EFFICIENT DYNAMIC DETECTION OF INPUT RELATED SOFTWARE FAULTS

by

Eric D. Larson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

Doctoral Committee:

Associate Professor Todd Austin, Chair
Professor Stephane Lafortune
Assistant Professor Scott Mahlke
Professor Karem A. Sakallah
Thomas Ball, Microsoft Research

ACKNOWLEDGMENTS

First, I would to thank my advisor Todd Austin. This thesis would not be possible if it weren't for his hard work and dedication, his ability to "market" my research, and his guidance and leadership throughout my graduate school career. I met Todd the first day I was in Ann Arbor and he graciously took me under his wing as an academic advisor and instructor for EECS 470. I also greatly appreciate Todd's confidence in me, even when I wanted to shift my research area to focus more on software. All in all, Todd was a great tormentor[1].

I would also like to thank Beth, my wonderful wife. Her constant support and love for me has kept me going throughout the graduate student process. I appreciate her patience when I had to work long hours or travel without her to present at a conference. Beth, you are a great wife and I love you very much!

In addition, I thank the committee members Tom, Stephane, Scott, and Karem for serving on my dissertation committee. Your valuable comments during this process have significantly strengthened the research contained within my thesis.

I would like to thank the members of SWERVE, our software verification reading group.

1. I mean "mentor". [1]
[1] T. Austin, personal communication, 1999-2004.

In particular, I thank Karem, Brian, and Paul for the experience of learning about the issues associated with software verification and bug detection together as well as their valuable comments when I presented results from my research.

I thank my fellow graduate students within Todd's research group including Raj, Dan, Chris W., Drew, Seokwoo, Soggy, Lisa, Chris D., Leyla, and I'm sure I have forgotten some people as well. A big part of the graduate school experience has been interacting with fellow graduate students, especially when we were not talking about research. I would also like to thank Keith, my project partner for several courses during my first two years. I would not have obtained the high grades I received if it were not for him. I also thank the National Science Foundation for their financial support throughout my graduate studies.

Finally, I would like to thank my parents Dave and Kathy and my two brothers Mark and John for their love and support, not just during the graduate school process, but the rest of my life as well. They passed their commitment to education along to me and I'm extremely grateful for their strong desire to have me continually learn and succeed.

TABLE OF CONTENTS

Acknowledgments iii

List of Figures ix

List of Tables xi

List of Appendices xiii

Chapter 1. Introduction 1

Motivation 1

Input Related Bounds Faults 5

Contributions 11

Overview of the Thesis 15

Chapter 2. Background and Related Work 19

Static Bug Detection 22

Dynamic Bug Detection 33

Testing and Debugging 37

Chapter 3. Dynamic Detection of Input-related Software Faults 41

Tracking Integer Input to Detect Array Overflows 42

Detecting Misuse of String Functions 48

Other Improper Uses of Input Data 56

Limitations 58

Chapter Summary 60

Chapter 4. MUSE: A General Purpose Instrumentation Infrastructure 63
MUSE Implementation 64
Correctness Specification Language 66
Creating Correctness Specifications 76
Benchmarks and Testing Methodology 77
Performance Impact of Simplification and Instrumentation Calls 81
Example Application: Memory Access Checking 85
Chapter Summary 91

Chapter 5. Implementation and Results 93
Implementation 93
Validating the Implementation 96
Bugs Detected 97
Comparison to Other Approaches 101
Preliminary Performance Results 102
Chapter Summary 106

Chapter 6. Efficient Dynamic Bug Detection 107
Compile-time Identification of Input-Derived State 108
Identification of Variables Used in Dangerous Operations 113
Dflow: Global Dataflow Analysis Tool 116
Implementation and Validation 120
Performance Results 122
Chapter Summary 126

Chapter 7. Efficient Management of Shadowed State 127
Shadowing Local Variables by Name 127
Implementation 130
Performance Results 131
Chapter Summary 134

Chapter 8. Conclusion 137

Summary of the Thesis 137
Future Directions 139

Appendices. 143

Bibliography 177

LIST OF FIGURES

Figure 1.1: Example of an array bounds error. 3

Figure 1.2: Attacking a stack buffer overflow 6

Figure 1.3: Detecting an array bounds error. 8

Figure 1.4: Fault due to improper use of string library functions. 9

Figure 1.5: Detecting a string copy error. 10

Figure 2.1: Finite state automaton model for correct locking behavior. 26

Figure 3.1: Program states for integer variables. 43

Figure 3.2: OpenSSH channel bug. 47

Figure 3.3: Example of detecting string bugs. 55

Figure 3.4: OpenSSH challenge bug. 58

Figure 3.5: Example of an unsound control path. 59

Figure 4.1: Non-assignment patterns in the specification language. 68

Figure 4.2: Assignment patterns in the specification language. 70

Figure 4.3: Pattern file for null checker 78

Figure 4.4: Instrumentation functions for null checker 78

Figure 4.5: Sample program for use with null checker 78

Figure 4.6: Sample program simplified (l) and instrumented with null checking (r) . . . 79

Figure 4.7: Pattern file for memory access checker (part 1) 88

Figure 4.8: Pattern file for memory access checker (part 2) 89

Figure 6.1: Tainted propagation lattice 109

Figure 6.2: Tainted propagation algorithm 110

Figure 6.3: Danger propagation algorithm 114

Figure 6.4: Performance improvement from removing unneeded instrumentation 123

Figure 6.5: Dynamic instruction count improvement from removing unneeded instrumentation 123

Figure 7.1: Shadow state management performance improvement 132

Figure 7.2: Shadow state management dynamic count improvement 132

Figure 7.3: Hash table accesses with respect to unoptimized instrumented program. . 134

LIST OF TABLES

Table 3.1: Representative rules for computing bounds on integer variables. 44

Table 3.2: Representative Rules for string buffer overflow checking. 50

Table 3.3: String function rules for string buffer overflow checking. 53

Table 4.1: Instrumentation functions parameters. 74

Table 4.2: Programs used during testing 80

Table 4.3: Effects on run-time performance when adding instrumentation. 82

Table 4.4: Effect on dynamic instruction count when adding instrumentation. 83

Table 4.5: Effect on compile time when adding instrumentation. 84

Table 4.6: Effect on code size when adding instrumentation. 85

Table 5.1: Bugs detected during testing 98

Table 5.2: Comparison to other bug detection approaches 102

Table 5.3: Preliminary run-time performance results. 103

Table 5.4: Code size and instrumentation site counts 103

Table 5.5: Preliminary dynamic instruction count performance results 104

Table 5.6: Breakdown of dynamic instrumentation calls. 105

Table 5.7: Percentage of useless dynamic instrumentation sites 105

Table 6.1: MUSE handling of integer statically derived attributes 116

Table 6.2: Performance comparison from removing unneeded instrumentation 123

Table 6.3: Breakdown of tainted and dangerous integers 124

Table 6.4: Impact on instrumentation sites from removing unneeded instrumentation 125

Table 6.5: Dflow analysis time 126

Table 7.1: Shadowed state management method based on type 129

Table 7.2: Shadow state management performance comparison 132

Table B.1: ZeroOpStmtCode: statements that take no operands 153

Table B.2: OneOpStmtCode: statements that take one operand 154

Table B.3: FuncExprCode: expressions that take a function name as a parameter 154

Table B.4: FuncOpExprCode: expressions that take a function name and an operand . 154

Table B.5: ZeroOpExprCode: zero operand expressions 154

Table B.6: OneOpExprCode: single operand expressions 155

Table B.7: TwoOpExprCode: binary operand expressions 155

Table B.8: LhsOneOpExpr: LHS single operand expressions 156

Table B.9: LhsTwoOpExprCode: LHS binary operand expressions 156

Table B.10: TypeCode: type identifiers 156

Table B.11: WildCardIdent: wildcards to represent classes of variables 157

Table B.12: NormalParm: parameter macros that take no additional arguments 157

Table B.13: IntParm: parameter macros that take an integer argument 157

Table B.14: TypedParm: parameter macros that take a type argument 158

Table B.15: TypedIntParm: parameter macros that take a type and integer arguments 158

LIST OF APPENDICES

Appendix A: Elemental C 145

Appendix B: Specification Language Grammar 151

Appendix C: Input Checker Pattern File Annotated 159

CHAPTER 1

INTRODUCTION

1.1 Motivation

Bugs in software can have devastating effects in today's world. Computer viruses and malicious users can exploit software bugs to run harmful code or gain access to restricted data. Even an extremely strong protocol can be compromised if the implementation of the protocol contains bugs.

Computers increasingly are being used to store vast amounts of data and information. Some information is highly sensitive and access should be granted to a select few. Secure systems must protect private data from users who do not have proper access. To accomplish this, secure protocols are developed that protect against a variety of malicious attacks that attempt to gain access to sensitive data.

In addition to the threat of attacks, substantial time and money is spent keeping software up to date. One report [30] estimates that software bugs cost the United State economy $59.5 billion annually. Even bugs without security implications can be problematic for users and companies. According to a PC Magazine article [28], Microsoft founder Bill Gates was quoted to claim that on average 5% of all Windows machines crash twice a month.

In order for secure protocols to fully protect against malicious users, they must be perfectly implemented in software - a very difficult task. Most of the errors detected in secure systems are implementation errors. While some of the errors are due to improperly matching the specification, many errors are caused by complications introduced when programming - especially when using an inherently unsafe language like C. If programmers are not careful, it may be possible for an attacker to overwrite regions of memory in such a manner they are able to execute arbitrary code and gain access to private data.

An example of a common bug is shown in Figure 1.1. This code segment will cause an array bounds error if the value of four is provided as an input since the array will be indexed with the value five which is outside the bounds of the array.

Complete software verification for arbitrary programs with unbounded memory is undecidable. Frequently, software verification is limited to a set of properties that must be satisfied for correct behavior. Some examples include memory access checking (pointers must point to valid data when dereferenced), array overflows, and proper use of network protocols. These properties exhibit well-known behavior that can be easily specified and are a common source of bugs.

Software bug detection can either be performed statically or dynamically. Static techniques allow the software developer to prove that a program correctly satisfies a given property. However, the number of possible states that must be searched to obtain such a proof is often extremely large even for a simple property. As a result, the verification of a property can be infeasible. Abstraction of code not relevant to the property can greatly reduce the search space but does not necessarily make searching feasible. Further abstrac-

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

Figure 1.1: Example of an array bounds error. This code segment will overflow the array if `x` is 4.

tions are used that limit the scope of the search. The ability to prove that a property is satisfied may be lost but previous efforts have shown that a large number of bugs can be found.

Refer back to the example in Figure 1.1. This error in this particular example would be relatively straight-forward to catch using static bug detection. The tool would have to track the array size of five and track the range of possible values that `x` may hold. When it executes the array reference, it will find that there is a value of `x` that can cause an array bounds error. A nice property of static bug detection is that, once the error has been properly fixed, the array reference is guaranteed to not have any bounds errors.

The general problem of proving all array indexing in a program to be safe is a much more difficult problem. For instance, many arrays are created dynamically meaning that the size is only known at run-time. In a static tool, the size must be represented symbolically requiring symbolic analysis between the size of the array and any variable that is a function of the array size. Another issue is that the array reference could be in a different part of the program, several statements and function calls later - some level of interprocedural analysis is necessary. As an example, a false error report would occur in our example if the tool was not aware that `fatal` terminates the program. Another difficulty with static analysis is modeling data on the heap. Imagine the input variable is stored deep within a heap structure. In order to properly verify the array index, the static analyzer would not only

have to model all possible heap structures but model all of the possible values that a heap variable may hold. In general, there are too many configurations to analyze, making the problem computationally infeasible. Typically, the scope needs to reduced for the whole program (not just for the heap data as outlined here) causing either bugs to be missed or a large number of false error reports.

Other tools look for software errors at run-time or dynamically. While their effectiveness is obviously restricted by the particular input set, they are effective at finding bugs on the common paths of execution. Unlike static techniques, dynamic techniques do not need to limit the scope of the search. Dynamic checkers can find bugs that span multiple function boundaries, library functions, or even process boundaries.

The largest downside to dynamic bug detection is its dependence on the input. For example, the code sequence in Figure 1.1 only triggers an error when the input value of `x` is four. In order to find the bug, it is necessary to run a test where four is assigned to `x`. This dependence is even more pronounced when the array reference is in a different section of the code that the input routine. In addition to the required data value, the proper path is necessary to ensure that the array reference is executed. Thinking about our running example, there may only be one possible path that both increments `x` and accesses the array. Furthermore, executing that particular path may depend on the values of other variables besides `x`.

Another issue is the actual detection of the bug. In Figure 1.1, writing outside of the bounds of the array may not cause an error with respect to the program output, depending on how memory is laid out for a particular run of the program. Therefore, it is important to

develop dynamic bug detection tools that detect bugs at the array reference rather than comparing the output to an expected output.

Since additional computation is necessary when detecting bugs dynamically, the impact on performance is another downside. In our running example, the bare minimum is a check at the array reference to determine if the current value of `x` exceeds the bounds of the array. Additional instrumentation that stores the array size would be necessary if the array was created dynamically. More sophisticated approaches require even more instrumentation. As a result, there exists a trade-off between quality of bug detection and overall performance impact.

1.2 Input Related Bounds Faults

Many software defects are related to improperly bound inputs that are either not checked or checked incorrectly. A highly visible and damaging example of this type of defect includes improperly bounded checks on network data. A common example of this class of defect is the buffer overflow - a situation where memory outside of the intended buffer (or array) is accessed. Unfortunately, these types of problems occur too often. Six of the eight Microsoft security bulletins already released by March 2004 [68] are due to input not being properly checked. The Linux kernel has also had buffer overflow problems that can be exploited [52].

Malicious users can exploit this bug by overwriting stack buffers in a way that overwrites the function return address to direct control to arbitrary code. This process is illustrated in Figure 1.2. The code sequence is shown in part (a), and it contains two functions: `foo` and `bar`. The function `bar` has a very basic buffer overflow error - the `gets` function is always

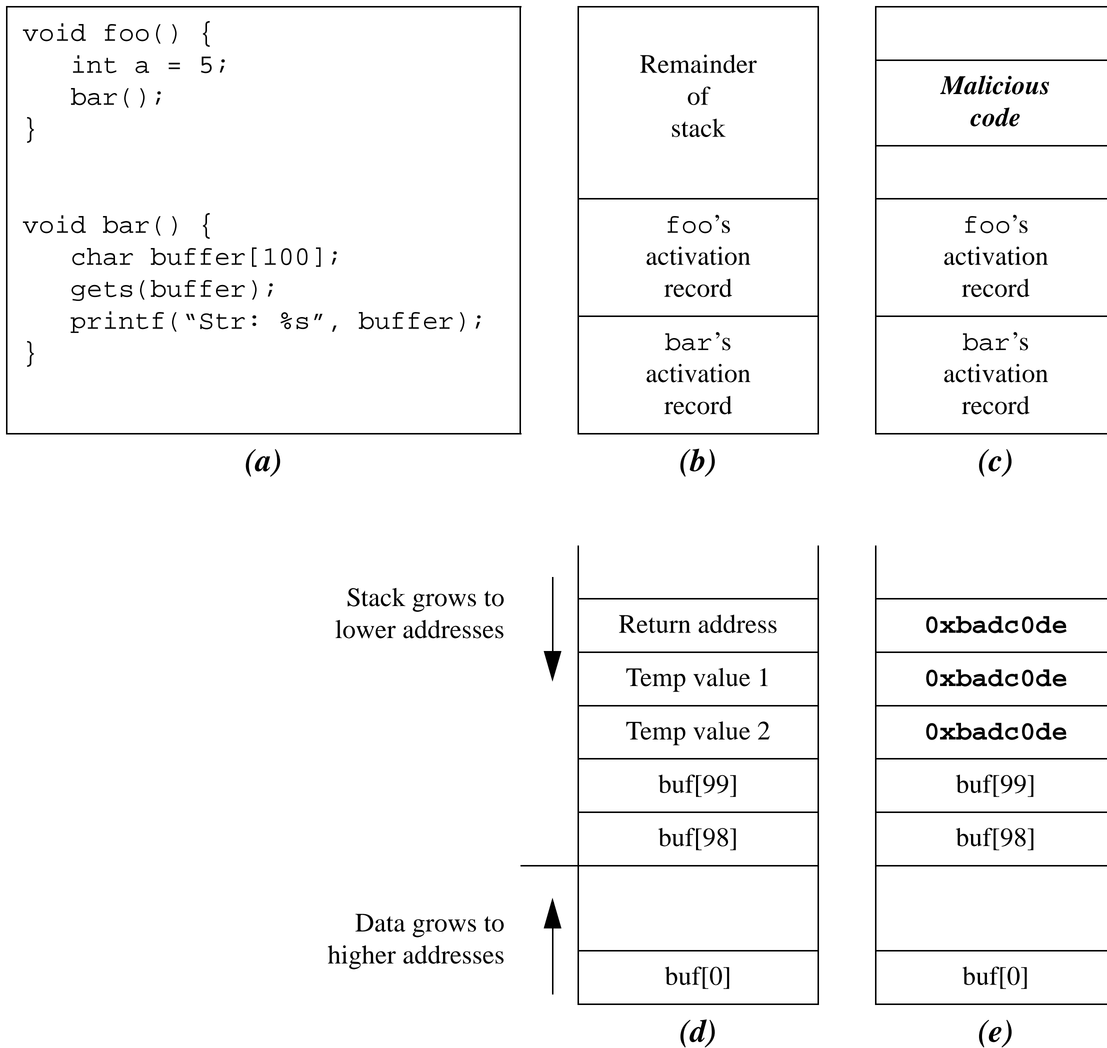

Figure 1.2: Attacking a stack buffer overflow.

unsafe since there is no check or way of restricting the size of the input. If the user inputs a string that is over 100 characters, the buffer will overflow. Before the buffer overflow occurs, the state of the stack is shown in part (b). At this point, the current activation record is that of function `bar`, which is shown in more detail in part (d). The activation record contains the return address, space allocated for local variables (the buffer in this case), and perhaps space for temporary values the compiler may need. Assume for illustrative purposes, the compiler creates two temporary values. In general, the layout of the activation will vary on the compiler and the optimization settings used to compile the

program. On many machines, however, the stack will grow to lower addresses and consecutive data in an array will grow to higher addresses.

In order to carry out an attack on the stack, two actions must be taken. The first is to overwrite the return address with an address to a higher address that resides on the stack since it might not be possible to know the precise layout of the activation record, an attacker will start overwriting the memory locations following the buffer with the same address in the hope that one of the locations represents the return address. This is illustrated in part (e) of Figure 1.2. Both temporary values and the return address are overwritten with the address `0xbadc0de`. The second part of carrying out an attack is to insert the malicious code onto the stack. Often this is a lone system command that executes a shell allowing access to the system. Since the attacker will not be able to precisely pinpoint the location where to start the code, a series of no-op instructions are prepended to the malicious code. The attack will work as long as the jump is to somewhere in the middle of the no-ops. For more information on attacking buffer overflows, the curious reader is encouraged to read [1].

To prevent a buffer overflow exploit, it is necessary for the program to check input data to ensure it does not exceed the bounds of any buffer it may be used to reference. However, many programs either fail to check input data or check the data incorrectly. The example in Figure 1.1 is a case where the data was checked but the check was wrong. This type of error can happen if the programmer writing the check was not aware that the input value was incremented before it accessed the array.

Our work centers on detecting bugs caused by improperly bounded input. It works by shadowing all input values (and variables derived from input) with a shadow state vari-

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

Figure 1.3: Detecting an array bounds error. The error is detected even though the input value of 2 does not directly cause an error.

able. Shadow state variables are introduced into the program when external inputs are read. External inputs encompass a variety of sources: command line arguments, input files, environment variables, and network data.

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

In Figure 1.3, we show how our technique can find the off-by-one error in the code segment from the example in Figure 1.1. In a conventional dynamic bug detection implementation, an error will not be detected unless `x` is four. When the value is first read from input, it is given a range to span all possible values. At the control points (after the `if` statement in the example), the interval constraint can be narrowed because the value of `x` is now known to be ≤ 4. When the value of `x` is incremented, the interval is adjusted up by one for both the upper and lower bounds. When the array access occurs, the interval of `x` is

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

Figure 1.4: Fault due to improper use of string library functions. If `src` has 16 characters (not including the null character), it will get copied into `dest` without a null character causing a problem in the subsequent `strcpy` function.

compared to the size of the array. Even though `x` has the legal value of two, an error is flagged since it is possible for the input to be five, which exceeds the bounds of the array.

Another common source of security bugs is improper use of the string library functions in C. Since the string functions provide no checking, the responsibility for safe behavior resides with the programmer. To complicate matters, there is little consistency on how the different string functions operate. For instance, the `strcpy` command always copies a null character but `strncpy` will not copy the null character unless one is present within the specified limit. An example of a bug involving strings is shown in Figure 1.4. In this case, there is a check to filter out strings that are greater than 16 characters. However, the `strlen` command does not count the null character. If the source string is exactly 16 characters (not including the null character), it will pass the check though it contains 17 characters, including the null character. As a result, the null character does not get copied by the `strncpy` command, creating a potentially dangerous `strcpy` because the source is not null terminated. This type of problem is difficult to catch during testing since it requires a source string of exactly 16 characters. Also, the bug may not manifest in an error in the output when such an input is presented; this is likely the case if the character after the `temp`

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

Figure 1.5: Detecting a string copy error. The error is detected because it is possible for the `strcpy` function to execute with a source that is not null terminated.

array happens to be a null.

While it is possible to catch faults involving strings using our array reference checking, we can find more bugs if we model strings separately, taking advantage of extra information provided by the strings and string library functions. For example, many string functions require source strings to be null terminated. Our array reference checker does not analyze the elements of the array, making it more difficult to find bugs related to null termination.

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

length that can trigger an error.

