused by an overflow in an array that was on the caller stack frame. C Wolf was unable to detect it because the model-builder does not handle modifications made to the caller stack frame from the callee function. This infact, turned out to be the biggest drawback of the current implementation.

The performance for the `web2c` sources was also not satisfactory. The `web2c` sources rely upon data read from files, and specifically, the contents of the files. Since this information is not available with model-builder, we found that some overflows in the sources went undetected. Furthermore, while analyzing the file `fixwrites.c`, the model-builder goes into an infinite loop since the terminating condition for the main loop in the program is never satisfied. However, C Wolf was able to detect three errors that were

```
void main ( int argc, char* argv[] )
{
  char Pattern[MAXPATHLEN];
  strcpy (Pattern, argv[0]);
}
```

Figure 6.1: Possible buffer overflow caused by a string operation

detected by Cleanness Checking in the `strpascal.c` file and one error in the `fixwrites.c` file.

In terms of raw performance, C Wolf proved to be faster. It must be noted though, that the performance metrics are not strictly comparable as C Wolf was implemented using Standard ML on Linux, while Dor *et al.* used C on the Windows platform for their implementation. Further, while Dor *et al.* developed their tool specifically for checking buffer overruns, C Wolf does more - it generates a model from the program. Still, we believe that the metrics give us important hints about the efficiency of Abstract Interpretation for detecting buffer overflows. Our metrics indicate a significant performance boost while using Abstract Interpretation; the maximum increase being a 1000-fold speed up in analysis time for the `fixoutput` program. Significantly, the execution time for C Wolf is the time taken to analyze the entire program, whereas the execution time for Cleanness Checking is the time taken to analyze the individual functions.

To summarize, Abstract Interpretation is faster than Cleanness Checking for detecting buffer overflows. Abstract Interpretation also detects some errors that go undetected by Cleanness checking. However, we found a significant possibility of false alarms, particularly when the values of the variables could not be ascertained during the analysis. We also found that our implementation was unable to detect buffer overflows that were caused on the caller stack by the callee function. We emphasize that this is strictly a shortcoming of the implementation and will be corrected once the model builder is able to interpret caller stack modification accurately.

Table 6.2 summarizes some of our results.

App.	Function	Cleanness Checking			Data Abstraction (C Wolf)		
		Errors found	False alarms	Time taken(s)	Errors found	False alarms	Time taken(s)
fixoutput							
	getchar	3	0	0.4	3	0	
	flush	0	0	3	0	0	
	backup	0	0	0.6	0	0	
	getstr	0	0	8	0	0	
	main	0	0	309.9	0	0	0.23
agrep	extend_re	0	0	3.2	0	0	
	init	0	0	1.3	0	0	
	countline	0	0	0.4	0	0	
	m_short	5	0	279.8	1	0	
	main	No data available			5	9	25.85
web2c	fprint_pascal_string	2	0	0.1	1	0	
	null_terminate	2	0	0.1	1	0	
	space_terminate	0	0	0.1	0	0	
	extendfilename	2	0	0.6	1	0	5
	remove_newline	0	0	3.4	0	0	
	insert_long	2	0	38.8	1	0	
	join	1	3	16	0	0	
	skip_balanced	0	1	5.2	0	0	
	bare	2	0	17.1	0	0	infinite loop
Note:	1. The time shown for C Wolf is the time taken to analyze the entire program						

Table 6.2: Comparison between cleanness checking and data abstraction

6.2 Analysis of the Pico Server

Pico Server is a compact web server written in C and is available for use on POSIX compliant platforms. The package contains many exploitable buffer overflows that may be used to allow remote attackers to gain user privileges on the web server. The sources themselves comprise of four C source files and equal number of C header files. The total length of the program is around 6500 lines.

We used C Wolf to analyze the sources of Pico Server. C Wolf was able to detect all the buffer overflow errors reported for Ver 2.0 of the program. A summary of the overflows detected is given in Table 6.3.

As shown in the table, C Wolf detects all the known buffer overflow errors in the Pico Server. Further, C Wolf also detects an hitherto unknown overflow that occurs in the `analyzeRequest` function. The overflow is caused when the program attempts to add a trailing *null* character at the end of the parameter `reqStr`. However, the code that attempts

Buffer overflow errors detected by C Wolf in ρServ						
File	Function	Known Errors detected	Known Errors missed	False alarms	Unreported errors found
main.c	main ()	0	0	0	0
	analyzeRequest()	3	0	1	1
	handleMethod()	2	0	0	0
mime.c	genMimeHeader()	0	0	1	0

Table 6.3: Buffer overflow errors detected in the Pico Server

to perform the operation attempts to write in the location `reqStr[BUFFER_SIZE]`, causing an *off-by-one* error since `reqStr` is allocated `BUFFER_SIZE` bytes of memory.