Figure 1.5 shows how to find the error in the example presented in Figure 1.4. Assume that the input to the function (`src`) is a null terminated string consisting of eight characters taken directly from input and that the string has not been constrained in any way. As a result, `src`'s maximum size will be unbounded and the `known_null` flag will be true upon entry to the function. Though the user entered an eight character string, it could have been a string of any length. Since the string has eight characters it passes the check but its maximum size is reduced to 17. (We count the null character in our definition of string size.) In the `strncpy` function, the maximum size of 17 can exceed the available destination size limit of 16. Consequently, there is no guarantee that the null character is copied, and the null flag remains off for the array `temp`. This leads to an error when `temp` is used in the `strcpy` function since it may not be null terminated. Even though the input string of eight characters does not expose the error, our approach still detects it because all possible string lengths are verified. It is also worth noting that our approach would detect an error if the string `src` were directly copied into `dest` in the `strcpy` instead of using the `temp` array. In this case, the error would get signalled because the maximum size of the source (17 characters) is greater than the destination (16 characters).

1.3 Contributions

The primary goal of this thesis is to develop and evaluate an effective and efficient technique for detecting errors related to input. A special emphasis is placed on buffer overflows due to their potential of being exploited by a malicious user. We chose a dynamic approach but we sought to find improvements that addressed some of the major draw-

backs associated with dynamic bug detection. Of the drawbacks, the dependence on the input was our largest concern followed by the performance of the instrumentation. In order to accomplish this work, it was necessary to build an infrastructure to instrument the code. When creating this infrastructure, it was designed to be flexible and general purpose so it can be used to catch different types of bugs and perform other instrumentation tasks. The remainder of this section highlights the four main contributions of this thesis.

1.3.1 Dynamic approach to finding input-related faults

Our approach for dynamic bug detection focuses on properly constraining input data to ensure that it is not possible for a user to supply an input that can cause a memory access that is outside the allowable bounds of the program. Our technique possesses the scope and program knowledge of a run-time technique while relaxing the requirement that the validator specify a set of inputs that exposes the defect.

Our system combines ideas from static bug detection and implement them in a dynamic environment. We implement our approach by shadowing variables derived from input with additional state that stores the allowable bounds for the given variable. The bounds get adjusted during arithmetic operations and get narrowed during control operations. At dangerous operations such as array references or string copies, the entire range of allowable values is verified to make sure that memory is not accessed outside of the proper bounds of the array or string. This allows validators to find bugs without having the precise input necessary to expose the bug.

Our approach is generic and can be applied to all programs. We have applied it to 9 programs and found 17 bugs including two major security bugs in a recent release of

OpenSSH. It is portable and does not require any modifications to the source code. Unlike techniques that are designed to prevent malicious behavior, our technique is intended to be used to find faults before software is released. This work was presented at the 2003 USENIX Security Symposium [63].

1.3.2 General-purpose instrumentation infrastructure

One of the products of this research is our dynamic instrumentation infrastructure called MUSE. Unlike most dynamic tools that focus on one particular property, our system is general purpose. A user, using our specification language, can specify any safety property they desire. Our system can be used to catch security related bugs and memory access violations using correctness specifications we have created.

MUSE was created in order to facilitate the rapid development of software verification tools. MUSE was used in all other aspects of my research presented in this thesis and we plan to continue to use MUSE in further research in software fault detection. Eventually, it can also be used by other research groups to assist their efforts in software verification. In addition to creating software bug detection tools, MUSE can also be used for other activities such as debugging and profiling.

Specifications are created by producing a list of patterns that match against statements, expressions, and other events in the program. When a match occurs, a call to an external function written in C is made. Since validators can write any instrumentation they desire, they have the ability to create powerful models of correctness.

1.3.3 Removing unnecessary instrumentation to improve performance

Our third contribution lessens the impact dynamic instrumentation has on the performance of the program by removing instrumentation that is unnecessary. Static analysis is used to determine which variables are derived from input. This can be done using an algorithm similar to constant propagation but propagates a tainted attribute that indicates the definition comes from input. Variables that do not contain program input do not need to be shadowed with additional state. A similar analysis finds variables that never produce a result that will be used (either directly or indirectly) in a dangerous operation. These variables also do not need shadowed state. During instrumentation, only statements that manipulate variables that both contain input data and produce a result that will eventually be used in a dangerous operation need instrumentation.

Our analyses show that at least 83% (dynamically) of instrumentation calls are not needed. With our static analysis routines, we reduced the number of instrumentation calls by 68%, improving performance by 50%.

1.3.4 Improved management of shadow state

Effective management of shadowed state, the additional state needed by the checker in order to find bugs, will also improve performance. Many dynamic bug detection systems access shadow state using a table indexed by address. This can be slow considering the number of accesses necessary by instrumented programs. Other dynamic bug detection systems manage shadowed state by increasing the size of the variable and embedding the extra state with the variable. This reduces access time but creates a compatibility problem when executing functions and system calls that were not instrumented.

We combine these two approaches by managing shadow state for local variables by name instead of by address. For each local variable that needs shadowed state, a temporary variable is created within the compiler that corresponds to the shadow state of the variable. This avoids an access to a shadow state table improving performance and does not modify the original variable, maintaining compatibility. Variables on the heap resort to being accessed by address using a table since they are unable to be accessed by name.

Implementing the improved shadow state management scheme described in this section improved performance by 33%. Combining the improved shadow state scheme with the useless instrumentation removal improved the runtime performance overhead by 58%.

1.4 Overview of the Thesis

This thesis contains eight chapters. Chapter 2 presents a tutorial on software bug detection. It gives the reader the necessary background on the problems encountered in software bug detection, definitions of the terminology used in the remainder of this thesis, and solutions from other research groups. The chapter closes with a section describing other areas related to software bug detection: testing, debugging, and instrumentation.

We introduce our high coverage dynamic approach to detecting security faults caused by improperly bounded inputs in Chapter 3. The chapter includes representative rules and their corresponding actions. Three examples of how bugs are detected are given in the chapter, including the two bugs found in OpenSSH.

Chapter 4 describes our instrumentation infrastructure, called MUSE. The MUSE tool allows users to specify safety properties using a pattern definition language and external

instrumentation written in C. This chapter describes the pattern language in detail and can be used as a reference for users wanting to create their own specifications. In addition, two specifications are described in the chapter. One is a null checker with function tracing. It is a simple specification (only four patterns) that illustrates how to use MUSE. The second is a memory-access checker similar to common memory-access detection tools such as Purify [45] and Safe C [4].

Chapter 5 describes the implementation of the input bounds checker is described in . In addition, the chapter gives results from using our checker, describing the 17 bugs and 6 false alarms we found while testing 9 programs. Performance results show that the overhead is very high but that there is room for improvement as many of the instrumentation sites were found to be unnecessary.

Improving instrumentation performance is the topic of Chapter 6. The algorithms for determining which variables hold input data and which variables hold results that could eventually be used in a dangerous operation are presented. In Chapter 7, our approach for improving management of shadow state is described in detail. Experimental results from this chapter explore the set of optimizations by analyzing the performance improvement, number of instrumentation sites, and number of accesses to the shadow state table across all benchmarks.

Chapter 8 summarizes the thesis and presents options for further research. The thesis also includes three appendices. The first appendix describes the grammar for a simplified form of the C language that is used to simplify the implementation of MUSE. Additional notes explains some of the decisions we made. The second appendix presents the grammar for

our specification language including tables that define the main keywords used in the language. Lastly, the third appendix shows an annotated pattern file for our input checker. In addition to showing how the input checker was implemented, it serves as a good reference and example for the creation of other specifications.

CHAPTER 2

BACKGROUND AND RELATED WORK

Verification is the process of determining if a product (in this case, software) is functionally correct or not. This often involves proving the program works correctly. Since it is undecidable to prove that an entire program works, software verification systems prove that a program is free of a certain type of bug under a set of assumptions. For example, a system may guarantee that a program is free of buffer overflows under the assumption that the system libraries are correct and all arrays are indexed using integers. Verification is one part of the *validation* process. Validation seeks to insure that *all* goals of the product are satisfied. In addition to functional correctness, validation involves the examination of attributes such as performance goals, compatibility concerns, and customer needs. *Testing* is the execution of a program using different inputs. While testing is used to find incorrect functional behavior, it cannot be used to fully verify anything but a trivial program.

Verification schemes can be gauged by two parameters: soundness and completeness. A scheme is *sound* if it can find every possible error in the program. A scheme is *complete* if every error it reports is an error that can actually occur during a run of the program. A complete program verification does not produce *false alarms* (a reported bug that is not actually a bug). We will use the generic term *bug detection tool* for any tool that is capable

of finding bugs. The term *verifier* will be reserved only for those tools that are able to prove that the program is free of bugs.

In order to verify a program, it is necessary to know what is considered correct behavior. A complete specification of a program is typically as complex as the program itself. This means errors are just as likely in the specification as in the program itself. As a result, matching the program against the correctness specification will not guarantee the program is free of errors. Instead, software verifiers and bug detection tools check a small number of important properties that are easy to specify, have well defined behavior, and are common sources of errors within programs. Some properties include memory access checking, buffer overflows, proper locking behavior, and program invariants. Soundness and completeness are computed with respect to the particular property that is being verified [49].

Specifying a model of correctness can be done in a variety of ways. In some cases, it is built right into the tool, especially if the tool is specifically designed to catch a particular type of error. The advantage of this approach is that the notion of correctness is tightly integrated allowing for a faster tool. Alternatively, correctness properties are specified by hand and either reside in an external file or are embedded into the program via annotations or programming language extensions. The advantage of the external file approach is that the property is specified in one place and can be written by someone who knows the protocol well. Programmers can concentrate on writing their code and rely on the checker to see if they violated any specified properties. Using annotated code forces the programmers to specify the properties. While this places an additional burden on the programmers, it forces them to think about the particular properties which may result in higher quality

code. Another advantage of this approach is that the model and the program are combined into a single entity which can simplify later phases of the process and serves as good documentation for future programmers.

Some work has been done to infer the correctness specifications from the code itself. Engler *et. al.* [33] developed an approach to look for patterns in the code and deviations from these patterns. Potential bugs are sorted based on how likely the deviation is a bug. The resulting list must then be processed manually to determine if the deviation is actually a bug. While their system has found several bugs, it suffers from a large number false alarms.

In many cases, verifiers are only able to prove a program is *safe* under a set of assumptions. In general, these assumptions include that the following entities are free of bugs:

- System libraries: The source code is often not available for system libraries and thus cannot be checked. Some systems provide interfaces that describe the behavior of well-known system calls.

- Compilers: The task of verifying the compiler and its optimizations is dealt with in [61, 97].

- Specification: Insuring that the correctness specification itself is free of errors is a non-trivial problem since it has to be done manually. Fortunately, most properties being checked are simple and well-known.

- Bug detection tool: Insuring that the verification tool is free from errors also can be a significant problem. The algorithms used by a verification tool are often proved to be correct. However, tools can still be subject to implementation errors.

For the remainder of the thesis, we assume that the above entities are free of bugs. Since the specification and the bug detection tool were created as part of our research, a brief description of how these were validated are described later in thesis.

Software bug detection tools can operate statically (at compile-time) or dynamically (at run-time). Each approach is explored in the next two sections: static bug detection in Section 2.1, dynamic bug detection in Section 2.2. Related work on testing, debugging, and program instrumentation is presented in Section 2.3.

2.1 Static Bug Detection

Though there are numerous approaches to static software bug detection but they generally follow the same high-level view process. The basic idea is to create a model of the program that is analyzed with respect to the correctness specification. Different schemes use different forms of analyses such as a SAT solver, a theorem prover, a model checker, a path simulator, or a static analyzer. The type of analysis will dictate the type of model that is used. For a theorem prover, the model will be a set of verification conditions. For a static analyzer, the model of the program may be an intermediate representation of the program. Good overviews of the software verification process can be found in [49, 92].

An advantage of static bug detection schemes is that there is no dependence on the input to the program. Inputs to the program can be represented symbolically and the analysis can

treat such variables in a manner that allows checking across the range of all possible values a variable can hold. This style of analysis can, in principle, prove that a program satisfies the given properties - the ultimate goal of static verification.

The goal of proving a program is safe is difficult to obtain in practice. Regardless of the verification method, static bug detection is computationally infeasible except for trivial programs or very simple properties. In essence, the static tools must do the equivalent work of traversing all of the paths of the program. To ensure soundness and completeness, exponential time or space algorithms are typically required depending on the property verified.

To address this problem, tools often employ an abstraction phase that removes portions of the program that are not relevant to the property being verified, leaving only sections of the program that are relevant to checking the desired property.

One technique for performing abstraction is program slicing [87]. A slice is a set of variables or objects and its formation is guided by the correctness specification. Only code that affects variables in the program slice are kept, including control flow. Program slicing may be unsound, incomplete, or undecidable depending on the implementation, program, and correctness specification. Java Pathfinder [12] and Bandera [23] are two systems that use program slicing.

Another popular technique is predicate abstraction [37]. Using the correctness specification, predicates are extracted from the program. Ranges for variables are reduced to boolean values based on the predicates creating a boolean program. Predicate abstraction is

sound but incomplete since it may add infeasible paths. The SLAM project [9] uses predicate abstraction.

Generalized data abstraction reduces the range of variables to a set of interesting abstract values. For example, an integer variable could be replaced with an abstract variable with three possible values: negative, zero, or positive. This approach is sound and incomplete and is often used in conjunction with other abstraction techniques.

In order to make the problem decidable and computationally feasible, various parameters can be restricted to a maximum number. Some example parameters include the lengths of lists, number of processes, size of user input, etc. This reduces the scope of the model but is neither sound nor complete. The rationale for such restrictions is that if a bug resides in a list that can be of any size, the bug will likely surface if the number of elements is capped at a relatively small value. In [55], the user provides bounds for the number of heap cells and loop iterations to ensure the problem is decidable.

The abstraction process is often manually controlled by the programmer. For instance, the programmer might have to specify the set of variables they are interested in. Research efforts have worked on automating this process [6, 66].

Many abstraction techniques retain soundness at the expense of completeness. Paths through the program are combined during the abstraction process. False alarms are introduced because the reported error occurs on a path that is not feasible. In these situations, it is desirable to implement refinement techniques that split the combined paths to remove the false alarms. This is commonly done by first checking if an error is on a feasible path.

If it is not on a feasible path, constraints are added and the process is repeated starting with the abstraction process. BLAST [48] uses lazy abstraction, an automated abstraction and refinement process that abstracts the program to the proper amount of precision necessary to verify a particular property. Statistical methods [60] can also be used to filter out false alarms based on how likely a reported failure is actually a bug.

The issue of soundness and completeness is a trade-off in static bug detection tools. For complex properties, retaining soundness with a computationally feasible algorithm will often result in an unbearable amount of false alarms due to the degree of abstraction necessary. The other end of the spectrum includes approaches that give up soundness but every error that is reported is guaranteed to be an error. Instead of combining paths, such approaches limit the number of paths they traverse with the hope that they traverse enough interesting paths to find all of the bugs. Obviously, these systems cannot verify that a property has been satisfied but are still very useful in finding bugs.

Static techniques struggle with analyzing loops and data in the heap. Abstraction techniques can help with loops by combining paths of different iterations into one path. However, it is possible that false alarms demand the paths be split. Heap data can be analyzed using different points-to analysis techniques [2, 42, 96] but all of these approaches in effect combine different variables into a single element; this leads to either unsoundness or incompleteness or both.

2.1.1 Software Model Checking

In software model checking, a model of the program is traversed to determine if a property can be violated during a path of execution. The state of the program is tracked during the

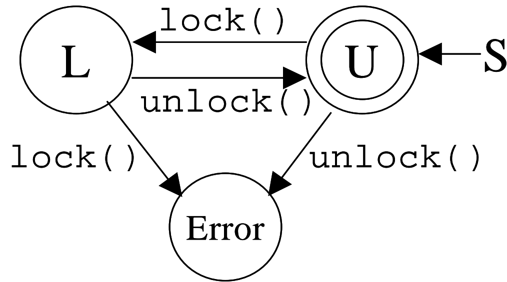

Figure 2.1: Finite state automaton model for correct locking behavior. This will detect errors where a lock is enabled twice or disabled twice in succession. It is also an error if the program ends with locks enabled.

traversal. The state that is stored varies depending on the property and level of abstraction used. At one end of the spectrum, a system may fully mimic the state of all variables, program stack, and heap. More commonly, bug detection tools abstract away most of the machine state and only track state that is relevant to the property at hand.

Model checking lends itself well to correctness properties that are specified as finite state automata (FSA). Figure 2.1 shows an example of a model for checking proper locking behavior. The starting state for each lock is U (unlocked). When a lock event occurs, the state machine transitions into the L (locked) state. An error occurs if there are two consecutive lock or two consecutive unlock commands. The U state is also an accepting state, if the program ends and the state machine is not in an accepting state, it is considered a failure. In this example, it is failure if the program ends while holding a lock.

The state transitions in this example are triggered by unlock and lock function calls. Other properties will use different events to cause transitions. For instance, a pointer dereference (such as `*a`) will be an event in a memory access checker or an array access (such as `a[i]`) will be an event in an array bounds checker. The state of each lock must be included in the tracked machine state during the traversal.

Since a static traversal of the program model is performed, software model checking can be thought of as a search problem. The goal is to search all possible program states to see if one exists that violates the property. Even with abstraction, there may be too many states to analyze depending on how much state the property requires. To combat this problem, restrictions are placed on the scope of the program model traversal. For example, properties could be checked within each function of the program instead of the entire program. Restricting traversals to functions is useful since most functions are small enough to search. The downside of this is that it will no longer be sound, missing bugs that manifest across function boundaries.

Microsoft's SLAM system [6, 9] is a good representative system of the model checking process. Correctness specifications are separate from the code and are written in the SLIC specification language [10]. A set of predicates is extracted from the specification and is used along with the program (written in C) as an input to C2BP [5] to create a boolean program. The boolean program is analyzed using BEPOP [7] which performs reachability analysis to determine if an error state can be reached. If an error is detected, their refinement tool NEWTON [8] checks to see if the path is feasible and adds predicates to remove false alarms. While their reachability analysis can take exponential time, experimental results show the SLAM process terminates within reasonable times for device drivers.

PREfix [16] uses a bottom-up approach for model checking. The call graph for the program is created and leaf functions are processed first. Besides finding bugs within the function, a model is constructed. The model consists of a set of outcomes. Each outcome has a set of constraints, guards, and results. Constraints are preconditions that must be true

or an error occurs. A guard is a conditional operator that must be true in order for the particular outcome to be valid. If the guard evaluates to false, it is not an error; it simply means the outcome is not valid (infeasible path). A result is the postcondition of the function and may involve more than one variable. The model of the function is used when it is called by other functions. Models may be created by hand for functions where source code is not available (such as system libraries).

The SPIN model checker [50] is designed for verifying distributed system protocols. A model of the protocol is written in PROMELA [51] and the correctness properties are written in linear temporal logic [79]. FSAs are created for the model and the correctness properties and subsequently checked. The PROMELA models are required to be bounded (forcing it to be decidable) and SPIN optimizes the checking process by reducing the number of states using partial order reduction and state compression. A more in-depth treatment of the theory used in SPIN is in [49]. The SPIN infrastructure is not limited to software. It has been used to verify concurrent systems and protocols in hardware, circuits, and other engineering disciplines.

Other model checking systems include Verisoft [43], CMC [71], and SMV [15, 67], a symbolic model checker that uses binary decision diagrams (BDDs). An example of using SMV is in [19]. MOPS [21] models the program using a pushdown automaton and the property as an FSA. Analysis is scalable and interprocedural for detecting ordering constraints. Though the system is unproven for more complex properties, it does show that simple ordering properties can be completely verified statically. Jackson and Vaziri [55] translate constraints, written in Alloy [54], into a boolean formula that can be solved by a

SAT solver.

2.1.2 Static Analysis

Many bug detection tools borrow or extend program analysis techniques that are done during compilation. For example, flow-sensitive approaches similar to data-flow analysis in a compiler can be used to find bugs. These techniques are fast and guaranteed to terminate since they iterate to a fixed-point solution. This often comes at a price of both unsoundness and incompleteness. A trade-off of these approaches and a comparison to model checking is discussed in [70].

Engler's research [32] has focused on finding bugs in operating system kernels. Specifications are separate from the program and written using their Metal language. The language allows users to specify events and the corresponding state transitions. The checker (which is part of their compiler) traverses each function looking for errors. While they have rudimentary support for finding errors across functions, their analysis is predominantly local. They have found numerous bugs in Linux and OpenBSD. In [3], Ashcraft and Engler constructed models to catch inappropriate uses of tainted data. In their model, tainted data becomes untainted if any check occurs. They only validate that checks are executed. Their approach is unable to determine if the checks are correct. This work was expanded by Xie, Choe, and Engler [93] who created ARCHER, a path-sensitive tool that uses symbolic analysis to find memory access errors.

Coen-Porisini *et. al.* [17] use symbolic execution for a subset of the C programming language and have applied their technique to safety-critical software systems. The FLAVERS system [18] creates a trace flow graph which is a reduced representation of an annotated

control flow graph. The reduction is an abstraction technique, removing control flow graph nodes that do not affect the verification of the property.

Using constraint-based analysis also is popular. Wagner *et. al.* [89] parse the program to create a set of constraints. The constraints are analyzed to find potential buffer overflows. Variables and buffers (any array or string) are represented as ranges and the problem is transformed into a system of integer range constraints. CSSV [27] statically verifies if string operations are safe by keeping track of the size of the string and whether or not the string is null but requires annotation for each function. It uses constraint variables to track relationships between variables. Potential violations are discovered using integer analysis on a set of linear inequalities. Rugina and Rinard [81] use constraint analysis to find bounds violations for pointers, arrays, and memory. They give an excellent overview of the process of converting symbolic expressions into linear constraints.

2.1.3 Type Systems and Safe Programming Languages

Significant research is devoted to extending or developing new type systems. Most languages perform some level of type-checking automatically. Types are useful in that they can be checked quickly in most cases and provide useful documentation to the programmer. The downside of types is that they are limited in what they are capable of checking. In some cases, type systems are more aggressive but rely on run-time checks if they are unable to completely verify a property statically.