C Wolf also erroneously reports an overflow in the `generateMimeHeader` function. However, this is caused because the abstraction process is unable to interpret the operation performed.

6.3 Analysis of the FTP daemon

Our final test of the concept was with the `wu-ftpd` daemon. The FTP daemon is infamous in security circles for the many buffer overflow vulnerabilities present in it. David Evans and David Larochelle reported 25 potential problems in the wu-ftpd 2.5.0 server using the Splint tool [17]. Security warnings on many websites [28, 34] indicate that more than a dozen vulnerabilities in the FTP daemon are caused by buffer overruns.

We chose the FTP daemon to get a precise idea of the capabilities and limitations of C Wolf. The daemon uses stack based arrays, arrays of structures and a variety of other C constructs that are not supported by C Wolf. Our intention in using the FTP daemon as an example was to come up with an estimate of the improvements that were needed in the implementation to support large-scale programs.

The FTP daemon consists of a total of 25 source files and 8 header files. Of these, the `ftpd.c` and `ftpcmd.tab.c` files are most important as they contain the bulk of the

implementation. Most of the other files contain functions that communicate using ports, read from files, write logs and interact with the operating system. We chose to ignore these sources as their activities cannot be abstracted by C Wolf.

In our initial attempt at analyzing the sources, we tried to analyze the entire block of source code by abstracting all the variables that were crucial to the implementation. These include the buffers used to read in data from the network, the global `yylval` structure, and the stacks used to keep track of the states. C Wolf detected some *possible* buffer overflows in the `main` function, mostly involving the command line arguments passed to the server. While some of these buffer overflows might occur, none can be used to attack the server, as the command-line arguments are passed during startup, which is usually done by the administrators of the system.

The first attempt at static analysis was unable to detect any of the security holes that have been found in the daemon. This is because the commands passed to the FTP server are parsed by a yacc-generated parser which uses an array of structures to implement a stack. Unfortunately, this array of structures cannot be abstracted by C Wolf, resulting in its inability to detect the buffer overruns.

In order to bypass some of the limitations of C Wolf, we extracted the relevant code from parsers, hard-wired the sources to take a single command and repeated the analysis procedure. We encountered greater success this time and were able to detect the occurrence of the `glob` overflow error in the FTP daemon [27].

We learnt some important lessons from the analysis of the FTP sources. First, C Wolf needs extensions that can track arrays of structures. This single shortcoming led to the buffer overruns being undetected. Second, it is necessary to provide abstractions for C library functions, particularly those that read from streams, in order to analyze large programs. Finally, C Wolf needs abstractions to support more pointer operations, particularly modifying variables on the caller stack frame.

Chapter 7

Conclusions and Future Work

In this work, we have presented a new technique of detecting buffer overruns in C programs. Our experience with the implementation has been encouraging, with the analysis of the pico server and web2c providing us with good results illustrating the strength of the technique. The technique works with all control flow constructs, handles function calls, and simple global structures.

There were a number of surprises and lessons learned during the implementation of this work. One of the them was the power of the dynamic range abstraction. Initially, we planned to abstract array indices using the Range abstraction provided by C Wolf. However, very soon we found out that this led to large increases in model size. We therefore, had to introduce the new semantics of the dynamic range abstraction, because of which, the model size still remained within manageable amounts.

We were also pleased with the additional information derived by abstracting global structures. Almost every practical C program uses structures, and most programs use global structures to store important data. By providing the ability to abstract members of global structure variables, we found that a large class of programs could now be analyzed with ease. Support for heap allocation of arrays and strings also increased the number of programs that could be analyzed using C Wolf.

The partial order analysis techniques introduced in C Wolf did improve the evaluation of conditional expressions by a substantial amount, particularly when the expressions had variables that were abstracted using the range and the dynamic range abstractions.

However, for the other abstractions, the partial order techniques did not benefit the analysis process to the extent expected. This is because the process of deriving partial order relationships depends on the values bound to the variables involved. In most cases, the results of the conditional expressions were obtained by comparing their values itself.

7.1 Enhancements

Some major limitations of the work still remain. Limitations like lack of support for multi-dimensional arrays, modifying caller stack frames, arrays of user-defined types and pointers handling can be addressed by reworking the implementation. Only basic support for structures has been provided, and this needs to be enhanced. Similarly, interfacing with C system calls needs to be improved, something that can be done with user-annotations. The system would then provide pre- and post-conditions for every important system call that reflects the state of the environment before and after the execution of the call. Loops that are controlled by the contents of a buffer are a problem if the contents cannot be known at compile time. For example, a loop that searches for a space in a string read from a file can cause the model-builder to get into an infinite loop. In such circumstances, the model builder should exit the loop when a buffer overrun is detected.