Some examples in this area include the Vault project [25], a type system to enforce properties of protocols for the Windows kernel and device drivers. Shankar *et. al.* [83] introduce a tainted type qualifier to detect vulnerabilities in format strings at compile-time. Data that

come from untrusted sources will have a tainted qualified type. When a tainted typed variable is used in potentially dangerous situations, an error is signalled. Boyapati *et. al.* [14] describe new ownership types for Java that ensure proper memory management and real-time system guarantees.

Another way of reducing bugs is to use safe languages that automatically guard against malicious behavior. Some properties may not be verifiable at compile-time, causing dynamic checks to be inserted when static analysis fails.

Cyclone [56] is a modified form of C that checks pointer accesses using fat pointers. The programmer can limit the number of checks by declaring pointers to be safe. To ensure safety, additional restrictions (for example, disallowing arithmetic) are placed on safe pointers. CCured [73] uses static program analysis to prove as many pointers to be memory safe as possible. Dynamic checks are inserted when the analysis is unable to prove if the pointer is safe or not.

New software development paradigms, if used effectively, can reduce the number of bugs. Aspect-oriented programming [31] allows programmers to specify properties (such as security and buffering) and the relationships between them. Mechanisms in the aspect-oriented environment take the specification to create a program. This moves the burden of correctness from the programmer to the environment. The programmer still has to specify the relationships and interfaces correctly, a common source of bugs.

Similarly, annotations can be added to a program to aid the verification process by describing program specific properties that need to be checked, providing tips to the

abstraction process, and to give valuable information that may not be gleaned from analyzing the source code. For example, Splint [35, 62] is an annotation-based system derived from Lint that is used to catch security bugs. They use lightweight constraints based on annotations provided by the programmer. No interprocedural analysis is performed unless constraints are present. Aspect [53] is another system that requires users to indicate the state before and after a procedure. Fast static analysis allows common types of bugs to be detected.

2.1.4 Theorem Provers

Theorem provers also can be used to verify software properties [74]. Theorem provers take a program and verification condition as input and attempt to satisfy the condition. The verification condition is stated in a way such that satisfying the condition indicates an error. The process is similar to model checking in that the program is parsed, abstracted, simplified, translated, and then checked. The main difference is that verification conditions are created instead of a model. The correctness properties are embedded into the verification conditions.

Extended Static Checking (ESC) [26] is a verification system that uses theorem proving. It verifies properties for code written in Modula-3. It requires programmers to annotate procedures with a list of preconditions, postconditions, and modified variables. The analysis is unsound but ESC has been used to find bugs such as array bounds errors, null dereferences, and program invariants. ESC for Java has been recently developed [36, 85]. Other work in this area includes Destiny [29] which parallelizes the theorem-proving process. Flanagan and Saxe [38] developed a technique for compacting the verification conditions.

Eau Claire [22] is an extension to ESC designed to find security flaws.

2.2 Dynamic Bug Detection

Dynamic bug detection tools are inherently unsound since they cannot verify that a program violates a particular property as sections of the program may not execute. In order to prove a program is free of bugs, the validator would have to execute all possible input combinations - an infeasible task for all but the most trivial programs. While coverage tools can be used to make sure all interesting paths are hit, many bugs are only caused by a precise data value within a particular path (off-by-one errors for instance). Merely executing a path does not necessarily imply the particular path is free of bugs.

Related to this problem is the manifestation of errors. One approach to testing compares the output of a program to an expected output. Not all problems result in changes to the program output. For example, a program may overwrite memory beyond the bounds of the array - a bug. But depending on how the program is laid out in memory, an error may or not manifest in the output depending on what value was overwritten. Dynamic bug detection tools try to mitigate this problem by detecting the error immediately when the property has been violated (when the memory was overwritten)[1].

Despite these shortcomings, dynamic approaches have advantages when compared to static bug detection systems. In dynamic bug detection, execution always is on a real path. This reduces the number of false alarms that are detected. Static schemes often use abstraction techniques that combine paths; the number of false alarms increases due to

1. The point where the property is first violated is not necessarily the same as the source of the error but is much closer to the error than the "end of the program".

reported errors on infeasible paths. In addition, static schemes may restrict the scope of their search in order to make their algorithm computationally feasible. For instance, static analysis may be restricted within a function with limited support for detecting bugs across function boundaries. Dynamic bug detection systems do not suffer from such restrictions as they execute on a single path during a run of the program.

Another advantage of dynamic bug detection systems is their ability to fully utilize the run-time data that is available, allowing for direct analysis of constructs that are difficult to analyze statically. This includes loops where dynamic bug detection tools only need to be concerned with the current number of iterations, not all possible iteration counts. In addition, the precise shape and sizes of data structures on the heap are available at every point in the program. To remain computationally feasible, static analysis must represent the heap using imprecise models and use abstraction techniques to model loops.

Dynamic bug detection tools can be used to prevent certain malicious behaviors before they occur. As an example, such tools can be used to prevent an unsafe array reference by inserting a check prior to the statement. The downside to these approaches is that they must be very fast and non-obtrusive since they are still present after the program is deployed. This requirement, in essence, places a limit on the degree of what can be checked. Other dynamic bug detection tools are designed to be used during testing and are more heavyweight. These tools allows testers to find more bugs and to supply more debugging information to isolate the error.

Most run-time bug detection systems target a small number of properties and in many cases, the notion of correct behavior is built into the tool itself. This is helpful since the

tool can be optimized specifically to handle this particular case with the least amount of impact to the performance of the program.

Dynamic bug detection schemes often need additional state to track properties of the program. For instance, a table that keeps track of which locations of memory are valid is useful in memory access checking. We refer to this state as *shadow state* and it is commonly stored in a table. The table is typically indexed by address, storing the shadow state associated with the variable located at that particular memory location. The problem with this approach is the cost of looking up the state in the table. Another way of storing shadow state, often called *fat variables*, is to embed the state into the variable itself. This eliminates the table lookup but requires an increase in the size of the variable causing an increased memory footprint and compatibility problems.

Memory access errors are a popular target for run-time checking. They account for a large number of errors, especially in languages like C where pointers are subject to very few restrictions. Examples of memory access errors include using an uninitialized pointer, dereferencing a pointer that points to invalid data, freeing dynamically allocated memory twice, and accessing memory outside the bounds of a dynamically allocated object.

Examples of dynamic bug detection systems include GNU's checker [20], Purify [45], and Valgrind [88]. These tools detect memory bugs by keeping track of the state of dynamically allocated memory using a shadow state table to keep track of the state of memory. The table is implemented using a bitmap array making accesses fast. However, the limited amount of information gained from a small number of bits restricts the checker in what types of errors that can be checked. Lhee and Chapin [65] intercept array references and

checks to see if they exceed the bounds. Sizes of heap allocated arrays are stored in a table. Electric Fence [77] places inaccessible pages before and after each dynamically allocated object. A segmentation fault occurs if an access occurs outside the object. The taint mode of Perl [78] can be used to prevent untrusted programs from gaining superuser access. StackGuard [24] is a run-time approach that adds a randomized canary word just below the return address. If the canary word is modified, an error occurs. Buffer overflow attacks that overwrite the return address will also overwrite the canary word negating a jump to the attacker's code.

Haugh and Bishop [46] check all of the interesting string library functions by comparing the allocated sizes of the arrays. This approach is similar to our string library maximum size checks. Their tool tracks coverage to ensure that each interesting string function is executed once. Our technique can potentially find more defects because it checks for proper null termination and array references. Safe C [4] stores extra state with each pointer that stores the bounds of the object the pointer is referring to. Accesses are compared to the bounds to see if an error occurs. Safe C also catches illegal temporal accesses by assigning a unique capability to each created object. Pointers store the capability of the object they are pointing to. A table keeps track of the capabilities that are currently valid. An error occurs if a pointer dereference occurs and the capability is no longer valid. Keen *et. al.* [59] implemented the table in hardware to improve the run-time performance of the system.

Jones and Kelly [57] use an object table to store state on which region of memory a pointer is allowed to point to. Parasoft's Insure++ [76] checks for a variety of different

errors such as memory reference errors, memory leaks, unsafe I/O operations and data conversion errors. The checking is accomplished by adding checking and testing instrumentation around each line of source code. CodeCenter [58] interprets C code and provides run-time type checking and memory access checking. Wasserman and Blum [91] discuss run-time results checking where results of algorithms are checked using a simple checker. This only works for algorithms where a simple check (less complex than the algorithm) is possible. Netzer and Miller [75] describe an approach for dynamically finding data races in shared-memory parallel programs.

Run-time checking can be used to prevent programs from untrusted sources to execute malicious code. Wahbe *et. al.* [90] introduce address sandboxing. An untrusted code module (binary) is instrumented to restrict accesses to that module's segment(s). The technique is programming language independent and incurs a minor overhead. Related to the issue of security is trust. How does one trust code from outside sources? Necula and Lee [72] describe proof-carrying code. A proof is embedded into the binary that shows the code satisfies a particular property. An untrusted binary can be verified to see if the proof is valid. This check only needs to be done once. The execution of the program incurs only a negligible overhead penalty.

2.3 Testing and Debugging

Testing is closely related to software verification as they both strive to eliminate bugs from software. Several of the systems we have looked at, such as run-time memory access checkers, are designed to be used during testing to find more bugs faster. Testing is inherently an unsound but complete process. Since it not possible to run all possible input com-

binations for large programs, testing teams will employ path coverage techniques. The goal is to create tests that will hit all of the interesting paths of the program. Random testing can also be effective - fuzz testing [39] has found several errors by using random input streams.

Often testing is directed at particular regions of code. Good testing targets include regions where past bugs have been found or areas considered critical to the behavior of the overall system. Profiling can be used to critical regions by revealing areas that are executed often or are extremely complex. Research groups have looked at methods for automatically generating tests based on the program and a specification. Many of these approaches borrow ideas from bug detection analysis techniques. For instance, Korat [13] creates a set of test cases based on a Java method's precondition. The postcondition of the method is used to test the correctness of the method. Freidman *et. al.* [41] analyze finite state machine models of the program to create a set of coverage conditions that can be used to automatically generate tests. Testing constraints are also created as part of the analysis to address the problem of having too many cases to cover.

When an error is found, figuring out the cause of the bug can be very puzzling. As a result, significant support to aid debugging has gone into many of these systems. Often the user is provided with the exact line number of the error. Even better, some systems give the user an execution path that results in the error. There has been work done by Andreas Zeller [94, 95] to automate the debugging process. His Delta Debugging system narrows in on a bug by finding the minimum difference between a successful run and a failing run. The process borrows many ideas from abstraction: irrelevant state is removed during the nar-

rowing process. Dynamic Probes [69] is a tool that creates dynamic traces that are used to debug operating system code. Probes are added like debugger breakpoints but probes are designed to operate while the program is running with minimal overhead.

A closely related area of study is the elimination of array bounds checks [11, 44]. The approach works by propagating the constraints implied by array bounds checks through the dataflow graph of a program. Similar to our dynamic constraint modifications, program operators and control points adjust propagated constraints accordingly. They also eliminate array bounds checks by using static analysis to determine if definitions of earlier (sufficient) checks can reach the bounds check in question. Bodik *et. al.* [11] propose a lightweight analysis, making it possible to eliminate checks within a dynamic compilation framework.

Many different projects have focused on instrumenting code for a variety of applications such as profiling, code coverage, debugging, and of course, bug detection. Most of the dynamic bug detection systems instrument code in some fashion. The MaC project [82] adds instrumentation for monitoring and checking. They use two definition languages to describe their properties: a low-level language that matches generic events to specific program constructs and a meta language used to describe safety properties. The advantage of this decomposition is that it allows the safety property to be independent of any programming language. Tikir and Hollingsworth [86] analyze dominator trees to reduce the number of instrumentation sites needed for code coverage. Daikon [34] is a tool that dynamically finds program invariants which can be helpful in assuring correct program behavior. Tools used for gaining profiling information and other microarchitectural

parameters are typically instrumented at the binary level. ATOM [84] and EEL [64] are two examples of such systems.

CHAPTER 3

DYNAMIC DETECTION OF INPUT-RELATED SOFTWARE FAULTS

In this chapter, we describe our dynamic high coverage approach to detecting security faults caused by improperly bounded inputs. Our technique possesses the scope and program knowledge of a run-time technique while relaxing the requirement that the user specify a set of inputs that exposes the defect. We implement our approach by shadowing all input values (and variables derived from input) with a state variables.

Array references can be checked by tracking integers with an interval constraint variable that stores the lower and upper bounds of the range of values that the given variable may hold. When a variable that is derived from input is used as an index, the entire range of input values are checked to ensure the access does not exceed the bounds of the array. More information can be found in Section 3.1.

Input strings are shadowed by state variables that hold the maximum possible size of the string and a flag that indicates if the string is known to contain a null character. String functions are checked to ensure that all strings passed as parameters are null terminated and there is sufficient room in the destination string for copy operations. Our approach, described in Section 3.2, will verify these functions for all possible string lengths up to the

maximum size making it unnecessary for a test to have the exact string length that can trigger an error.

3.1 Tracking Integer Input to Detect Array Overflows

In order for an array reference to be considered safe, the index must be checked to determine if it can exceed the bounds of the array. This is accomplished by attaching interval constraint information to every variable that contains data derived from program input. We start with a model of *tainted* data that is similar to that used by Ashcraft and Engler [3] and is summarized in Figure 3.1. Data that comes from input is considered tainted and includes environment variables, command line inputs, data read from files, and network packets. Tainted data is shadowed with interval constraint variables that track the lower and upper bounds for the variable. When an access to an array occurs, the bounds of the array index are compared with the run-time size of the referenced array. An error is declared if there is an index that can exceed the bounds of the array. When a variable is assigned a value that is not dependent on input (untainted), the destination variable is reset to the untainted state, releasing shadow state[1].

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

1. Our approach only detects array overflows when indices are derived from input. However, checks for indices not derived from input can be added with minimal modifications.

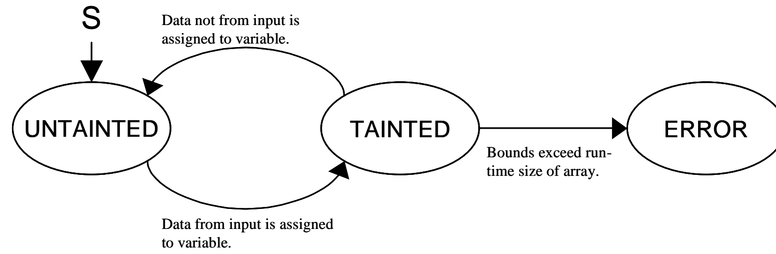

Figure 3.1: Program states for integer variables.

As tainted variables are operated upon or tested with control predicates, their interval constraints must be adjusted accordingly. Table 3.1 shows a list of representative operations and their effect on the upper and lower bounds of an interval constraint. In the table, ticked variables `a', x'`, and `y'` refer to tainted variables while `y` represents an untainted variable. The notation `x'.lb` represents the lower bound of tainted variable `x'`. The expressions `MIN_VAL(a)` and `MAX_VAL(a)` refer to the minimum and maximum values that `a` can have based on its type.

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

Addition and other arithmetic operations adjust the bounds of the destination variable. In the first addition pattern (Rule 2), a tainted value is added to an untainted value. The bounds of the destination variable are computed by adding the run-time value of the untainted variable to the bounds of the tainted variable. If both variables are tainted (Rule

Table 3.1: Representative rules for computing bounds on integer variables. x', y', and a' are tainted and y is untainted.

Rule	Operation	Input Interval Constraint
1	a' = x'	a'.lb = max(MIN_VAL(a'), x'.lb) a'.ub = min(MAX_VAL(a'), x'.ub)
2	a' = x' + y	a'.lb = max(MIN_VAL(a'), x'.lb + y) a'.ub = min(MAX_VAL(a'), x'.ub + y)
3	a' = x' + y'	a'.lb = max(MIN_VAL(a'), x'.lb + y'.lb) a'.ub = min(MAX_VAL(a'), x'.ub + y'.ub)
4	a' = x' % y'	a'.lb = 0, a'.ub = max(abs(y'.lb), abs(y'.ub))
5	if (x' < y)	if (x' < y): x'.lb = x'.lb, x'.ub = min(x'.ub, y - 1) else: x'.lb = max(x'.lb, y), x'.ub = x'.ub
6	if (x' < y')	if (x' < y'): x'.lb = x'.lb, x'.ub = min(x'.ub, y'.ub - 1) y'.lb = max(y'.lb, x'.lb + 1), y'.ub = y'.ub else: x'.lb = max(x'.lb, y'.lb), x'.ub = x'.ub y'.lb = y'.lb, y'.ub = min(y'.ub, x'.ub)
7	if (x' == y)	if (x' == y): x'.lb = y, x'.ub = y else if (x'.lb == y): x'.lb = y + 1, x'.ub = x'.ub else if (x'.ub == y): x'.lb = x'.lb, x'.ub = y - 1 else: x'.lb = x'.lb, x'.ub = x'.ub
8	if (x' == y')	if (x' == y'): x'.lb = y'.lb = max(x'.lb, y'.lb) x'.ub = y'.ub = min(x'.ub, x'.lb) else: x'.lb = x'.lb, x'.ub = x'.ub y'.lb = y'.lb, y'.ub = y'.ub
9	while (x' < y)	in loop: x'.lb = x'.lb, x.ub = min(x'.ub, y - 1) after loop: x'.lb = max(x'.lb, y), x.ub = x'.ub

3), the bounds are added together to form the new worst-case bounds. Rule 4 singles out the modulus operator because the new range is strictly dependent on the value of the second operand. We can also detect overflow situations; this is mentioned in more detail in Section 3.3.

Rules 5-9 narrow the input interval constraint based on knowledge gained from control predicates. In Rule 5, if the if condition is true, the upper bound is reduced to y-1 unless the existing upper bound is already lower than y-1. If the condition is false, the lower bound must be at least y. Rule 6 refers to a situation where two tainted variables are compared to one another. If x' < y' is true, then no change is necessary to the lower bound of x' and the upper bound of y'. The upper bound of x' must be at least one less than the

upper bound of `y'` in order to make the equality true. `x'` also cannot exceed its own upper bound. Similarly, the lower bound of `y'` is the maximum of the lower bound of `x'+1` and the lower bound of `y'` before the statement.

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

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

Notable omissions from the list are the logical-or (`||`) and logical-and (`&&`) operators. The simplification phase (discussed in Section 4.1 and Appendix A) converts these short-cir-

---

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

cuited operators into the appropriate if-then-else constructs.

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

3.1.1 Array Reference Example

To illustrate our approach, we describe a bug that was discovered in OpenSSH. It occurred in the channel code (*channel.c*). The relevant code is shown in Figure 3.2. In function `channel_new`, the channels array is a dynamically growing array with a size equal to `channels_alloc`. The starting size of the array is ten.

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

At line 46, there is a check to make sure that index is within the bounds of the array. If the run-time value of `id` is out of the range $0 \leq \texttt{id} \leq \texttt{channels_alloc}$, the error would not be detected since the function call returns before the array access. At the array access in

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

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

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

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

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

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

Figure 3.2: OpenSSH channel bug. The `channels` array has a size of `channels_alloc`. The bug occurs in `channel_lookup` where it is possible to access `channels[channels_alloc]` which is outside the bounds of the array.

line 50, the interval constraint of id has a lower bound of zero and an upper bound equal to the run-time value of channels_alloc. The channels array has a run-time size equal to channels_alloc, indexed from zero to channels_alloc-1. Since the upper bound of id is channels_alloc, it exceeds the bounds of the array and an error is declared. If id is in the range 0 ≤ id ≤ channels_alloc, the error will be detected despite the fact that an error only occurs when id equal to channels_alloc. For any value of id that executes the array access, our technique will detect the error. To fix this bug, line 46 must be changed to use '>='.

3.2 Detecting Misuse of String Functions

String functions such as strcpy can lead to software faults since no check is performed to determine if the source string will fit in the destination buffer. While it may be possible to detect string-related errors using the array reference checker presented in the previous section, it has some shortcomings when attempting to validate the higher-level string properties. In particular, the array reference checker does not analyze the values of the array and hence has difficulty detecting problems associated with null termination. In addition, the checker uses the run-time size of the array for the check. When strings come from input, the size of the string should be treated similarly to an integer that comes from input. As a result, strings that come from user input are tracked separately from integers. Since a string provided by a user can have arbitrary length, our technique assumes that all input strings can have infinite length initially. Comparisons made to the length of the string adjust the maximum length of the string. When a string copying function is called, the maximum length of the string is checked to ensure it will fit in the destination.

Another common problem is when strings are not terminated with a null character. While input strings automatically contain a null character when they are first created, they could be copied using functions such as `strncpy` which do not copy the null character in all cases. Another common mistake is to forget that the `strlen` command does not include the null character in its count, leading to an off-by-one error if used incorrectly. Since we consider input strings to have an arbitrary length, it is not necessary for a user to supply a string of the precise length in order to find such errors.

All strings and arrays in the program are tracked with the three fields: `actual_size`, `max_str_size`, and `known_null`. The field `actual_size` stores the actual run-time size of the array and cannot change (except for calls to `realloc`). The field `max_str_size` stores the maximum size of the string in the array. It refers to the largest possible size of a string that a user can supply. For example, strings that come from the command line have an initial `max_str_size` of infinity (`INT_MAX`) and strings that are created using `fgets` have a `max_str_size` equal to the supplied limit. For arrays that do not contain strings, `max_str_size` is equal to `actual_size`. The `known_null` field is a flag that is true if the string is known to contain a null character. If it is false, it is not known if a null character is present or not present. During checking, we assume that the string is not null terminated if `known_null` is false. We will represent accesses to these fields using structure notation: `s.max_str_size` refers to the field `max_str_size` associated with the array `s`.

Strings can be created in a variety of ways as shown in the first five rules of Table 3.2. Strings that come from the command line or environment variables (Rule 1 in Table 3.2)

Table 3.2: Representative Rules for string buffer overflow checking. `s` is an array, `p` is a pointer, `n` and `c` are integers, and `m''` and `n''` are integers that store a string length.

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

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

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

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

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

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

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

the size of the allocation is dependent on the size of another string. This case (Rule 5) is described in the next paragraph. Similar rules exist for `calloc`, except that `known_null` is initialized to true.

In order to properly adjust the maximum size of a string, it is necessary to track integers that store string lengths. Any integer that is storing a string length will have state that stores the starting address of the corresponding string (denoted using the field `string`). In addition, a `size_diff` field is also stored that is the difference between the value stored in the integer and the actual length of the string. This is important as our approach includes the null character in the length of a string while the `strlen` command does not. Therefore, the initial size difference for a `strlen` result is -1. In Table 3.2, variables that store string lengths are represented with two tick marks (such as `n''`). Rule 6 shows the `strlen` call. Since `strlen` requires the input string to be null terminated, a check is made to make sure that is the case. Addition and subtraction operations (Rule 7) on the string length adjust the size difference appropriately. For example, adding one to a `strlen` result to account for the null character will result in a size difference of zero. The maximum length of the string can be reduced with a control operation; this is illustrated in Rule 8. If (`n'' <= c`) is true, the maximum size of the string s is adjusted to `c + n''.size_diff` unless the maximum size is already smaller. If (`n'' <= c`) is false, no adjustment is made to `s` since there is no restriction on the maximum size of `s`.

Refer back to Rule 5 where a string length is used as the size parameter to a dynamically allocated array. This is commonly done before a string copy to ensure the destination has enough space to hold the source string. As a result, the field `max_str_size` of the newly

allocated region is initialized to properly reflect the maximum size of `n''.string` rather than using `n''`. Our current implementation limits the usage of string length integers. We do not handle arithmetic operations except addition and subtraction and we do not handle operations that involve two string length integers. For more information on these restrictions and the limitations they cause, see Section 3.4.

Assigning zero to an element of an array will set the `known_null` flag to true (Rule 9). The `known_null` flag is also set to true when functions `bzero` and `memset` (to zero) are called.

String arrays are often accessed via pointers. This case is handled by shadowing the pointer with the base address of the array (denoted using `array_base`). Pointers that do not point to arrays are not shadowed. In the event that the string is addressed using an interior pointer, the state of the array is obtained using the shadowed base address. This is illustrated in Rule 10. For brevity, we will omit this level of indirection and assume arrays are used in all future rules.

Table 3.3 shows how string library functions are handled. Rules 11 and 12 illustrate how string copies are handled. In `strcpy`, the source must be null terminated and the size of the maximum string size must fit in the destination. The size of the destination is determined by taking the maximum of `actual_size` and `max_str_size`. In the case where `max_str_size` is larger than `actual_size`, `max_str_size` is chosen since the only way this situation can occur is when the array was dynamically allocated with a string length. Our approach naively assumes that the string length used referred to the length of the source string. This assumption could lead to undetected bugs and is discussed in more

Table 3.3: String function rules for string buffer overflow checking. `s`, `d`, `set`, `fmt` are strings, and `p` is a pointer to a string. `n` is an integer and refers to a parameter that restricts the number of a characters written into a destination buffer. The macro `SIZE(s)` is equal to `MAX(s.actual_size, s.max_str_size).`

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

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

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

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

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

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

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

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

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

detail in Section 3.4. For brevity, we use `SIZE(s) = MAX(s.actual_size, s.max_str_size)` to represent the size of the destination buffers. If both the null check and size check pass, the destination will then have `known_null` set to true and a `max_str_size` equal to that of the source. In `strncpy`, a check ensures that the destination size is less than the supplied size (`n`) parameter. This is done regardless of the size of the source string since nulls are padded at the end if the source is smaller. The destination will have a `max_str_size` that is the smaller of `n` and `max_str_size` of the source. The `known_null` is only true if the entire source string is copied.

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

Strings that come from input functions such as `fgets` (Rule 18) will have a maximum size equal to the size that was supplied as the limit to the function and are null terminated.

	Code	State
	`char buf0[12];`	buf0.max_str_size = 12, buf0.known_null = FALSE
	`char *buf1;`	
	`char buf2[18];`	buf2.max_str_size = 18, buf2.known_null = FALSE
	`char *p;`	
5		
	`strncpy(buf0, argv[1], 12);`	buf0.max_str_size = 12, buf0.known_null = FALSE
	`buf1 = strdup(argv[2]);`	buf1.max_str_size = ∞, buf1.known_null = TRUE
	`if (value) {`	
10	`p = buf0 + 1;`	p.array_base = buf0
	`strcpy(buf2, p);`	p.array_base is buf0 → buf0.known_null == FALSE → ERROR
	`}`	
	`else if (strlen(buf1) <= 6){`	buf1.max_str_size = 7, buf1.known_null = TRUE
	`buf0[12] = 0;`	buf0.max_str_size = 12, buf0.known_null = TRUE
15	`sprintf(buf2, "%s%s", buf0, buf1);`	(buf0.max_str_size + buf1.max_str_size = 19) > (buf2.max_str_size = 18)
16	`}`	→ ERROR

Figure 3.3: Example of detecting string bugs. The `strcpy` in line 11 can fail because `buf0` is not null terminated. The `sprintf` in line 15 can fail because the sizes of the two source strings could exceed the size of the destination.

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

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

limitations prohibit more sophisticated analysis. This remains as future work.

3.2.1 String Example

A detailed example illustrating how string errors can be found is shown in Figure 3.3. The two buffers `buf0` and `buf2` have an initial maximum size equal to their static sizes. In line 6, the input value `argv[1]` is copied into `buf0` using `strncpy`. While the specified size of 12 does not cause an overflow, the null flag for `buf0` remains off since a null would not have been copied if `argv[1]` has at least 12 characters. The `strdup` in line 7 will duplicate the state values of `argv[2]` into `buf1`. If `value` is true, execution will continue to line 10 where the pointer `p` is assigned to point to the second element of `buf0`. This causes `p` to be shadowed with the base address of `buf0`. When `p` is used in the `strcpy`, an error gets properly signalled because `buf0` may not be null terminated. In the case where `value` is false, a comparison is made based on the length of `buf1`. Assuming it is less than or equal to 6, control will be taken to line 14 and the maximum size of `buf1` will be restricted to 7 (6 plus 1 for the null character that `strlen` does not count). The `known_null` flag is set for `buf0` in line 14. In line 15, an error results because the sum of the maximum sizes of the two source buffers (19) can exceed the size of the destination (18).

3.3 Other Improper Uses of Input Data

Our approach can also be used detect other situations where input data could be used dangerously and possibly lead to software faults. Unconstrained input used to control the number of loop iterations, the size of a memory copy, or the size of a memory allocation could be dangerous [3]. We check to make sure that the variables controlling these uses

have been constrained in some fashion. An error is signalled if the upper bound is equal to the maximum value allowed by the type of the variable. For these situations, it is important to note that these types of uses are not always errors. In some cases, input is constrained later within the loop and malicious behavior is properly thwarted. In other cases, the loop may not do anything that can be exploited. Memory allocations are likely not dangerous if the output is properly checked to ensure the allocation was successful.

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

While this type of problem doesn't necessarily lead to a security exploit, it is possible. For example, we describe another security bug found in OpenSSH. The code is listed in Figure 3.4. Data is received from a packet and then is subsequently used to allocate an array. At the time of the `malloc`, no restriction has been placed on the input. A malicious user could supply an extremely large value in order to cause an overflow on the multiplication in the `malloc` call resulting in a small allocation. Since the same input value controls the number of iterations within the loop that follows, the array accesses within the loop can be used to access memory outside of the array.

During our experiments, we did not activate arithmetic overflow checking except for one

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

Figure 3.4: OpenSSH challenge bug. Unbounded data from a packet can cause overflow when calling `malloc`.

experiment that is outlined at the end of Section 5.3. The reason is that arithmetic overflow checking caused too many false alarms and hindered run-time performance too much to be overly useful. In addition, many of the bugs that were found by overflow checking were also found by other checks. For instance, the OpenSSH bug that was described in the previous paragraph was detected because an unconstrained value was used in the call to `malloc`.

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

3.4 Limitations

Since our approach is dynamic and relies on the particular control path taken through a program, it is an *unsound* approach, meaning that it is possible to miss actual bugs. With respect to a particular control path, our approach also is unsound. One problem stems from the use of run-time data. An example is where the actual size of the array is used during array checks. On a different run with the same control path, the size of the array, if con-

```
    unsigned int size;
    unsigned int index;
    int *array;

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

    /* initialize array */
10
    index = getchar();
    if (index < 0 || index > 9) exit();
13  y = array[index];
```

Figure 3.5: Example of an unsound control path. This code segment will overflow the index is greater than the array size.

trolled by input, could be smaller and be subjected to an array buffer overflow. This is illustrated in Figure 3.5. The size of the array is controlled by user input and could be any value from one to ten. However, the filter of illegal accesses for the index in line 13 is valid when the array size is ten and invalid for all other accesses. As a result, an error will be missed if ten is supplied as the array size.

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

Our technique also is *incomplete* in that it can produce false alarms, signalled bugs that are not actual bugs. As alluded to earlier, these often occur due to a lack of symbolic analysis. Not all possible relationships between different strings or variables are tracked and this can cause operations that narrow bounds to be missed. A specific example of where sym-

bolic analysis is not present is the limited functionality associated with integers that store string lengths. When an unsupported operation occurs, a warning is emitted, and the result is not shadowed. Another problem arises in that our technique does not keep track of which position the null character is in. In order to maximize the number of detected bugs, we assume it is in the last position. This assumption also leads to an increase in false alarms.

There are also a few instances where we intentionally allow for incompleteness and unsoundness in order to improve performance. An example of this is detecting when a zero gets written into an array. In rule 9 in Table 3.2, the `known_null` flag will be set when the constant zero is written into an array element. For soundness and completeness, we should also have a similar rule when a variable is written into an array element. If the value of the variable is zero, then the `known_null` flag should also be set. If the value is not zero, it is possible that null was overwritten and `known_null` should be cleared. However, we do not include such a rule because it significantly increases performance overhead and is very rarely useful. When a variable is assigned to an array element, the value of the variable is typically not zero and the array usually is not a string. The constant case (rule 9) is commonly used to ensure that a null appears in the array. In practice, we found the number of false alarms to be manageable. We describe the false alarms triggered in Section 5.3.

3.5 Chapter Summary

We presented our technique for detecting input-related bounds errors. The concept behind our model is to track variables using shadow state that holds input data. For integers, the

lower and upper bounds of the value are stored. For strings, the maximum length of the string is stored along with a flag that indicates if a null character is present in the string. Control operations narrow the bounds and arithmetic operations adjust the bounds. At dangerous operations, the entire allowable range of values is checked to ensure that the memory is not accessed out of bounds.

Since the allowable range of values is checked, it is not necessary to have precise input data that exposes the bug, a common deficiency of dynamic bug detection systems. This work eliminates the data dependence but there is still a dependence on control since the analysis still is tied to a single path of execution. Using intelligent path coverage can mitigate the control path dependence.

Besides checking array references and string library functions, we explored other operations where unconstrained input data can be problematic: controlling the number of loop iterations, controlling dynamic memory allocation and arithmetic overflow bugs.

The next chapter describes MUSE, our program instrumentation infrastructure. Chapter 5 describes the implementation of the input bounds checker within MUSE. In addition, Chapter 5 gives results about bugs detected and the performance impact of the instrumentation.

CHAPTER 4

MUSE: A GENERAL PURPOSE INSTRUMENTATION INFRA-STRUCTURE

MUSE, our dynamic instrumentation tool, takes as input an instrumentation specification that is hand-written and is external to the program. No modifications to the program source code or special run-time systems are needed. The specification and the program are input to the compiler which adds instrumentation to the code at appropriate points in the program.

MUSE is general-purpose and allows users to write specifications using our stylized specification language and C code. The purpose of MUSE is to provide a rapid prototyping environment for developing dynamic analysis tools. It will provide capabilities for techniques that require dynamic or static analysis of program execution.

Though it was designed for bug detection, MUSE can be used in a variety of ways. It can be used as a debugging aid. Instrumentation can be added to track variables, invariant expressions, or the sequence of function calls. The tool also can be used to create traces of selected events or instructions for use by simulators or analyzers. It also can be used as a dynamic analyzer or profiler. User have the ability to create their own statistical measures including those not found in off-the-shelf profilers.

This chapter describes how to use MUSE and its specification language. Section 4.1 presents an overview of the internal implementation of MUSE. The specification language is described in Section 4.2 and an example specification is illustrated in Section 4.3. This chapter concludes with some preliminary performance results and experiences gained when using a memory access checker created using MUSE.

4.1 MUSE Implementation

MUSE is implemented as a phase in the GCC C compiler and operates at the abstract syntax tree (AST) level. Our system requires source code to be present in order to add instrumentation. The functionality of functions without source code, system libraries in particular, can be characterized by adding instrumentation before or after the system call.

The first step of instrumentation simplifies the program. We convert the program into *Elemental C*, an intermediate representation very similar to that developed by Hendren [47]. We used code from a development branch from GCC [40] as a starting point for this phase. The purpose of this simplification process is to reduce the complexity of the pattern matching phase. Complex C statements are broken down into simple statements with at most two operands and an assignment (such as `a=b+c`). Side effects and short circuited operators are eliminated. An advantage of the simplification process is that instrumented function calls are restricted to statement boundaries. This is not possible for complex statements since there might be an important event in the middle of a long expression. The full *Elemental C* language is described in Appendix A. An example of a simplified program is given in Figure 4.6.

Once the program has been simplified, the instrumentation specification is processed. The

specification consists of two parts: a pattern file and a set of instrumentation functions. The pattern file is written in our specification language (described in Section 4.2) and consists of a set of pattern-function pairs. Patterns correspond to statements, expressions, or special events (such as the start of a function) within the *Elemental C* language. MUSE parses the pattern file and constructs a table of patterns. Then, MUSE traverses the program one statement at a time to see if there is a match. If there is match, the appropriate instrumentation code is added depending on the corresponding function (of the matching pattern-function pair) and the current mode of operation. There are two modes of operation for adding instrumentation: normal mode or inline mode. In either mode, the user must specify whether the instrumentation should be added before or after the given statement. It is possible to mix these two modes of operation in the same specification.

In normal mode, the function specified in the pattern refers to an external function and a call to this function will be inserted into the code. The external function, written by the user in C, is responsible for performing any actions the user desires. The specification language allows parameters to be passed into the instrumentation function using a set of predefined parameter macros. Example macros include addresses and values of variables, the current line number, and type information. External functions are compiled normally (without further instrumentation) and must be linked into the final executable.

In order to take full advantage of the capabilities of the compiler, it is also possible to inline instrumentation. When a pattern match occurs in inline mode, the corresponding instrumentation function refers to a function that will be called within the compiler. The function will be responsible for adding the appropriate instrumentation. This allows the

user to perform tasks like creating temporary variables, adding arbitrary statements, and further refining the match using compiler information. The obvious downside to using this mode is that it requires the user to be familiar with the internals of GCC which has a relatively steep learning curve. To mitigate this effect, MUSE contains a number of helper functions for common tasks.

Once the instrumentation has been added, the remaining compiler phases are executed creating an instrumented executable. Compiler optimizations do not need to be disabled in order to add the instrumentation code. The optimizations can recoup some of the performance penalty that occurs during the simplification process, especially in sections where there is little or no instrumentation.

We assume that specification writers are fairly experienced with the internals of the program they are trying to verify. Common events such as memory allocation may be implemented differently from program to program. This is not an unreasonable requirement as other software bug detection systems assume intimate knowledge of the program, especially those that require modifications to the source code.

4.2 Correctness Specification Language

This section outlines the specification language of the pattern file. The full grammar of the specification language can be found in Appendix B. Section 4.3 shows an example of a correctness specification. It may be helpful to refer to this section in understanding how to use the specification language.

The pattern file consists of lines in the following form:

```
<pattern>: before: <before_func> after: <after_func>;
```

- `<pattern>` is the pattern to be matched. It can refer to an event, a statement, or an expression. Patterns are described in Section 4.2.1.

- `<before_func>` and `<after_func>` are the instrumentation functions that are executed before and after a statement that matches a pattern. Only one of the functions `<before_func>` or `<after_func>` need to be specified and both can be specified if desired. In normal mode, a set of predefined parameters can be passed into instrumentation functions. The parameters are described in Section 4.2.5.

All keywords (except `before` and `after`) in the language are preceded with a dollar sign '$' to eliminate naming conflicts with the program that is being verified. C++ style comments are supported in the pattern file.

4.2.1 Basic Pattern Syntax

Patterns are used to match Elemental C statements. The basic syntax of a pattern is:

```
[$default] [$ignore_gbl] [$assign <lhs_indentifier>]
<main_identifer> [<qualifiers>]
```

The key part of the pattern is the main identifier which can refer to a statement, an expression, or an event. The list of identifiers was initially created by taking the names of the nodes that correspond to statements and expressions in GCC's AST. The list was expanded to include events that are important for bug detection and are not easily represented by an expression or statement. Wildcard identifiers can be used to represent a set of identifiers.

1. *ZeroOpStmtCode*
2. *OneOpStmtCode Operand*
3. *FuncExprCode ExpressionType FunctionIdentifier*
4. *FuncOpExprCode ExpressionType FunctionIdentifier Operand*
5. *ZeroOpExprCode ExpressionType*
6. *OneOpExprCode ExpressionType Operand*
7. *TwoOpExprCode ExpressionType Operand Operand*
8. *LhsOneOpExprCode ExpressionType Operand*
9. *LhsTwoOpExprCode ExpressionType Operand Operand*

Figure 4.1: Non-assignment patterns in the specification language. The underlined terms refer to identifiers. The non-underlined terms are qualifiers.

The pattern can be preceded by two optional keywords: `$default` (described in Section 4.2.4) and `$ignore_gbl` (described in Section 4.2.6). Patterns that represent an assignment use the `$assign` keyword and have an additional identifier that represents the left hand side of the assignment. The right hand side is represented using the main identifier. Assignment patterns are described in Section 4.2.2.

Figure 4.1 gives a list of valid non-assignment patterns. Identifiers are classified into nine groups depending on whether the identifier refers to a statement or an expression, the number of operands, and whether or not it can appear on the left hand side of an assignment. Consult Tables B.1 - B.9 in Appendix B for the list of operators that make up each group.

Qualifiers are used to restrict the pattern further. There are currently three types of qualifiers: *ExpressionType* (match against the resulting type of an expression), *FunctionIdentifier* (match against the name of the function), and *Operand* (match against the type and name of an operand). Matching against operands and types is described in more detail in Section 4.2.3.

Pattern 1 (in Figure 4.1) refers to statements that have no additional qualifiers. Instrumentation is added each time the particular type of statement occurs. Pattern 2 matches state-

ments that contain an argument and includes control statements such as `if` and `while` where the operand refers to the variable or constant that is tested against zero.

Pattern 3 corresponds to pattern identifiers that take a function name as an argument. The *FuncExprCode* group refer to events associated with function calls: A call to a function, the beginning of a function, and the end of a function. The *FunctionIdentifier* qualifier is used to match the function name; its use is detailed in Section 4.2.3. The *ExpressionType* qualifier is optional and can be used to restrict matches to functions that return the given type. The pattern "`$begin_fn main`" can be used to trigger the start of the program. This event is useful for initializing variables and data structures used in the instrumentation routines. Pattern 4 is used for parameter passing and returning values between functions. The two operands allow the user to specify both the function and to match against the particular parameter or return value. For function calls with multiple arguments, each parameter is matched independently. This means that separate instrumentation can be added for each parameter. Special instrumentation also is available for matching fields within structures that are function arguments or return values.

Patterns 5-7 refer to expressions with zero, one, or two operands respectively. All three patterns accept an optional *ExpressionType* qualifier and the necessary number of *Operand* qualifiers. Patterns 8-9 are identical except they can appear on the left hand side of an assignment pattern, described in the next section. Three wildcard identifiers are also available: `$any_unary_op` (single operand expressions), `$any_binary_op` (two operand expressions), and `$any_cast_op` (casting operations).

10. `$assign` LhsExpr *FuncExprCode* *ExpressionType FunctionIdentifier*
11. `$assign` LhsExpr *ZeroOpExprCode* *ExpressionType*
12. `$assign` LhsExpr *OneOpExprCode* *ExpressionType Operand*
13. `$assign` LhsExpr *TwoOpExprCode* *ExpressionType Operand Operand*
14. `$assign` LhsExpr `$simple_rhs` *Operand*
15. `$assign` LhsExpr `$integer_cst` <integer>

Figure 4.2: Assignment patterns in the specification language. The underlined terms refer to identifiers. The non-underlined terms are qualifiers. The term LhsExpr refers to a pattern that is used to match the left hand side of the assignment.

4.2.2 Assignment Patterns

A common expression in C is the assignment operation. In *Elemental C*, a simple statement will have at most a single destination and two source operands. Figure 4.2 lists the set of assignment patterns. The term 'LhsExpr' refers to a pattern that corresponds to the left hand side of the assignment and must be either in the form of pattern 8 or 9 in Figure 4.1. Patterns 10-13 extend the similar non-assignment pattern in the expected way. In order for instrumentation to be added, both the left hand side and right hand side must match the given statement.

Pattern 14 refers to assignments where the right hand side is a single variable or constant and pattern 15 is used in cases where the right hand side is a constant and the user wants to match against a particular value of the constant (zero, for instance, is a commom value to match against).

4.2.3 Matching Operands and Types

An important aspect of the specification language is the ability to match operands. An operand consists of a type and a variable. In order for the operand to match, the type and the variable must both match. Types are also used to match the types of expressions.

The type is specified in the following form:

```
TypeIdentifier TypeName(optional) PtrStar(optional)
```

The *TypeIdentifier* can either refer to a built-in type such as `$integer_type`, `$float_type`, or `$char_type` or a basic form of a more complex type such as `$record_type` (structs), `$union_type`, and `$enumerated_type`. If a complex type is used, it will match all instances of that type. To match against a specific complex type, an type name can be specified. For instance, to match against the type "`union foo`", use the operand pattern "`$union_type(foo)`". Pointers can be specified by adding a star or stars to the end of the type. As an example, the pattern "`$integer_type **`" will match pointers to integer pointers. There is also `$pointer_type` which matches all pointers and `$any_type` which matches all types. Consult Table B.10 in Appendix B for a complete list.