Since static detection of buffer overflow errors is undecidable, we can only hope to push the envelope far enough to support a lively subset of C programs.

We also found that the performance of C Wolf deteriorated as buffer sizes exceeded 8Kbytes. Performance enhancements are therefore required. It remains to be seen if porting C Wolf to other ML compilers would improve performance.

Most of the enhancements mentioned in [14] are also applicable to this thesis. Any improvement in performance or any addition to the class of constructs supported will directly benefit buffer overflow detection as well.

At a conceptual level, the biggest problem with the use of data abstraction for detecting buffer overflows is the amount of effort involved in the process. Like model generation, detecting buffer overruns requires that the user know about and specify correctly, abstractions for every important variable and function in the program. While we have simplified this somewhat by automatically adding array subscript variables to the abstraction map, we still require that the user explicitly specify all the arrays in the abstraction map,

and provide C Wolf with their *correct* types. For example, the string functions would fail if a string was abstracted as an integer array. Ignoring an important variable or a function can lead to false alarms and/or false negatives. Techniques of automatically detecting and abstracting important variables (probably using some heuristics, like the frequency with which a variable is used), are worth investigating, but are outside the scope of this thesis.

7.2 Summary

This thesis is an attempt to apply abstract interpretation techniques for detecting buffer overruns in C programs. We have enhanced the C Wolf suite of model generation tools to support bounds checking. We have also shown that the system detects bounds violations provided the variables in the program are abstracted correctly and all the constructs used in the program are supported by C Wolf. We have also shown that the technique is more scalable and more accurate than some other static analysis techniques. However, we recognize that many limitations still exist, particularly with the implementation. In the future, we hope to extend the C Wolf system to improve its accuracy, scalability and enlarge the set of programs that can be successfully analyzed by it.

Bibliography

[1] anonymous. Once upon a free .... *Phrack*, 11(57), August 2001.

[2] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. *In IEEE Symposium on Security and Privacy, Oakland, California,May 2002*, 2002.

[3] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[4] blexim. Basic integer overflows. *Phrack*, 11(60), December 2002.

[5] http://www.cs.berkeley.edu/ daw/boon/.

[6] B.W.Kernighan and D.M.Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[7] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[8] http://www.compuware.com/products/devpartner/bounds.htm.

[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[10] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal*, 6(1):69–95, January 1999.

[11] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[12] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. *Lecture Notes in Computer Science*, 2126:194–??, 2001.

[13] Ray Duncan. *Advanced MS-DOS Programming: The Microsoft Guide for Assembly Language and C Programmers*. Microsoft Press, 2nd edition, 1988.

[14] Daniel C DuVarney. *Abstraction-based generation of Finite State Models from C programs*. PhD thesis, North Carolina State University, 2002.

[15] Daniel C DuVarney and S. Purushothaman Iyer. C wolf - a toolset for extracting models from c programs. In *Lecture Notes in Computer Science*, volume 2529, pages 260–275, 2002.

[16] Eleftherios Koutsofios et al. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA., November 1996. This report, and the program is included in the graphviz package, is available for non-commercial use at the URL: http://www.research.att.com/sw/tools/graphviz/.

[17] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, February 2002.

[18] M. Frhlich and M. Werner. Demonstration of the interactive graph visualization system davinci. In R. Tamassia and I. Tollis, editors, *Lecture Notes in Computer Science*, volume 894, pages 266–269. Springer-Verlag, October 1994.

[19] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[20] S. Harbison and G. Steele. *C: A Reference Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1991.

[21] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[22] http://www.infilsec.com/vulnerabilities.

[23] Michel Kaempf. Vudo malloc tricks. *Phrack*, 11(57), August 2001.

[24] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[25] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. 2001.

[26] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services, 1995. Available at http://www.cs.wisc.edu/ bart/fuzz/fuzz.html.

[27] http://www.securityfocus.com/bid/2548.

[28] http://cgi.nessus.org/plugins/dump.php3?family=FTP.

[29] Aleph One. Stack smashing for fun and profit. Vol. 7, Issue 79 Phrack 49. http://www.insecure.org/stf/smashstack.txt.

[30] http://www.rational.com/products/purify_nt/index.jsp.

[31] http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html.

[32] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[33] Hassen Saidi and Susanne Graf. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *Lecture Notes in Computer Science*, volume 1254, pages 72–83. Springer-Verlag, 1997.

[34] Security focus. http://www.securityfocus.com.

[35] http://lclint.cs.virginia.edu/.

[36] http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98_html/.

[37] http://www.cs.iastate.edu/ ghelmer/unixsecurity/unix_vuln.html.

[38] U.Shankar, K. Talwar, J.Foster, and D.Wagner. Detecting format string vulnerabilities with type qualifiers. pages 201–220, August 13-17 2001.

[39] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.