The variable portion of the operand is either the name of a variable or one of three wildcards. The name of the variable will match any variable with the particular name. There are three wildcards: `$any_var` (any variable - no constant), `$any_const` (any constant), or `$any_value` (any variable or constant).

When matching two operand expressions, the order of operands matter. For example, the pattern "`$plus_expr $integer_type a $integer_type b`" will match "`a+b`" but not "`b+a`". The reason for this is that several binary operators do not have the commutativity property. In cases where the order of operands does not matter, two patterns can be used, one for each ordering.

Matching functions, using the *FunctionIdentifier* qualifier, is similar to matching to vari-

able names. The only difference is that there is a different set of wildcards: `$any_fn` (any function including function pointers), `$any_fn_const` (any function excluding function pointers), and `$any_fn_ptr` (function pointers only). Note that many of the patterns that accept *FunctionIdentifier* qualifiers will never have a function pointer as an argument (`$begin_fn` is an example).

All of the expression patterns allow for the type of the expression to be specified (the *ExpressionType* qualifier). This is optional and is useful for situations where the expression type is not easily derived from the source operands such as function calls and casting operations. The type is specified in the same format as with the operand, the only difference is that the type is surrounded by parentheses. For example, the pattern "`$call_expr ($integer_type) $any_fn`" will match a call to any function that returns an integer.

A list of possible matches can be created using a wildcard such as `$any_var` and the `$except` keyword followed by a list of names that should be excluded. This can be done for variables, types, and function names. As an example, the pattern "`$end_fn $any_fn $except(main, foo)`" will match the end of any function except those named `main` or `foo`.

4.2.4 Default Patterns

Normally, instrumentation will be added for each pattern that is matched by a particular statement. To change this behavior, default patterns can be used. Default patterns are only analyzed if no other patterns have previously matched for the given statement and main identifier. Default patterns are designated using the "`$default`" keyword and are otherwise normal patterns. There can be multiple default patterns for a given identifier but at

most one can match a particular statement. An error is signalled if a statement matches two default patterns.

4.2.5 Passing Parameters to Instrumented Functions

The pattern matching is applied statically - no run-time information is used. However, the use of run-time data is often necessary in the instrumentation routines for additional matching, error checking, and maintaining state of the program. A set of pre-defined parameters is provided for this purpose. The list is outlined in Table 4.1. Some parameters require parameters themselves. Value parameters require a type. While this probably could be extracted automatically, it simplifies the implementation. The type must correspond to an actual type (no wildcards) and be identical to the type indicated in the instrumentation function. Function arguments[1] and operand parameters require an index number that is used to select the appropriate argument or operand. For operands, macros `$src0` and `$src1` refer to the source operands and `$dest` refers to the destination operand. When creating the instrumentation function, the parameters must appear in the same order with the proper type. They need not have the same name.

4.2.6 Global Variables

It is often desirable to have instrumentation associated with the initialization of global variables. However, this can often occur outside of any function where instrumentation cannot be added. To address this problem, a special global variable processing function is created for each file that is compiled. Only instrumentation code that pertains to global variable initialization will be executed in this function. At the beginning of function `main`,

1. The index for the first argument is zero.

Table 4.1: Instrumentation functions parameters.

Parameter	Type	Description
$file_name	`char *`	Name of the current file.
$line_num	`int`	Current line number.
$smpl_line_num	`int`	Current simple statement number
$id	`int`	Identifier that will be unique each time this parameter is used.
$tree_code	`enum pat_code`[1]	Code that represents the current operation.
$elt_offset	`int`	Offset between the given function argument or field and the first function argument.[2]
$fn_name	`char *`	Name of function for patterns that use a function name qualifier (such as `$call_expr`).
$fn_arg_name(*index***)**	`char *`	Name of argument.
$fn_arg_type(*index***)**	`char *`	Type of argument.
$fn_arg_size(*index***)**	`unsigned int`	Size of argument.
$fn_arg_obj_size(*index***)**	`unsigned int`	Size of object that is being pointed to by a pointer argument.
$fn_arg_addr(*index***)**	`void *`	Address of the argument.
$fn_arg_value(*index, type***)**	*argument type*	Value of the argument.
$fn_arg_lb(*index***)**	`long long`	Lowest possible value, based strictly on the argument type.
$fn_arg_ub(*index***)**	`long long`	Highest possible value, based strictly on the argument type.
$expr_type	`char *`	Type of the expression.
$expr_size	`unsigned int`	Size of the type of expression.
$expr_obj_size	`unsigned int`	Size of object that is being pointed to by the result of an expression.
$expr_value(*type***)**	*argument type*	Value of the expression.
$expr_lb	`long long`	Lowest possible value, based strictly on the expression type.
$expr_ub	`long long`	Highest possible value, based strictly on the expression type.
$op_name(*index***)**	`char *`	Name of the operand.
$op_type(*index***)**	`char *`	Type of the operand.
$op_size(*index***)**	`unsigned int`	Size of the operand.
$op_obj_size(*index***)**	`unsigned int`	Size of object that is being pointed to by a pointer operand.
$op_addr(index)	`void *`	Address of the operand.
$op_value(index, type)	*argument type*	Value of the operand.
$op_lb(*index***)**	`long long`	Lowest possible value, based strictly on the operand type.
$op_ub(*index***)**	`long long`	Lowest possible value, based strictly on the operand type.

1. Requires the instrumentation routines to include a file "muse-ext.h" to access the enum type.
2. This is used instead of passing the argument number as fields of a structure will have different offsets but the same argument number.

all of the global variable processing functions are called.

Due to special nature of these global processing functions, there are situations where instrumentation should not be added despite a match. Any pattern that is preceded by the keyword “`$ignore_gbl`” will be ignored during the global processing functions but active for the remainder of the program.

One unsolved problem is processing local static variables that are declared within a function. They cannot be processed in the special processing function as it violates scoping rules enforced by GCC. However, during our analysis, this only created two problems and in both cases, specialized instrumentation was used to eliminate the problem.

4.2.7 System Calls

System functions are not commonly part of the source code of the function and consequently will not be instrumented. Instead, the burden falls on the user to write instrumentation for calls to system functions that are important to them. Alternatively, a user can supply the source code for system functions and MUSE will instrument those functions like normal functions. In our experiments, we use the former approach and write instrumentation for system functions.

In most cases, there will be a moderate number of system calls the user is interested in (such as string and input functions) and a large number of system calls the user is not interested in. Default patterns can be used for instrumenting uninteresting system calls if desired.

There could be situations where it may be easier to not instrument the body of the func-

tion, but instead summarize the functionality using instrumentation when the function is called. As an example, someone could write their own memory allocation function in which the underlying details are unimportant, just the fact that space has been allocated. One obvious way to accomplish this task is to not instrument the file where the function resides. However this solution is unacceptable if the file contains other functions that need to be instrumented. Instead, the pattern language supplies an `$ignore` directive for this purpose. The directive "`$ignore my_malloc`" would direct MUSE not to instrument the function `my_malloc`.

4.3 Creating Correctness Specifications

To illustrate the process of creating a specification, this section describes a simple example of a checker that will intercept every dereference operation and check if the pointer is null. If the pointer is null, an appropriate error message will be displayed and the program will exit. In addition, we will add some tracing instrumentation that will optionally display a message indicating when the program has entered and left each function.

Figure 4.3 shows the MUSE pattern file. The first two patterns trigger at the beginning and end of each function and are used for the tracing portion of the instrumentation. The `$any_fn` operand indicates instrumentation will be applied to every function. When there is a match, two parameters are passed to the instrumentation: the function name and the current line number. These will be used in the trace output. The last two patterns match against all dereference operations. Two patterns are necessary because MUSE distinguishes between dereferences that occur on the left hand side of an assignment (`$lhs_indirect_ref`) versus those that occur on the right hand side (`$indirect_ref`).

This distinction is not relevant so they both call the same instrumentation function. The file name and current line number are the first two parameters passed into the null checking function and are used in the error message if the pointer is null. The third parameter passes the current value of the pointer into the function so it can be tested for null.

The instrumentation functions are shown in Figure 4.4. The tracing functions are straightforward, printing a message that displays the name of the function they are entering or exiting. The trace messages only are only displayed if the global variable `debug_on` is set. The null checking function compares the pointer to null. If the pointer is null, an error message is printed and the program exits. Otherwise, the instrumentation silently returns control back to the original program.

To further illustrate how the process works, consider the sample program in Figure 4.5. A simplified version of this program is shown in Figure 4.6 (left). Temporary variables have the names `T.1`, `T.2`, etc.[1] Some things to notice from this code are how the short-circuited logical-or is replaced with an if statement and how arithmetic expressions get split into multiple statements using temporary variables. Figure 4.6 (right) shows the instrumented code with the added instrumented calls highlighted in bold.

4.4 Benchmarks and Testing Methodology

Programs that we used for our experiments are listed in Table 4.2. All programs were compiled using GCC and with an -O4 optimization level. Four of the programs (*anagram*, *ft, ks*, and *yacr2*) are from the pointer-intensive benchmark suite [80] and were selected

1. The temporary variable names given in GCC are illegal in C as they contain periods but are legal in the AST phase of GCC.

```
$begin_fn $any_fn:
after: function_begin($fn_name, $line_num);

$end_fn $any_fn:
before: function_end($fn_name, $line_num);

$lhs_indirect_ref $any_type $any_var:
before: check_null_pointer($file_name, $line_num,
  $op_value($src0, $pointer_type));

$indirect_ref $any_type $any_var:
before: check_null_pointer($file_name, $line_num,
  $op_value($src0, $pointer_type));
```

Figure 4.3: Pattern file for null checker.

```
#include <stdio.h>
static int debug_on = 0;

void function_begin(const char *fn_name, int lineno)
{
  if (debug_on) fprintf(stderr, "TRACE: Start function %s at line %d\n", fn_name, lineno);
}

void function_end(const char *fn_name, int lineno)
{
  if (debug_on) fprintf(stderr, "TRACE: End function %s at line %d\n", fn_name, lineno);
}

void check_null_pointer(const char *file_name, int lineno, void *p)
{
  if (p == NULL) {
    fprintf(stderr, "FATAL ERROR: Accessing null pointer at %s:%d\n", file_name, lineno);
    exit(-1);
  }
}
```

Figure 4.4: Instrumentation functions for null checker.

```
void foo(int *x, int **y)
{
  if (*x == **y || *x != 12)
    **y = -4;
  else
    *x = 23;
}

int main(void)
{
   int a, b;
   int *a_p, *b_p;
   int **a_pp, **b_pp;

   a = 5;
   b = a + 3 * a;
   a_p = &a;
   b_p = &b;
   a_pp = &a_p;
   b_pp = &b_p;
   **a_pp = 10;
   **b_pp = *b_p + *a_p;
   foo(a_p, b_pp);
   printf("a: %d, b: %d\n", a, b);
}
```

Figure 4.5: Sample program for use with null checker.

```
void foo (int * x, int **y) {

  int T.1, T.3, T.4, T.5, T.6;
  int *T.2, *T.7;

  T.1 = *(x);

  T.2 = *(y);

  T.3 = *(T.2);
  T.4 = T.1 == T.3;
  T.6 = T.4 == 0;
  if (T.6) {

    T.5 = *(x);
    T.4 = T.5 != 12;
  }
  if (T.4) {

    T.7 = *(y);

    *(T.7) = -4;
  } else {

    *(x) = 23;
  }

}

int main () {

  int a, b, *a_p, *b_p;
  int **a_pp, **b_pp;
  int T.8, T.11, T.12;
  int *T.9, *T.10;
  char *T.13, *T.14;
  const char *T.15;
  a = 5;
  T.8 = a * 3;
  b = a + T.8;
  a_p = &(a);
  b_p = &(b);
  a_pp = &(a_p);
  b_pp = &(b_p);

  T.9 = *(a_pp);

  *(T.9) = 10;

  T.10 = *(b_pp);

  T.11 = *(b_p);

  T.12 = *(a_p);

  *(T.10) = T.11 + T.12;
  foo(a_p, b_pp);
  T.13 = "a: %d, b: %d\n";
  T.14 = (char * )T.13;
  T.15 = (const char * )T.14;
  printf(T.15, a, b);

}
```

```
void foo (int * x, int ** y) {
  function_begin("foo", 6);
  int T.1, T.3, T.4, T.5, T.6;
  int *T.2, *T.7;
  check_null_pointer("ref2.c", 6, x);
  T.1 = *(x);
  check_null_pointer("ref2.c", 6, y);
  T.2 = *(y);
  check_null_pointer("ref2.c", 6, T.2);
  T.3 = *(T.2);
  T.4 = T.1 == T.3;
  T.6 = T.4 == 0;
  if (T.6) {
    check_null_pointer("ref2.c", 6, x);
    T.5 = *(x);
    T.4 = T.5 != 12;
  }
  if (T.4) {
    check_null_pointer("ref2.c", 7, y);
    T.7 = *(y);
    check_null_pointer("ref2.c", 7, T.7);
    *(T.7) = -4;
  } else {
    check_null_pointer("ref2.c", 9, x);
    *(x) = 23;
  }
  function_end("foo", 10);
}

int main () {
  function_begin("main", 14);
  int a, b, *a_p, *b_p;
  int **a_pp, **b_pp;
  int T.8, T.11, T.12;
  int *T.9, *T.10;
  char *T.13, *T.14;
  const char * T.15;
  a = 5;
  T.8 = a * 3;
  b = a + T.8;
  a_p = &(a);
  b_p = &(b);
  a_pp = &(a_p);
  b_pp = &(b_p);
  check_null_pointer("ref2.c", 27, a_pp);
  T.9 = *(a_pp);
  check_null_pointer("ref2.c", 27, T.9);
  *(T.9) = 10;
  check_null_pointer("ref2.c", 28, b_pp);
  T.10 = *(b_pp);
  check_null_pointer("ref2.c", 28, b_p);
  T.11 = *(b_p);
  check_null_pointer("ref2.c", 28, a_p);
  T.12 = *(a_p);
  check_null_pointer("ref2.c", 28, T.10);
  *(T.10) = T.11 + T.12;
  foo(a_p, b_pp);
  T.13 = "a: %d, b: %d\n";
  T.14 = (char * )T.13;
  T.15 = (const char * )T.14;
  printf(T.15, a, b);
  function_end("main", 31);
}
```

Figure 4.6: Sample program simplified (l) and instrumented with null checking (r).

Table 4.2: Programs used during testing.

Pointer Intensive Benchmarks		Networking Servers and Applications	
anagram	anagram generator	betaftpd	file transfer protocol daemon
ft	fast Fourier transform	gaim	instant messenger
ks	graph partitioning	ghttpd	web server
yacr2	yet another channel router	openssh	openssh secure shell
		thttpd	web server

due to the difficulty of analyzing these programs statically. The other five programs are networking applications, including the popular secure shell program *openssh* and *gaim*, a popular instant messaging program. The other three server programs include a FTP server (*betaftpd*) and two web servers (*ghttpd* and *thttpd*).

For experiments involving finding bugs, we ran a variety of different tests. We did not attempt to test the programs thoroughly using sophisticated software testing techniques. Our goal was to demonstrate that our techniques were effective at finding bugs rather than completely testing each program. For the pointer intensive benchmarks, we used the set of provided tests for our testing. The networking applications were primarily tested by manually hammering them with several commands while running them in a number of different configuration. For *openssh*, our testing primarily focused on the server program `sshd` but we also did some testing on the client `ssh`. In *gaim*, testing occurred primarily within three instant messaging protocols: ICQ, MSN, and Yahoo. We also had some directed tests, including all tests that found errors, in order to provide a level of consistency when enhancements were made to the infrastructure and checker.

To measure performance, we used two metrics: time and dynamic instruction count. Time was measured using the UNIX `time` command. However, we were concerned with the

margin of error from using this command so we also measured performance by keeping track of the dynamic instruction count. The dynamic instruction count was obtained by profiling the program and instrumentation routines using `gprof`. Due to problems beyond our control, we were unable to gather baseline dynamic instruction count metrics for *ghttpd* and *openssh*. The dynamic instruction count is precise but is not sufficient as it does not account for the fact that different instructions take different lengths of time. This is particularly important as the instrumentation increases the memory footprint of the program, thereby increasing the number of long-latency memory operations.

When testing the performance of the pointer-intensive benchmarks, we used the longest running test that was included in the test suite. In the case of *anagram*, where all of the provided tests were too short, we created our own test. The performance test of *betaftpd* consisted of transferring a very large file. Similarly the test for *ghttpd* consisted of sending an extremely big web page. The test for the other web server *thttpd* was different, as it was comprised of a sequence of one thousand requests for average-sized web pages. The *openssh* performance test only analyzed the server `sshd` and consisted of a scripted session with a (uninstrumented) client. Due to the interactive nature of *gaim*, it was not used in any performance experiments.

4.5 Performance Impact of Simplification and Instrumentation Calls

Our initial performance experiments measured the effect of the simplification process and how much compiler optimizations were hindered when instrumentation calls were present. Table 4.3 shows the result of our initial performance experiment. The baseline column refers to the time it takes to execute a program without the simplification process or any

Table 4.3: Effects on run-time performance when adding instrumentation.

Program	Baseline Time[1]	Simplification Only		Empty Instr. (Worst)		Empty Instr. (Best)	
		Time	Ratio[2]	Time	Ratio	Time	Ratio
anagram	0.06	0.07	1.17	0.39	6.50	0.15	2.50
ft	0.18	0.25	1.39	0.62	3.44	0.18	1.00
ks	0.05	0.05	1.00	0.41	8.20	0.09	1.80
yacr2	0.12	0.14	1.17	2.25	18.75	0.99	8.25
betaftpd	0.07	0.05	0.71	0.17	2.43	0.15	2.14
ghttpd	0.52	0.66	1.27	0.67	1.29	0.63	1.21
openssh	0.70	0.85	1.21	1.09	1.56	0.96	1.37
thttpd	0.15	0.25	1.67	0.39	2.60	0.36	2.40

1. All time measurements (this and future results) are given in *seconds*.
2. Ratios, unless specified otherwise, are with respect to the baseline.

instrumentation. The "Simplification Only" column looks at the performance of programs that go through the simplification process but no instrumentation is added. The last two columns add instrumentation calls according to our input checker described in Chapter 3. However, the bodies of the instrumentation routines do nothing but immediately return. The purpose of this experiment is to determine the affect of adding instrumentation will have on the program in terms of lost opportunity for compiler optimizations and the overhead associated with the calls. The worst-case instance of this experiment refers to the case where the added instrumentation is unoptimized. In the best-case, useless instrumentation sites have been removed (see Chapter 6).

From Table 4.3, it is apparent that the simplification process has a small negligible effect on performance. This makes sense as the compiler is able to recoup most of the performance lost when optimizing the code. When instrumentation is added, the performance degradation is noticeable. For the four pointer-intensive benchmarks, the average increase

Table 4.4: Effect on dynamic instruction count when adding instrumentation. In this table, instruction counts are given in thousands of instructions. Due to problems beyond our control, we were unable to gather baseline dynamic instruction count metrics for *ghttpd* and *openssh*.

Program	Baseline Count	Simplification Only		Empty Instr. (Worst)		Empty Instr. (Best)	
		Count	Ratio	Count	Ratio	Count	Ratio
anagram	12,736	13,398	1.05	59,210	4.65	24,575	1.93
ft	67,038	67,134	1.00	143,356	2.14	67,376	1.01
ks	51,092	50,425	0.99	109,004	2.13	54,800	1.07
yacr2	185,007	195,475	1.06	489,721	2.65	323,108	1.75
betaftpd	1,728	1,982	1.15	7,556	4.37	6,854	3.97
thttpd	16,958	17,394	1.03	39,192	2.31	26,623	1.57

in time was 6x and ranged from 1.3x (*ft*) to 19x (*yacr2*). The network applications did not suffer a significant performance increase. This difference is due to the fact that the pointer intensive benchmarks have more instrumentation sites and more computationally intensive loops. As a result, the programs benefit more from compiler optimizations but the added instrumentation prevents the compiler from taking full advantage of the optimizations since the compiler must be conservative when optimizing across function calls.

The dynamic instruction count, shown in Table 4.4, increased relatively consistently for all programs ranging from 2.13x (*ks*) to 4.65x (*anagram*). The dynamic instruction count did not correlate well with the time. This is due to that some of the additional instruction count in this experiment is due to instructions that are responsible for passing parameters. These instructions are relatively quick, either moving parameters to the appropriate register or storing on them on the stack. In future experiments, the added dynamic instrumentation count associated with the instrumentation routine will be the primary source of added instructions and the instructions used for parameter passing will be a second-order effect at best.

Table 4.5: Effect on compile time when adding instrumentation.

Program	Baseline Compile Time	Simplification Only		Empty Instr. (Worst)		Empty Instr. (Best)	
		Time	Ratio	Time	Ratio	Time	Ratio
anagram	0.19	0.24	1.26	1.02	5.37	0.46	2.42
ft	0.40	0.41	1.03	1.17	2.93	0.70	1.75
ks	0.26	0.31	1.19	1.14	4.38	0.67	2.58
yacr2	1.09	1.32	1.21	8.57	7.86	4.94	4.53
betaftpd	0.82	0.96	1.17	4.31	5.26	2.58	3.15
ghttpd	0.50	0.63	1.26	2.59	5.18	1.63	3.26
openssh	21.67	24.37	1.12	113.77	5.25	50.76	2.34
thttpd	2.71	3.70	1.37	14.97	5.52	8.64	3.19

The performance in the best-case scenario improved for each program as there were fewer instrumentation calls in each program. Only *yacr2* still suffers a performance degradation greater than 3x. It still exhibits a slowdown of over 8x because it does not benefit from the static optimizations as much as the other programs did.

The time to compile programs and the size of the programs also increased when instrumentation is added. Table 4.5 shows how long it takes to compile the program using MUSE. The time to compile an instrumented program includes the time to add the instrumentation since MUSE runs as a phase in the compiler. Table 4.6 shows how the size of the program is affected when instrumentation is added. One interesting result is that simplifying the program (with no additional instrumentation) creates a smaller program. This is likely due to the compiler being able to find more dead code when statements are simplified. The increase in the size of the program is dependent on the number of instrumentation sites. Not surprisingly, *yacr2* has a large number of instrumentation sites and suffers from the largest increase in code size and run-time performance. In this experiment, the

Table 4.6: Effect on code size when adding instrumentation.

Program	Baseline Size (KB)	Simplification Only		Empty Instr. (Worst)		Empty Instr. (Best)	
		Size	Ratio	Size	Ratio	Size	Ratio
anagram	18.1	18.0	0.99	112.2	6.19	51.0	2.81
ft	24.4	24.3	0.99	103.3	4.23	67.3	2.76
ks	19.9	19.5	0.98	110.3	5.55	74.7	3.76
yacr2	39.3	37.6	0.96	547.4	13.94	285.1	7.26
betaftpd	37.6	36.5	0.97	345.5	9.19	200.8	5.34
ghttpd	33.6	33.4	0.99	197.1	5.87	137.7	4.10
openssh	1600.1	1595.7	1.00	5236.7	3.27	3298.8	2.06
thttpd	143.7	144.0	1.00	986.0	6.86	624.2	4.34

instrumentation model accounted for 23,880 bytes[1] of the added size and the remainder is due to the instrumentation calls themselves.

4.6 Example Application: Memory Access Checking

As an exhibition of our instrumentation infrastructure, we created a memory access checker using MUSE. This checker is similar in nature to Safe C [4]. In this approach, additional state is stored with each pointer that indicates the base address and the size of the intended object the pointer is supposed to point to. When the pointer is dereferenced or used in an array reference, a check occurs to make sure the pointer is accessing memory within the object it is pointing to. In addition, a capability is stored with the pointer that serves as an id for the object it is pointing to. A capability table keeps track of all objects that are currently live. When an object goes out of scope or is freed (for heap objects), the id is removed from the capability table and never used again for the duration of the pro-

1. This amount may seem large for code that only contains functions that do nothing but return. But as you will see in Section 5.1, we use a large number of instrumentation functions. There is a trade-off between a large number of fast specific instrumentation functions or a fewer number of slower more general-purpose functions.

gram. During pointer dereferences, the check will also make sure the capability is valid - insuring the pointer is pointing to its intended object.

Within the memory access checker, the additional state associated with the pointers is stored in a shadow state table. The shadow state table is implemented using a hash table that is indexed using the address of the pointer. Each entry in the hash table contains four fields: base address, size, capability id, and mode. The first three fields were discussed in the previous paragraph. The mode provides the instrumentation information about the type of data the pointer is pointing to and is used to catch errors such as freeing memory that is not on the heap or trying to invoke a function using a pointer that does not point to a function.There are five possible modes:

- ARRAY: Points to a local or globally declared array.
- HEAP: Points to dynamically allocated memory.
- FUNCTION: Points to a function.
- CONSTANT: The pointer is a constant or is a variable that is used as an offset in an arithmetic operation. For example, given the expression `p+i` where `p` is a pointer and `i` is an integer, GCC will create a pointer temporary value that has the value of `i`. This pointer will be considered a constant pointer.
- PLAIN: Any pointer that does not fit any of the above classifications. This includes pointers to local variables (except arrays).

Since arrays and pointers are considered separate variables in GCC, a separate shadow table is kept for arrays. Keep in mind that GCC does not consider arrays created on the heap to be an array - all arrays references are done via pointers. The shadow state table for

arrays is the same as the pointer shadow state table with two key differences. The first difference is that arrays are indexed by the base address. The second difference is that the size is the only field that is necessary. The base address is not needed since it is used as the index. The mode will always be ARRAY. Capabilities are also unnecessary since only arrays that are in scope are kept in the array shadow state table.

The pattern file for the memory access checker is presented in Figures 4.7 and 4.8. For brevity, the parameters passed into the instrumentation functions are not indicated and patterns that are nearly identical to other patterns have been omitted. The numbers next to the patterns are for reference purposes and not part of the pattern.

Pattern 1 initializes the checker by creating the capability table. Patterns 2 and 3 create entries for locally and globally declared arrays respectively. The `create_array_state` function creates entries for the array shadow state table. Pattern 4 triggers when a locally declared array goes out of scope and as a result the array is removed from the shadow state table. Patterns 5-9 correspond to situations where pointer entries are created in the shadow state table. Slightly different functions are used but, for the most part, they do the same thing. This is not a complete list of pointer creation patterns - for instance, there are patterns for `calloc` and `realloc` in addition to `malloc`.

Patterns 10-13 are used to propagate shadow state across function boundaries. This is also not a full list of the patterns necessary. The same protocol is used in the input bounds checker and is described fully in Appendix C. To summarize, pattern 10 is used to temporarily store shadow state for arguments while control is transferred to the function. Pattern 11 transfers the temporary state to the corresponding parameter. Patterns 12 and 13 accom-

```
   // Initialization
 1 $begin_fn main:
   after:init_checker();

   // Creation of array and pointer state
 2 $birth_local $array_type $any_var:
   after:create_array_state();

 3 $birth_global $array_type $any_var:
   after:create_array_state();

 4 $death_local $array_type $any_var:
   before:remove_array_state();

 5 $assign $lval $pointer_type $any_var $pointer_type $any_const:
   before:create_ptr_state();

 6 $assign $lval $pointer_type $any_var
           $addr_expr $any_type $except($array_type, $function_type) $any_var:
   before: create_ptr_state();

 7 $assign $lval $pointer_type $any_var $addr_expr $array_type $any_var:
   before: create_ptr_state_from_array();

 8 $assign $lval $pointer_type $any_var $addr_expr $function_type $any_var:
   before: create_fn_ptr_state();

 9 $assign $lval $pointer_type $any_var $malloc_expr:
   after:  process_malloc();

   // Propagation across function boundaries
10 $call_expr_arg $any_fn $pointer_type $any_var:
   before: store_ptr_fn_arg();

11 $birth_parm $any_fn $except(main) $pointer_type $any_var:
   after:  propagate_ptr_fn_arg();

12 $return_stmt $any_fn $except(main) $pointer_type $any_var:
   before: store_ptr_return_value();

13 $assign $lval $pointer_type $any_var $call_expr $any_fn:
   after:  propagate_ptr_return_value();

   // Scope altering events
14 $begin_scope:
   after:add_local_capability();

15 $switch_stmt $any_type $any_value:
   before:add_local_capability();

16 $end_scope:
   before:remove_local_capability();

17 $continue_stmt:
   before:remove_local_capability();

18 $break_stmt:
   before:remove_local_capability();

19 $end_fn $any_fn:
   before:remove_local_capability();
```

Figure 4.7: Pattern file for memory access checker (part 1).

```
   // Pointer arithmetic and copy operations
20 $assign $lval $pointer_type $any_var $pointer_type $any_var:
   before:process_ptr_copy();

21 $assign $lval $pointer_type $any_var
   $component_ref $any_type $any_var $any_type $any_var:
   before:process_ptr_copy();

22 $assign $lval $pointer_type $any_var
   $array_ref $any_type $any_var $any_type $any_value:
   before:process_ptr_copy();

23 $assign $lval $pointer_type $any_var
   $indirect_ref $any_type $any_var:
   before:process_ptr_copy();

24 $assign $lval $pointer_type $any_var
   $ind_component_ref $any_type $any_var $any_type $any_var:
   before:process_ptr_copy();

25 $assign $lval $pointer_type $any_var
   $ind_array_ref $any_type $any_var $any_type $any_value:
   before:process_ptr_copy();

26 $assign $lval $pointer_type $any_var
           $plus_expr $pointer_type $any_var $pointer_type $any_const:/
   before: process_ptr_copy();

27 $assign $lval $pointer_type $any_var
           $plus_expr $pointer_type $any_const $pointer_type $any_var:
   before: process_ptr_copy();

28 $assign $lval $pointer_type $any_var
           $minus_expr $pointer_type $any_var $pointer_type $any_value:
   before: process_ptr_copy();

29 $assign $lval $pointer_type $any_var
           $plus_expr $pointer_type $any_var $pointer_type $any_var:
   before: process_ptr_binary_expr();

   // Uses of pointers and arrays
30 $indirect_ref $any_type $any_var:
   before: process_indirect_ref();

31 $ind_component_ref $any_type $any_var $any_type $any_var:
   before: process_indirect_ref();

32 $array_ref $array_type $any_var $any_type $any_value:
   before: process_array_ref();

33 $array_ref $pointer_type $any_var $any_type $any_value:
   before: process_array_ref_via_ptr();

34 $ind_array_ref $any_type $any_var $any_type $any_value:
   before: process_indirect_array_ref();

35 $call_expr $any_fn_ptr:
   before: process_fn_ptr_call();

36 $call_expr free:
   before: process_free();
```

Figure 4.8: Pattern file for memory access checker (part 2).

plish the same task when a pointer is returned.

Patterns 14-19 signify events that cause the scope of the program to change. In general, a new scope is created when a '{' (`$begin_scope`) is encountered. A new scope is also created when a switch statement is executed since the next '{' (`$end_scope`) will be skipped. Scopes are stored using a stack and removing a scope is a bit trickier since more than one scope can be removed in an operation. Pattern 16 which is equivalent to a '}' is straightforward - remove the current scope. A continue statement removes the scope associated with the innermost loop and removes all scopes contained within the loop. A break statement is similar but must account for scopes due to switch statements as well as loops. Ending a function whether it is by bottoming out of a function or via a return statement removes the scope associated within the function and any scopes contained within the function.

Pointer copy operations are displayed in patterns 20-28. Patterns 20-25 are straightforward copy operations where the shadowed state is transferred verbatim from the source to the destination. Patterns 26-28 correspond to addition and subtraction operations where one operand is a constant. These operations also copy the shadowed state without modifications even though the destination pointer is likely not pointing to the base address of the object. This is not reflected in the shadowed state but is accounted for by looking at the value of the pointer when checking occurs. Pattern 29 is different in that handles the addition of two pointer variables. This is only legal if and only if one of the pointers is an offset and is in CONSTANT mode. The offset variable is treated as a constant and state from the other pointer is copied into the destination pointer's shadowed state entry.

The next set of patterns (30-34) are used for checking indirect and array references. Different functions are used depending on the type of array reference. The functions compute the allowable range given the base address and size. The checks use the current value of the pointer and the index (for array references) to determine if the access is within the allowable bounds of the object. In addition, the capability is checked to ensure the intended object is still valid. Patterns involving dereferences and array references on the left hand side (as discussed in the null checker example) are also needed but have been omitted for brevity. Pattern 35 is for function calls and ensures the pointer is actually pointing to a function. Pattern 36 checks for common errors associated with `free`: freeing the same memory twice, attempting to a free an object that is not dynamically allocated, and using an address that is not the base of an object. In addition, the function removes the capability associated with the freed object.

We were unable to find any bugs in the benchmark programs with the memory access checker aside from some directed tests we created. The primary reason for this is that we did not thoroughly test programs using this checker. The goal of creating the memory access checker is to show the general purpose nature of MUSE and how it can be used to create checkers similar to those that are available today.

4.7 Chapter Summary

This chapter describes MUSE, our instrumentation infrastructure. MUSE works by matching user-specified patterns against a simplified form of the C language. When a match is detected, instrumentation is added and can either be a call to an external function or, for a greater flexibility, an inlined sequence of arbitrary C statements. Users are able to specify

instrumentation to do a wide variety of tasks including bug detection, profiling, debugging, and code coverage. Two example specifications were described: a null checker with function tracing and a memory access checker.

Early results show that simplified form of C does not impact the performance of the program but adding instrumentation does. An experiment where empty instrumentation calls were added to the program caused the performance to increase by a factor of six. This is likely due to the lost opportunity for the compiler to optimize across the instrumentation call boundaries. Not surprisingly, adding instrumentation calls also increased the time it took to compile the program and the overall size of the program.

CHAPTER 5

IMPLEMENTATION AND RESULTS

5.1 Implementation

This section describes how our input bounds checker was implemented using MUSE. The implementation consists of three parts: the pattern file, an instrumentation file within GCC, and external instrumentation functions. The pattern file is presented in Appendix C.

Many of the functions specified in the pattern file use the inline designation to take advantage of information in the compiler. For example, the wildcard `$any_binary_op` is used for all of the binary operation patterns. Separate patterns do not exist for different binary operations such as add, subtract, etc. However, separate external instrumentation routines do exist for all of these operations and the internal instrumentation function within GCC is responsible for making sure the right function is selected based on the operation. Alternatively, we could have written separate patterns for each of the different binary operators. It has no bearing on the resulting instrumentation we found that this organization was more straightforward since it allows for similar operations to be grouped together, a deficiency of the pattern description language. On the other end of the spectrum, we could have one external instrumentation routine for all binary operations. In this case, the operation would be passed to the function. The function would have to figure out what the operation was

and act accordingly. This organization degrades performance since a run-time decision is made on the operation, something that is known at compile-time. The advantage of this approach is that it will produce smaller instrumented programs since there are fewer functions. We chose to use separate instrumentation functions because we are more concerned with speed than size.

Advanced inline features such as using arbitrary statements instead of function calls were not used. We did try replacing some of the less complex instrumentation routines by using a series of statements. There was some performance improvement but for the most part insignificant. Since instrumentation in this fashion hindered our ability to gain accurate statistics, we did not use this approach.

The external instrumentation functions keep track of the shadowed state associated with the variables and perform checks when dangerous operations occur. In our initial implementation, the shadowed state is stored in two hash tables. One table stores state for integers and the other is used for arrays and pointers. All arrays are inserted in the array table when they are created and are indexed by their base address. Each entry in the array table contains five fields: `actual_size`, `max_str_size`, `known_null`, `base_addr`, and `is_input`. The first three fields are described in Section 3.2. The `actual_size` field is also used in the array reference checking. The `base_addr` field is the base address of the array. The `is_input` field is used to mark arrays that contain program input. If an array from input is used in a function that converts a string to an integer, such as `atoi`, the resulting integer will be treated as input.

Pointers also are stored in the table indexed by their current value. The `base_addr` field

stores the base address of the array or object the pointer is pointing to. When accessing the max_str_size and known_null fields, another access to the array hash table is made using the base_addr if the pointer value is not equal to the base address. The extra level of indirection is not needed for actual_size since it never changes once the array is created. Entries for pointers in the hash table will have an accurate size.

The integer table only contains entries for integers that currently contain shadowed state. An integer can require state for three reasons: (a) it contains input data, (b) it contains string length data, or (c) it is a boolean value that will narrow the bounds if used in a conditional expression. A mode field is used to distinguish between the three cases. In case (a), there are upper and lower bound fields as described in Section 3.1. In case (b), there are string and size difference fields as in Section 3.2. In case (c), the entry contains an address indicating the variables[1] to be updated, the appropriate bounds (lb, ub, and max_str_size) when the condition is true and bounds for when the condition is false. If the conditional value is used in a control statement, the bounds of each variable are updated using this information.

Integer operations that update state will differ depend on the operation. In general, the integer hash table will be accessed for each of its operands. Run-time values will be used for constants and variables that currently do not contain input data. If no operand contains input data, the entry associated with the destination will be removed from the hash table if one exists. Otherwise, the destination's hash table entry will be updated, or created if it

1. A maximum of two variables can be updated for a single conditional variable (as is the case when the statement is c = a < b and both a and b contain input data. Remember that short-circuited logical operators get converted into control operations, eliminating the possibility of a conditional variable being a function of other conditional variables and eliminating the situation where a conditional variable can limit an arbitrary number of variables.

does not exist, with the appropriate state based on the current operation and the operands. During a function call, shadow state needs to be passed from the caller's argument to the callee's parameters. This is accomplished using a separate table that is indexed by offset (assuming the parameters are laid out sequentially in memory). Using offset as an index allows integers within structures to have different indices. The same approach is used when a structure is returned from a function. When an integer is returned from a function, a special variable will temporarily hold the shadow state of the return value until control is transferred back to the callee.

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

5.2 Validating the Implementation

Since we assume the checker is free of bugs during its application, it is necessary to validate it before using it. To accomplish this, we used a variety of techniques. Our first and foremost concern was that we were not missing any patterns that need to be detected. To test this, we created some generic patterns that captured every time an integer or pointer is written to (or appears on the left-hand side of an assignment operation). In each case, there

needed to be some other instrumentation call that captured the entire assignment operation. This caught several missing patterns. We also did a rigorous examination of the *Elemental C* grammar and determined what different patterns are possible and necessary for the input checker.

Within the instrumentation, a message will be printed out when a situation that is currently not handled is encountered. For instance, we do not support the case when a string length is used in a shift operation. If this happens, a panic message is displayed. To further validate the functionality of the checker, we created a variety of directed tests and also ran the checker on real programs. The real programs, being larger, exposed more errors. Most of the errors triggered a situation that was not handled or signalled an error that should not have been reported. This testing was not sufficient for cases when the instrumentation thought something did not hold input data when it reality it did. To handle this situation, additional tests and code inspection focused on making sure that input data was properly propagated. Once we had a high level of confidence that the system was working, we could validate any enhancements by comparing the output to a known output - typically the output consisted of the list of errors, list of false alarms, and a breakdown of the number of useful calls by instrumentation site.

5.3 Bugs Detected

Using our input checker, we were able to find 17 bugs in the 9 programs, shown in Table 5.1.

The two defects found in *openssh* (described in Chapter 3) are both security flaws present in version 3.0.2. The channel id bug would be difficult to locate via static analysis. The

Table 5.1: Bugs detected during testing.

Program	Defects Found	False Alarms	Program	Defects Found	False Alarms
anagram	2	0	betaftpd	2	1
ft	2	0	gaim	1	1
ks	3	0	ghttpd	3	2
yacr2	2	1	openssh	2	0
			thttpd	0	1

array is dynamically allocated and its size can change during execution. In addition, creation of the array, reading of input, and accessing the channel array each occur in three distinct functions. Any static approach to locating this bug would require interprocedural analysis.

In *gaim*, a defect occurs when reading the configuration file. Each field is placed into a large temporary buffer. The fields are processed and copied into a data structure. In some cases, the fields are copied into a smaller buffer without checking to see if it will properly fit. Examples of fields where this occurs are the username and password. While this bug could not be exploited remotely, it could cause the program to crash.

The three defects in *ghttpd* were all due to misuse of string functions. In one case, a `strncat` function contains a limit that does not account for the null character. For a given limit `n`, it is possible for `n+1` characters to be written since a null character is always written. Another defect was caused by calling `strstr` on an uninitialized local array. The third defect in *ghttpd* and one of the defects found in *betaftpd* were the result of using data received from the network without any guarantee that there is a null character. The other defect in *betaftpd* is the result of too many characters in the source strings of a `snprintf`

command. This does not cause a buffer overflow but can truncate a long path name during the copy causing undesirable results.

One bug in *anagram* permits a user to overflow a buffer with characters from an input file. The buffer is dynamically allocated in proportion to the size of the input file. Extra space is added to store additional information about each word in the file. The size of the extra space is controlled by a fixed compile-time constant representing the maximum number of words allowed in the file. If the file contains more words than this constant, the buffer could overflow. The other bug is the result of using `gets`, automatically a dangerous function. In *ks,* two bugs resulted from an input being used to reference an array without any checking to see if it exceeded the array bounds. The other defect and one of the two defects in *ft* are due to a loop based on input that is not checked. The other defect in *ft* is attributed to the lack of a check for a negative value for inputs that indicate the size of the data stream. Both defects discovered in *yacr2* were due to a multiplication overflow that were used in dynamic allocation of arrays. These bugs are very similar to the OpenSSH challenge bug in Figure 3.4.

All of the bugs that were detected using our checker were not exposed during the particular test. In fact, none of the tests we used caused any output errors or crashes. This is important in that it shows our technique is effective at finding bugs without having the precise input. Another positive aspect is that it is possible to write a test that exposes each bug in all cases except one. The lone exception is the *ghttpd* defect that stems from the use of an uninitialized array. In this case, the error is dependent on the garbage value of the uninitialized array rather than an input value.

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

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

We also performed some experiments to detect arithmetic overflow. In general, we found that checking for arithmetic overflow hindered run-time performance, triggered too many false alarms and not enough useful bugs. As a result, we did not consider arithmetic overflow for the bugs that were found above nor in the performance experiments that follow. However, we were able to find two bugs that we did not find otherwise. One defect was in *openssh*. The defect occurred when one was added to an unbounded integer. It is not

exploitable but could lead to undesirable results. The other defect occurred in *ks*. In addition, the arithmetic overflow checking errors also triggered for four of the errors that were detected when overflow checking was absent. However, the arithmetic overflow error was triggered earlier in the run which could make it easier for the programmer to find the bug. This result also shows that arithmetic overflow errors often cause other errors, mitigating the need to check for them.

5.4 Comparison to Other Approaches

The same benchmarks and tests were run on different bug detection tools to see if they catch the same bugs. We used the static bug detection tools BOON [89] and Splint [62] and the dynamic tool Valgrind [88] in our comparison. The results are shown in Table 5.2.

BOON did not detect any bugs and problems processing the source code prohibited analysis of *gaim* and *openssh*. Splint reported a large number of errors. The main reason for this is that we did not annotate the source code which would eliminate a large number of the false reports since it is conservative and signals an error if it does not know anything about the buffer size. Due to the large number of error reports, we only manually looked for errors we found with our input checker and error reports that were marked as “likely” errors (a vast majority of the errors were labeled as “possible” errors). It was able to find seven out of the seventeen bugs that we detected. Most of the bugs it missed were due to unconstrained loops and allocations sizes - properties that Splint does not check for. It did miss a couple of array reference bugs, though. It was able to find one bug in *gaim* (one of the “likely” errors) that we did not find. This error occurred in the Yahoo IM protocol and assumed that a packet would have a particular key-value pair. If the key never appeared in

Table 5.2: Comparison to other bug detection approaches.

	BOON		SPLINT		VALGRIND	
	Bugs	Reports	Bugs	Reports	Bugs	Reports
anagram	0	0	2	33	0	0
ft	0	0	0	120	0	0
ks	0	0	2	71	0	0
yacr2	0	0	0	197	0	0
betaftpd	0	0	0	0	1	1
gaim	didn't work		2	2112	0	0
ghttpd	0	0	1	52	1	1
openssh	didn't work		1	473	didn't work	
thttpd	0	0	0	24	0	0

the packet, a string pointer would remain null causing a null pointer dereference later on in the program. This error was not detected with our checker as the error was highly dependent on the control path.

Valgrind found two bugs. Our approach found the bug that Valgrind found in *ghttpd* - the bug that was due to the use of an uninitialized array. The bug in betaftpd was due to the dereference of a pointer that had been previously freed. This was not found in our input checker as it does not check for this type of error. A comparison to the run-time performance of Valgrind is made in the next section.

5.5 Preliminary Performance Results

Our preliminary performance experiments demonstrated that the run-time performance is severely hindered with the addition of the input checker instrumentation. It is important to point out that we have not focused any effort on performance optimization at this point - this is the topic of Chapters 6 and 7.

The results of the performance experiment are shown in Table 5.3. The amount of slowdown experienced is dependent on the program. The four server programs exhibited the

Table 5.3: Preliminary run-time performance results.

	Base line	Our Work		Valgrind	
		Time	Ratio	Time	Ratio
anagram	0.06	3.15	52.50	1.88	31.33
ft	0.18	5.32	29.56	5.92	32.89
ks	0.05	3.96	79.20	4.16	83.20
yacr2	0.12	22.63	188.58	3.83	31.92
betaftpd	0.07	0.53	7.57	0.45	6.43
ghttpd	0.52	1.08	2.08	35.60	68.46
openssh	0.70	1.00	1.43	didn't work	
thttpd	0.15	2.57	17.13	0.29	1.93

Table 5.4: Code size and instrumentation site counts.

	Code Size (KB)			Simple Stmts	Static sites	Dynamic sites
	Orig	New	Ratio			
anagram	18	325	17.94	1,848	538	48,469,011
ft	24	312	12.79	2,881	559	76,221,854
ks	20	319	16.06	2,738	582	58,597,111
yacr2	39	759	19.33	11,891	3,817	300,490,072
betaftpd	38	557	14.83	8,186	2,205	6,320,450
ghttpd	34	410	12.20	4,471	1,256	6,178,897
openssh	1,600	5,412	3.38	97,851	26,858	493,716
thttpd	144	1,157	8.05	23,804	6,362	24,024,093

least amount of slow down with *thttpd* having the most with just over 17x. The four pointer intensive benchmarks suffered significant slow-down from a factor of 30x in *ft* to 186x in *yacr2*. The disparity in the results can be attributed to the fact that the pointer intensive benchmarks have more integer processing than the servers and have a significantly higher number of dynamic instrumentation calls, shown in Table 5.4. For example, *yacr2* has over 600 times as many instrumentation sites as *openssh* but the uninstrumented version of *openssh* takes 6 times longer than *yacr2*.

When comparing the performance results to that of Valgrind, our approach suffers a similar magnitude of slowdown. Most of the benchmarks that had exhibit a low overhead with our approach also saw a low overhead with Valgrind. A similar statement can be made for the benchmarks that incur a large performance penalty. Two exceptions to this trend are

Table 5.5: Preliminary dynamic instruction count performance results. Due to problems beyond our control, we were unable to gather baseline dynamic instruction count metrics for *ghttpd* and *openssh*.

	Baseline	Instrumented	Ratio
anagram	12,736,329	1,078,067,927	84.65
ft	67,037,678	1,812,517,481	27.04
ks	51,091,948	1,309,114,977	25.62
yacr2	185,006,871	9,745,392,260	52.68
betaftpd	1,728,029	78,178,396	45.24
thttpd	16,964,607	454,129,566	26.77

yacr2, where our approach was much slower, and *ghttpd*, where Valgrind was much slower. It is important to note that Valgrind and our input checker look for different types of bugs making a precise performance comparison impossible. Instead, this experiment shows that our input checker runs at a similar speed to other dynamic bug detection tools.

Table 5.4 shows that the effect on code-size is fairly significant. The instrumentation functions consume 241 KB of space and is a noticeable factor in the smaller programs. For larger programs *openssh* and *thttpd*, most of the overhead is due to the added instrumentation calls. Table 5.5 shows the difference in dynamic instruction count for each of the benchmarks. The instruction count significantly increased for all of the benchmarks and averaged 44x. The increase in instruction count did not correlate to a similar increase in time as all of the server programs had much less severe time increases while the integer programs had a larger performance overhead.

The breakdown of the different dynamic instrumentation sites is shown in Table 5.6. *Array state* sites keep track of array information such as dynamic array sizes and state associated with strings. *Array references* sites include a call to an array reference check function when the index has been controlled by input. *Pointer operations* sites are calls to track pointers with their associated array. *Integer state* sites call an associated function to propa-

Table 5.6: Breakdown of dynamic instrumentation calls.

Program	Array State	Array References	Pointer Operations	Integer State	Control Points	String/Input Functions
anagram	1.0%	5.2%	7.8%	68.3%	15.7%	1.9%
ft	0.0%	7.7%	0.0%	59.1%	33.1%	0.0%
ks	0.0%	9.0%	0.0%	56.8%	34.2%	0.0%
yacr2	0.0%	8.8%	1.7%	67.5%	22.0%	0.0%
betaftpd	0.9%	0.0%	4.5%	73.7%	20.9%	0.0%
ghttpd	0.5%	13.1%	0.4%	60.7%	24.8%	0.5%
openssh	2.4%	4.4%	2.5%	67.3%	22.5%	0.9%
thttpd	1.0%	3.2%	10.6%	67.7%	16.9%	0.7%

Table 5.7: Percentage of useless dynamic instrumentation sites. On average, 83% of the instrumentation is useless in that it does not manipulate input-derived data.

anagram	ft	ks	yacr2	betaftpd	ghttpd	openssh	thttpd
83.2%	71.7%	59.6%	76.7%	94.5%	98.0%	94.6%	85.3%

gate and adjust interval constraints and string lengths. *Control points* are calls that narrow interval constraints or maximum string lengths. *String/Input functions* sites include calls to functions that process strings or input.

The results are not surprising - integer operations account for majority of the instrumentation calls. Control points, which contribute to about a quarter of the instrumentation calls, and array references make up the next two largest contributors. Further analysis shows that many of the instrumentation sites in these three classifications are useless. An integer operation and control point is defined to be useless if does manipulate input data or string lengths. Similarly, an array reference is useless if its index is not derived from input data.

The percentage of useless instrumentation sites is displayed in Table 5.7. On average, 83% of the dynamic instrumentation sites are useless as they do no useful work because they do not manipulate input data. Yet, the instrumentation routines will still access the hash table only to find out there is no shadow state associated with the given variables. Another inter-

esting result is that the programs with the fewest number of useless instrumentation exhibited the largest slowdowns (see Table 5.3). This is to be expected because a useful instrumentation site does execute more code than a useless one. Techniques in the next chapter will focus on reducing the instrumentation overhead by eliminating useless instrumentation sites and reducing the time it takes to run the instrumentation functions with better management of shadow state.

5.6 Chapter Summary

Our technique for detecting input-related bounds errors was implemented using MUSE. With our models, we were able to find 17 bugs in 9 programs. Six of the eight defects that were found in our server applications could be exploited by malicious users to gain access to the system, including two known faults in OpenSSH. The number of false alarms was manageable - only six were detected, half of them are due to legal uses of unconstrained input data controlling loops.

Another problem with using dynamic instrumentation is the effect it has on the run-time performance of the program. Our approach saw an average increase of 47x over all benchmarks though the pointer-intensive benchmarks fared much worse than the server applications. In fact, the server applications all suffered a 17x or less increase in execution time. One reason for this severe impact is the large amount of instrumentation executed that is useless (instrumentation that operates on instructions that do not manipulate input-derived data). Removing this instrumentation is the central topic in the next chapter.

CHAPTER 6

EFFICIENT DYNAMIC BUG DETECTION

One of the problems with dynamic instrumentation is the high performance overhead associated with executing the instrumentation. Performance results from Section 5.5 show that a large percentage (83% on average) of instrumentation sites are unneeded, resulting in slowdowns of 47x on average. To address this problem, static analysis is used to determine which variables are derived from input and variables that produce results that could be used in a dangerous operation. Variables do not need to be shadowed if they do not contain program input and if they do not produce a result that eventually will be used in a dangerous operation. Instrumentation can be removed from a statement if the destination does not need to be shadowed.

Other research groups have looked at using static analysis to remove instrumentation. One approach that is used in CCured [73] is to use a static verifier to prove as many dangerous operations safe as possible and then use instrumentation to catch any bugs for operations that cannot be proved safe. In other words, instrumentation is removed for operations that have been proved safe. Their approach complements our work - a static verifier could also be used with our approach to further reduce instrumentation. Bodik *et. al.* [11] explored the elimination of array bounds checks. They use a lightweight static analysis to determine

if array bounds checks are redundant with earlier checks. However, their analysis is designed within in a dynamic compilation environment where our work is implemented using instrumentation that is applied statically. In the future, we plan to look at the implementing this type of optimization within our system but it would require either a static approach to removing array bounds checks or a dynamic approach to eliminating checks if they are redundant.

Our analysis is applied to the entire program using Dflow, described in Section 6.3. Dflow starts by conducting traditional compiler analyses. Reviewing compiler terminology, a *definition* of a variable is a statement that may assign a value to the given variable. If a variable is assigned in different statements, unique definitions will be assigned to each statement. The set *Defs* contains all definitions associated with a particular statement. A similar set, called *Uses*, contains all definitions that could be read or accessed during the statement. These two sets are heavily used in our static analyses described in this chapter.

6.1 Compile-time Identification of Input-Derived State

This section describes our technique for identifying which integers are derived from input and more importantly which variables never contain input at any point in the program. Instrumentation will not be applied to integer operations that do not manipulate input data, control points that do not narrow integer bounds, and array references that do not have an index that was controlled by input.

The analysis is similar to constant propagation, a compiler optimization that replaces variables that are known to be constant. The key difference is that we will be propagating attributes associated with each definition instead of values. Recall that integers can have

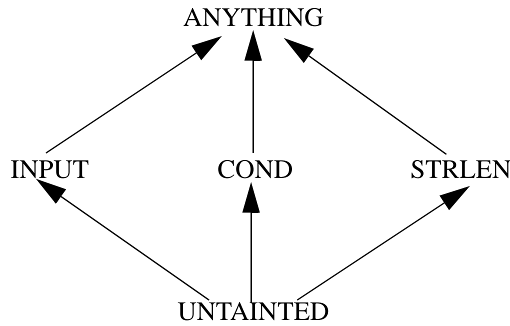

Figure 6.1: Tainted propagation lattice. Definitions start out in the untainted state. If a definition is determined to be tainted, it moves up the lattice based on the shadow state it must contain. If a definition can hold more than one different type of shadow state, it moves up to the anything state.

three different types of shadow state: data derived from input (INPUT), string length data (STRLEN), and conditional data (COND). These three types form the lattice used for tainted propagation shown in Figure 6.1. Definitions that do not directly come from input or a string length operation start in the UNTAINTED state. During the algorithm, definitions may become tainted and move into the appropriate state based on the type of shadow state that is necessary for the definition. If a definition can contain more than one different type of shadow state, it moves to the ANYTHING state at the top of the lattice and remains in that state for the duration of the algorithm. Since each definition can only move upward on the lattice, the algorithm is guaranteed to terminate.

The tainted propagation algorithm is shown in Figure 6.2. The first part of the algorithm sets the initial state for each definition. All definitions that are written from input during input system functions such as `getc` or `scanf` will be in the INPUT state. In addition, all return values from string conversion functions such as `atoi` are also conservatively placed in the INPUT state as our analysis does not track input attributes for strings. The output of `strlen` instructions are considered to be in the STRLEN state. All other definitions are placed in the UNTAINTED state.

```
// Acquire attribute function - adds attribute attr to definition def
acquire_attribute(def, attr) {
    if (attr = UNTAINTED) return
    if (Taint(def) = ANYTHING) return
    if (Taint(def) = UNTAINTED) {
        Taint(def) = attr
        Changed = TRUE
    }
    else if (Taint(def) ≠ attr) {
        Taint(def) = ANYTHING
        Changed = TRUE
    }
}

// Initialization
Changed = TRUE
InputFunctionCalls = stmts that call input-producing functions
StrlenCalls = stmts that call strlen
RelationalExprStmts = stmts that contain relational operator (such as < ) expressions.

// Find all initial tainted variables
 foreach stmt s {
    if (s ∈ InputFunctionCalls)
        foreach (d ∈ Defs(s)) { Taint(d) = INPUT }
    else if (s ∈ StrlenCalls)
        foreach (d ∈ Defs(s)) { Taint(d) = STRLEN }
    else
        foreach (d ∈ Defs(s)) { Taint(d) = UNTAINTED }
}

// Iterate until stable
while (Changed) {
    Changed = FALSE
    foreach stmt s {
        if (s ∉ RelationalExprStmts) {
            foreach (u ∈ Uses(s))
                foreach (d ∈ Defs(s)) acquire_attribute(d, Taint(u))
        }
        if (s ∈ RelationalExprStmts) {
            if (∃u ∈ Uses(s) s.t. Taint(u) ≠ UNTAINTED)
                foreach (d ∈ Defs(s)) acquire_attribute(d, COND)
        }
    }
}

// Assign attributes to variables
foreach variable v
    foreach (d ∈ Defs(v)) acquire_attribute(v, Taint(d))
```

Figure 6.2: Tainted propagation algorithm.

Now, the algorithm proceeds iteratively and processes each statement individually during each iteration. At the heart of the algorithm is the *acquire_attribute* function which is responsible for making transitions in the tainted propagation lattice. If the definition already has the particular attribute, then no change is necessary. All definitions, by default, have the UNTAINTED characteristic and no changes are necessary if the acquired attribute is UNTAINTED. If the definition is in the ANYTHING state, it has all of the possible attributes and no change is needed. If the definition is currently in the UNTAINTED state and the acquired attribute is anything else, the definition will be set to the same state as the attribute. The last part of the *acquire_attribute* function is only executed if the definition is in one of the three middle states (INPUT, COND, or STRLEN) and the attribute is tainted in some fashion (not UNTAINTED). If the definition and the acquired attribute are the same, no change is necessary. If they are different, then the definition is capable of holding at least two different types of shadow state and must move into the ANYTHING state. The *acquire_attribute* function will also set the *Changed* variable to true whenever a change is made. This variable is used to stop the algorithm.

For most statements, all definitions will acquire all of the attributes of its source uses. The includes arithmetic operations and copy operations. For instance, if a variable is assigned to a dereferenced pointer, all attributes associated with the variable are acquired by all definitions the dereferenced pointer can point to. However, statements that contain relational operators are treated differently. All definitions created by relational operator statements will acquire the COND state provided that at least one use contains some type of tainted data. If no use contains tainted data, no updates are made.

The algorithm continues to iterate until no changes are made to any of the definitions. Since MUSE expects a list of attributes associated with each variable (rather than by definition), the final part of the algorithm sets to the attributes for each variable by acquiring all of the attributes associated with all of its definitions. The algorithm is guaranteed to terminate because a definition can only change state twice. In the worst-case, the algorithm will make `2d` iterations through the loop where `d` is the number of definitions. Since the number of definitions is proportional to the number of statements, the worst-case running time of the algorithm is $O(s^2)$ where `s` is the number of statements. In practice, the number of iterations is much smaller than the worst-case.

The list of variables and their attributes is read by MUSE. For the purposes of instrumentation, MUSE considers variables to be tainted if there are in any of the four tainted states (INPUT, STRLEN, COND, and ANYTHING). Due to the large number of combinations, the instrumentation routines do not distinguish between the shadow state associated with the integer assuming tainted integers could hold any type of shadow state. A mode flag within the shadow state tracks the type of shadow state associated with the variable. Though the tainted states are combined for this particular optimization, the separate states are used in the optimization described in the next section.

MUSE will suppress instrumentation for loop statements when the loop condition is untainted. Similarly, no instrumentation will be added for array references when the index is untainted. For arithmetic operations, instrumentation will not be added when the destination is untainted. When the destination is tainted, instrumentation will be added with untainted source operands being considered as compile-time constants. If all source oper-

ands are untainted, the expression collapses to a constant and the instrumentation will remove the destination state from the hash table. Even though instrumentation is not removed in this instance, performance is improved because the remove function only requires one access to the hash table. Previously, three hash table accesses could be made: one for each operand and one for the destination. Arithmetic expressions that have a tainted destination and two tainted operands will still use the original instrumentation routine.

6.2 Identification of Variables Used in Dangerous Operations

Up to this point, we have defined useless instrumentation sites to be cases where variables never hold input data or other forms of shadow state. Instrumentation sites can also be considered useless if the data they are manipulating is never used in a dangerous operation. Even if the instrumentation contains valid input data, it is not worthwhile if the data is never checked. In this section, we present an algorithm for detecting variables that are not directly or indirectly involved in the production of a value that is used within a dangerous operation.

The dangerous operation algorithm is very similar to the attribute propagation algorithm presented in the previous section but with two key differences. First, the algorithm starts with dangerous operations and works backwards propagating attributes from the definitions to uses. Secondly, there are only two attributes: SAFE and DANGER. All definitions start in the SAFE state and once they move into the DANGER state, they remain in that state for the duration of the algorithm.

The dangerous propagation algorithm is presented in Figure 6.3. The initial danger state is

```
// Initialization
Changed = TRUE
DangerousStatements = dangerous statements such as array references

// Find all initial tainted variables
foreach stmt s  {
   if (s ∈ DangerousStatements)
      foreach (u ∈ Uses(s)) { Danger(u) = DANGER }
}

// Iterate until stable
while (Changed) {
   Changed = FALSE
   foreach stmt s  {
      if (∃d ∈ Defs(s) s.t. Danger(d) = DANGER) {
         foreach (u ∈ Uses(s)) {
            if (Danger(u) = SAFE) {
               Danger(u) = DANGER
               Changed = TRUE
            }
         }
      }
   }
}

// Assign attributes to variables
foreach variable v
   if (∃d ∈ Defs(v) s.t. Danger(d) = DANGER) Danger(v) = DANGER else Danger(v) = SAFE
```

Figure 6.3: Danger propagation algorithm.

set to DANGER for uses of dangerous operations. This includes array indices, loop conditions, and arguments to dynamic memory allocation functions. All other definitions are set to SAFE. Each statement is examined individually. If a definition exists that is marked with the DANGER state, then all uses for that statement are updated to be in the DANGER state. As before, the *Changed* flag is set when a change is made to a definition and the algorithm terminates when no change is made during an iteration. Lastly, the state of each variable is set. If any of the definitions is considered dangerous, the variable is marked as dangerous.

From an instrumentation perspective, this algorithm only works with integers that only

hold INPUT (or bounds) data. It does not account for integers that hold STRLEN or COND data. For instance, an integer holding COND data may be used in an `if` statement and is not used anywhere else in the program. Since it is not used in a dangerous operation, it is marked SAFE. The instrumentation still may be necessary as the variable could still control a useful, dangerous integer. As a result, MUSE will use the attributes from the tainted propagation algorithm and the danger propagation algorithm to determine if a variable is considered tainted or untainted. If a variable contains input data but is safe, it will be treated as untainted (like a constant). But variables in the STRLEN, COND, or ANYTHING state will still be considered tainted regardless if they are safe or dangerous. This is summarized in Table 6.1.

It is not necessary to retain instrumentation for all variables in the COND state. If the conditional variable narrows integers that do not need shadowed state, instrumentation is not needed for the conditional variable. The tainted propagation algorithm will blindly mark the variable as a COND since the source operands are in the INPUT state. To eliminate this case, the tainted propagation algorithm needs the output of the danger propagation algorithm. This is easily done by running the danger propagation algorithm before the tainted propagation algorithm. This also requires a change to the *acquire_attribute* function in Figure 6.2. Variables that are marked SAFE cannot acquire the INPUT attribute. This means that safe variables that would only hold shadow state for input data would remain UNTAINTED. Variables that could hold other types of shadow state will still acquire those attributes and be considered tainted in MUSE.

Table 6.1: MUSE handling of integer statically derived attributes. All integers that are untainted remain untainted. All integers in the STRLEN, COND, and ANYTHING states are considered tainted. The danger level is only looked at for INPUT data.

Taintedness	Danger Level	Considered As...
UNTAINTED	SAFE	UNTAINTED
UNTAINTED	DANGER	UNTAINTED
INPUT	SAFE	UNTAINTED
INPUT	DANGER	TAINTED
STRLEN	SAFE	TAINTED
STRLEN	DANGER	TAINTED
COND	SAFE	TAINTED
COND	DANGER	TAINTED
ANYTHING	SAFE	TAINTED
ANYTHING	DANGER	TAINTED

6.3 Dflow: Global Dataflow Analysis Tool

To implement the analyses presented in this chapter, we developed a data flow analysis tool called Dflow. We decided not to use GCC's data flow analysis routines because (a) the routines worked on GCC's low-level intermediate representation instead of on the AST and (b) GCC only works with one file at a time. Dflow works at the AST level and accepts the entire program (all files) as input. To analyze a program in Dflow, a AST file is required for each file. The AST file is created by running GCC in a special mode that only parses the program and dumps the internal AST into a file. The format of this file expresses the AST in the same manner as GCC's internal AST except that irrelevant fields have been removed.

During processing of the AST files, Dflow creates a list of variables. Local variables[1] are given unique names so that separate declarations that use the same variable name will

1. Variables are considered to be local if they are only visible in one file (this includes all static variables). Variables are considered to be global if they could be visible in multiple files.

have different names in Dflow. Structure variables are considered a single entity and are not separated by field. An array will introduce two variables: one for the array and one for the base pointer of the array. A special variable is created for each memory allocation site in the program. They are marked with as having a special heap type indicating the variable could be of any type.

Dflow starts by creating a control flow graph for each function. The functions are divided into basic blocks and the set of predecessors and successors are created are for each basic block. Basic blocks that cannot be reached from the start of the function are removed.

Then, a points-to analysis, implemented using Andersen's algorithm [2], is performed to detect aliases between the variables and to determine what functions can be called when a function pointer is invoked. The algorithm starts with all array base pointers pointing to their arrays and pointers returned from allocation routines pointing to the special allocation variable for that site. Pointer assignments may increase the set of variables a pointer may point to. For example, in the statement '`a = b`', the variable `a` will now point to the same variables that `b` points to. The analysis is flow-insensitive so the variables that `a` could point to before the assignment are still in its points-to set. The algorithm continues to iterate until the points-to set does not change for any variable. In the worst-case, the running time is $O(sv^2)$ or $O(s^3)$ where `s` is the number of statements and `v` is the number of variables which is linearly proportional to the statements. In reality, the algorithm runs much faster, often completely in less than ten iterations.

Once the points-to information has been computed, a call graph is created. The call graph uses the points-to information to determine where function pointers can point to. Since

there is a dependence between the points-to analysis and the creation of the call graph, these two analyses are placed with an iterative loop until neither analysis changes. Typically, only three passes are necessary: the first pass computes intraprocedural points-to information, the second pass computes interprocedural points-to information, and the third pass confirms that steady state has been reached. Additional passes are only necessary if function pointers are passed as parameters to functions that are called using function pointers. Once the final call graph has been constructed, the original control flow graph is altered to represent function calls and returns using edges. This organization makes future analyses context-insensitive.

The next phase of Dflow computes, for each statement, the set of definitions that are created (called *Defs*) and killed. Each variable assignment creates a unique definition. For a typical assignment statement, the definition set contains the lone unique definition for the destination variable. The killed set contains all definitions corresponding to the destination variable except for definitions that appear in the *Defs* set. This information is then summarized at the basic block level.

The next step is to use apply a reaching definitions analysis to determine which definitions are live upon entry to each basic block. The algorithm works by traversing each basic block. The new reaches set of a basic block `b` is determined by:

```
foreach predecessor p of basic block b {
  b.reaches = b.reaches | (p.defs | (p.reaches & ~p.killed))
}
```

The set of reaching definitions is formed by taking the current reaches set and adding definitions from predecessors that reach the current block. Definitions from predecessors can

come from two sources: definitions created in the predecessor block[1] or definitions that reach and were not killed in the predecessor block. This algorithm iterates until steady state is reached. The algorithm could make O(s^2) iterations resulting in a worse case running time of O(s^4)[2]. In practice, there are significantly fewer iterations but enough to make this algorithm the slowest phase of Dflow.

After the set of reaching definitions for each basic block have been computed, each statement is traversed to determine which definitions are used (called *Uses*) or read by the statement. This set contains definitions that reach the statement and correspond to one of the variables read in the statement.

Finally, the tainted propagation and danger propagation algorithms are executed. These algorithms use the *Defs* and *Uses* sets that are computed in previous phases of Dflow. Dflow will dump a list of variables along with a list of attributes associated with that variable. This list is processed by MUSE as described in the next section.

In addition to the identification of input-derived and dangerous variables, Dflow also provides two additional lists of information that are used by MUSE. One list consists of called functions where source code was not present. This list will include any system functions (assuming the source code is not present) as well as any other non-system functions where the source code was not present. This list of functions serves as the list of "system functions" for MUSE as described in Section 4.2.7. The other list consists of all variables that

---

1. `p.defs` only contains definitions that have not been killed at the end of the block.
2. In reality, the running time is O(s^2e) where `e` is the number of edges in the control flow graph. For arbitrary programs, `e` is proportional to s^2 in the worst case. However, for well-structured C programs, `e` is proportional to `s`.

have been used as an operand to the address-of ‘&’ operator. This indicates variables that could be aliased within our improved shadow management scheme (see Chapter 7).

While these analyses are specific to the input checker, Dflow is structured such that it would be a straightforward for a user with some background in compilers to add different analyses when MUSE is used for other purposes.

6.4 Implementation and Validation

The attribute propagation algorithm and dangerous propagation algorithm are implemented within Dflow. Once the analyses have completed, Dflow will output a list of variables and two attributes associated with the variable: taintedness and danger level. The taintedness attribute can either be UNTAINTED, INPUT, COND, STRLEN, or ANYTHING. The danger level is either DANGER or SAFE. MUSE will convert this information into tainted or untainted based on the chart in Table 6.1.

All instrumentation that relies on taintedness will be marked with the inline designation meaning the instrumentation function resides in the compiler. The function will first determine if instrumentation is necessary based on the taintedness of operands. If it necessary, the function will emit a call to the appropriate external instrumentation function. The precise function is dependent on the current operation and the taintedness of the operands.

Since compiler optimizations are implemented to remove unneeded instrumentation, we wanted to make sure that we were not removing any instrumentation that is necessary or altering how the instrumentation performed. To make sure we did not make any implementation errors, we built in some techniques that validated our optimizations.

To validate the tainted propagation algorithm, we tracked the number of dynamic useful instrumentation sites for the both the unoptimized instrumented version and the instrumented version. If they differ, then the optimization is flawed in someway. Additional debugging information within the instrumentation was able to track the number of useful instrumentation calls per site in order to isolate the error. This feature was very helpful in debugging the implementation. Most of the errors were due to complications in the alias analysis portion of Dflow; this phase was difficult to get right due to the different handling of pointers and arrays in the AST and the fact that dynamic memory allocation sites can contain both.

This validation scheme only addressed the cases where instrumentation is removed because the instruction never manipulates input-related data. Our initial definition of useful instrumentation does not take into account operations that never produce a result that is used in a dangerous operation. Such operations also are unneeded and can be removed. Unfortunately, it is very difficult to obtain the number of useful dynamic instrumentation sites with this updated definition. This is due to the fact that when an instrumentation site executes, knowledge about the future is needed - specifically “will the value be directly or indirectly used in a dangerous operation?”. Instead, we relied on testing to validate this algorithm. We insured that the same bugs and false alarms were properly detected when the optimizations were present. Since the validation tainted propagation algorithm stressed all phases of Dflow in addition to the algorithm, we feel confident that the only thing we needed to test was the danger propagation operation. A number of directed tests were focused on different parts of the algorithm (in particular, proper handling of the points-to information).

6.5 Performance Results

In this section, we explore the effect of our two static analyses. The run-time results of this experiment are shown in Figure 6.4. Three experiments were run for each benchmark comparing the run-time performance to the unoptimized instrumented program presented in the previous chapter. The first two experiments (columns) show the performance improvement that results from removing instrumentation from executing each algorithm in isolation. For the first column, instrumentation that never manipulates input derived data is removed. In the second column, instrumentation that never manipulates data that will be used in a dangerous operation is removed. The last column shows the performance improvement when both optimizations are applied in tandem. Results from the same experiment that show the improvement in dynamic instruction count are presented in Figure 6.5. Table 6.2 shows the times and ratios with respect to the baselines for the optimizations.

When removing instrumentation pertaining to input data, performance improved by 30% on average. In general, programs that had the worst unoptimized impact on performance saw the best performance improvement. For instance, *yacr2*, which had a factor of 186x slowdown, saw a performance improvement of 48%. On the other end of the spectrum, *openssh* saw little performance improvement since it suffered very little performance degradation initially. One exception to this rule is *ks* which had little performance improvement despite having a large performance degradation. This can be explained using the data in Table 6.3. There is a larger percentage of COND variables, the slowest type of instrumentation to process, in *ks* then compared to other programs. Fortunately, all but one of these variables marked COND also are safe so they disappear when both optimizations are

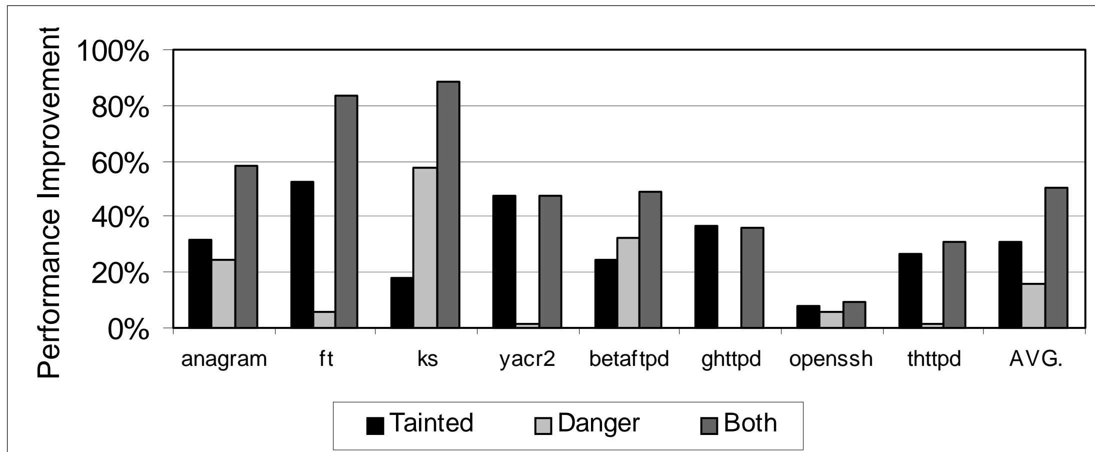

Figure 6.4: Performance improvement from removing unneeded instrumentation.

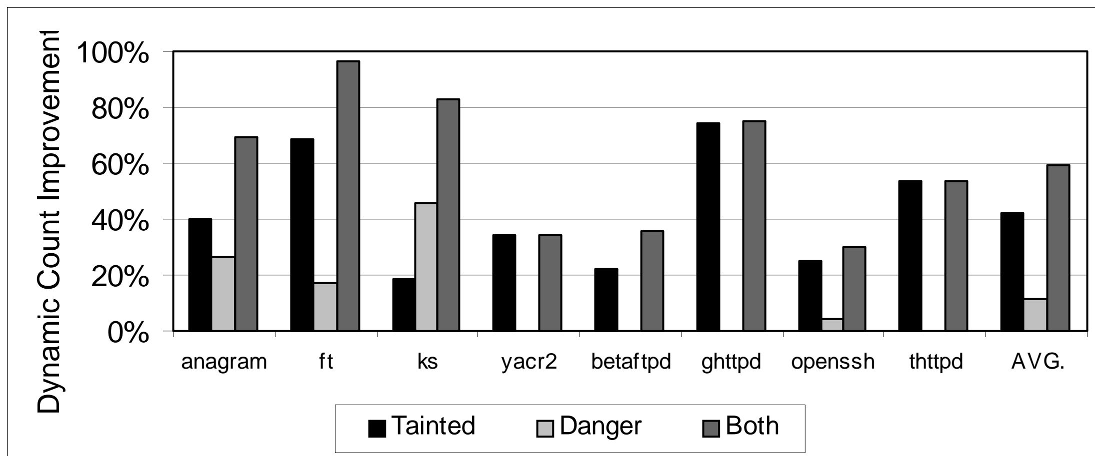

Figure 6.5: Dynamic instruction count improvement from removing unneeded instrumentation.

Table 6.2: Performance comparison from removing unneeded instrumentation.

	Base line	Unoptimized		Input Derived		Dangerous		Both	
		Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
anagram	0.06	3.15	52.50	2.15	35.83	2.37	39.50	1.32	22.00
ft	0.18	5.32	29.56	2.52	14.00	5.03	27.94	0.88	4.89
ks	0.05	3.96	79.20	3.24	64.80	1.69	33.80	0.45	9.00
yacr2	0.12	22.63	188.58	11.82	98.50	22.35	186.25	11.87	98.92
betaftpd	0.07	0.53	7.57	0.40	5.71	0.36	5.14	0.27	3.86
ghttpd	0.52	1.08	2.08	0.68	1.31	1.08	2.08	0.69	1.33
openssh	0.70	1.00	1.43	0.92	1.31	0.94	1.34	0.91	1.30
thttpd	0.15	2.57	17.13	1.88	12.53	2.53	16.87	1.78	11.87

Table 6.3: Breakdown of tainted and dangerous integers.

	anagram	ft	ks	yacr2	betaftpd	ghttpd	openssh	thttpd
Total Integers	257	233	244	1,508	950	420	11,727	2,493
Untainted	210	202	140	855	673	352	8,208	1,609
Tainted - INPUT	39	23	60	0	24	4	402	75
Tainted - STRLEN	0	0	0	0	170	46	162	185
Tainted - COND	8	6	44	145	51	2	1,296	188
Tainted - ANYTHING	0	2	0	508	32	16	1,659	437
Safe	202	213	179	1,163	854	365	8,871	2,106
Dangerous	55	20	65	345	96	55	2,856	387
Untainted and Safe	200	211	178	693	633	314	6,343	1,536
Untainted and Dangerous	47	20	23	162	64	42	2,449	147
Tainted and Safe	2	2	1	470	221	51	2,528	570
Tainted and Dangerous	8	0	42	183	32	13	407	240

applied together.

Removing instrumentation from operations that never produce results that could be used in a dangerous operations was a mixed bag depending on the benchmark. Three benchmarks (*anagram*, *ks*, and *betaftpd*) saw significant performance improvements, two (*ft* and *openssh*) saw small performance improvements, and the remaining three (*yacr2, ghttpd,* and *thttpd*) saw virtually no performance improvement. The results are somewhat surprising in that most of the statements are marked safe (see Table 6.3) meaning that most instrumentation should be removed. However, an inspection of *yacr2* showed that dangerous variables tended to be used heavily in loops whereas safe variables primarily resided outside of loops. This is intuitive since many array references come inside a loop.

Combining the instrumentation removal strategies showed a large increase for programs that benefited from the danger propagation algorithm. Programs that did not benefit from this algorithm did not see much more improvement than using the tainted propagation algorithm alone. One exception to this was *ft*, which saw a large combined performance improvement despite a small performance gain from the danger propagation algorithm.

Table 6.4: Impact on instrumentation sites from removing unneeded instrumentation.

	unoptimized	tainted		danger		both	
anagram	48,469,011	29,279,571	(39.6%)	36,978,977	(23.7%)	13,381,551	(72.4%)
ft	76,221,854	20,373,010	(73.3%)	76,181,787	(0.1%)	242,192	(99.7%)
ks	58,597,111	42,429,376	(27.6%)	46,683,971	(20.3%)	4,393,168	(92.5%)
yacr2	300,490,072	143,471,938	(52.3%)	300,490,072	(0.0%)	143,471,938	(52.3%)
betaftpd	6,320,450	4,469,339	(29.3%)	6,109,908	(3.3%)	3,255,667	(48.5%)
ghttpd	6,178,897	682,551	(89.0%)	6,178,893	(0.0%)	682,547	(89.0%)
openssh	493,716	316,877	(35.8%)	488,460	(1.1%)	298,060	(39.6%)
thttpd	24,024,093	11,425,811	(52.4%)	24,026,707	(0.0%)	11,426,317	(52.4%)

This is due to the fact that not a single integer variable was both tainted and dangerous.

In Table 6.4, the number of instrumentation sites is shown for each of the experiments. The percentage is the reduction in instrumentation sites over the unoptimized case. In general, the reduction in instrumentation sites correlates well to the performance improvements realized in the benchmarks.

Most of the instrumentation that remains is either instrumentation for pointers and arrays (not subject to removal for this optimization) or integers on the heap. While the algorithms track the state of heap data at the allocation site level (which itself is conservative), MUSE is conservative and only removes instrumentation for statements that only operate on local and global variables (including parameters). Since instrumentation is added statically, there may be cases where instrumentation is useless some of the time. However, this was rarely the case. In all but a few cases, an instrumentation site was either useful all the time or useless all the time.

The time to analyze each program using Dflow is shown in Table 6.5. In most cases, the analysis time is minimal. Since the worst-case algorithm within Dflow has a running time of O(s^4) where `s` is the size (number of simplified statements) of the program. As a result,

Table 6.5: Dflow analysis time.

anagram	ft	ks	yacr2	betaftpd	ghttpd	openssh	thttpd
0.03	0.08	0.06	0.55	0.24	0.11	146.68	2.77

larger programs such as *openssh* and *thttpd* take considerably longer to analyze.

6.6 Chapter Summary

In this chapter, we improved the run-time overhead associated with dynamic bug detection by using static analysis. To facilitate whole-program static analyses, we developed Dflow, an interprocedural data-flow analysis tool. Dflow will read in the AST of simplified C generated by GCC for the entire program. Analyses determine which variables were dependent on input using taint propagation and which variables never hold a result that will be used in a dangerous operation. Subsequently, instrumentation was only applied to statements that manipulate variables that needed shadowed state. These optimizations improved performance by 50%.

CHAPTER 7

EFFICIENT MANAGEMENT OF SHADOWED STATE

Many dynamic bug detection systems access shadow state using a shadow state table indexed by address. This can be slow considering the number of accesses tracked by instrumented programs. Other dynamic bug detection systems manage shadowed state by increasing the size of the variable and embedding the extra state with the variable. This reduces access time but creates a compatibility problem when executing functions and system calls that were not instrumented.

We combine these two approaches by managing local variable shadow state by name. For each local variable that needs shadowed state, a temporary variable is created within the compiler that corresponds to the shadow state of the variable. This avoids an access to a shadow state table improving performance and does not modify the original variable, maintaining compatibility. Variables on the heap are accessed by address using a shadow state table.

7.1 Shadowing Local Variables by Name

In our baseline implementation, all variables accessed their shadowed state by looking up their value in a shadow state table implemented using a hash table indexed by address. Most instrumentation calls will typically have two or three accesses to the shadow state

table. One or two table lookups occur to read input shadowed state and a table update is used to write shadowed state to the destination. As a result, most of the time spent in the instrumented code (or the entire instrumented program for that matter) is spent in accessing the shadow state table. With tighter integration into the compiler, it is not necessary to have shadow state table accesses for all variables.

In other dynamic bug detection systems, shadowed state is usually maintained using one of the following two techniques: fat variables or a shadow address space. Fat variables, such as the fat pointers in Safe-C [4], increase the size of the variable and includes the additional state as part of the variable itself. The main advantage of this approach is that is fast to access shadowed state associated with the variable as it is not necessary to lookup the data in a table. There are two main problems with this approach. By increasing the size of all pointers in the program, the performance suffers since a single pointer will no longer fit in a register. The other problem is the complexity associated of insuring functional correctness. Functions that rely on the size of an object must be modified and conversion routines must be implemented when calling a function that does not support fat pointers.

Using a shadow address space, as is done in Purify [45], is straight-forward in that it handles all types of variables including those created on the heap in the same manner. The main drawback is the access time associated with the table access.

Our approach combines these two techniques by shadowing local variables by name and shadowing all other variables in the shadow state table. Instead of storing shadowed state for local variables in the hash table, a temporary variable containing the shadowed state is created by the compiler. An important feature is that the shadowed state is kept in a sepa-

Table 7.1: Shadowed state management method based on type.

Variable Type	Shadowed State Access Method
local (unaliased)	by name
local (aliased)	by address
parameter (unaliased)	by name
parameter (aliased)	by address
global	by address (could be done by name)
heap	by address

rate variable, the local variable remains the same size. As a result, no modifications are required to uninstrumented code to accommodate this change. In addition, this technique can be combined with our other static analyses so that shadow state variables are only created for variables that need them. Our technique obtains the benefits of both existing techniques by making shadow state access fast for local variables while being straightforward to implement as the original variables are not modified.

This technique shifts the burden of accessing shadowed state from run-time to compile-time, reducing execution time. For local variables and parameters, the compiler keeps track of all shadowed state based on the name of the variable. Unfortunately, this is not possible for variables created on the heap since the compiler cannot assign a name to dynamically allocated memory. We use the shadow state table to look up the shadowed state of heap data at run-time using their address. Global variables also could be accessed by name. However, due to the high amount of aliasing and the use of separate compilation units in GCC, global variables also use the shadow state table. This had a negligible impact on performance. Access to shadowed state is summarized in Table 7.1.

One problem with using temporary variables to shadow the state of locals is the possibility of local variables being aliased. Compilers might not be able to detect when a local vari-

able is written if it is accessed via a dereferenced pointer. To address this problem, any integer variable that is used in the address-of '&' operator will be treated like a heap variable and use the shadow state table to store shadowed state.

7.2 Implementation

Implementing the new shadow state management strategy requires two primary changes. The first, is that the compiler must create temporary variables for all integers that are eligible - local variables that need shadow state and are never aliased. This information is obtained from Dflow, see Section 6.3. The state variable is a structure that contains a mode indicating which type of shadow state is currently stored in the shadow state and a union containing the proper shadow state fields. The mode is slightly different in that it must also note when the variable is untainted as the state variable is always present. Due to the large amount of state associated with the COND entry, the union only contains a pointer to a separate structure. This structure is dynamically created if and when conditional data is stored. Actually, the shadow state table entries are stored in this same format to conserve memory and to simplify the instrumentation routines.

The second change is that the instrumentation routines need to be modified to account for the situations where state is shadowed by name. Instrumentation routines that use shadow state created by the compiler will have a pointer to this state as a parameter. The routines will simply use the provided shadow state instead of accessing the hash table. The problem lies in that different operands can have different shadow state access methods. Consider an add instruction where the destination and each source operand are tainted. Each of the three operands could be accessed by name or by address. This results in eight different

instrumentation functions for add alone and does not even account for the additional cases where one or more of the operands are untainted. Due to the high number of instrumentation functions, we developed a code generator that automatically generates the code for all of the binary operations. Other instrumentation functions, such as the parameter passing functions, were modified by hand.

Testing the new scheme for managing shadow state is straightforward. The number of instrumentation calls is the same when compared to the case when all shadow state is accessed by address. The only difference is that different external functions will be called when shadow state is accessed by name. This difference should not affect the underlying functionality of the instrumentation. Therefore, we validated this approach by comparing a trace that prints a message anytime a variable changes state. If the traces differ, then a problem has occurred with the implementation.

7.3 Performance Results

This section looks at the results from experiments involving our new approach to shadow state management will be presented. In addition, the experiments also measure the combined effect of removing useless instrumentation and using the improved shadow state management scheme.

Accessing shadow state for local variables by name improved performance for all programs as shown by the performance graph in Figure 7.1, the dynamic instruction count graph in Figure 7.2, and comparison table in Table 7.2. In the two graphs, the three bars represent (in order from left to right), the performance improvement of applying the instrumentation removal routines, the effect of shadowing state by name, and combining

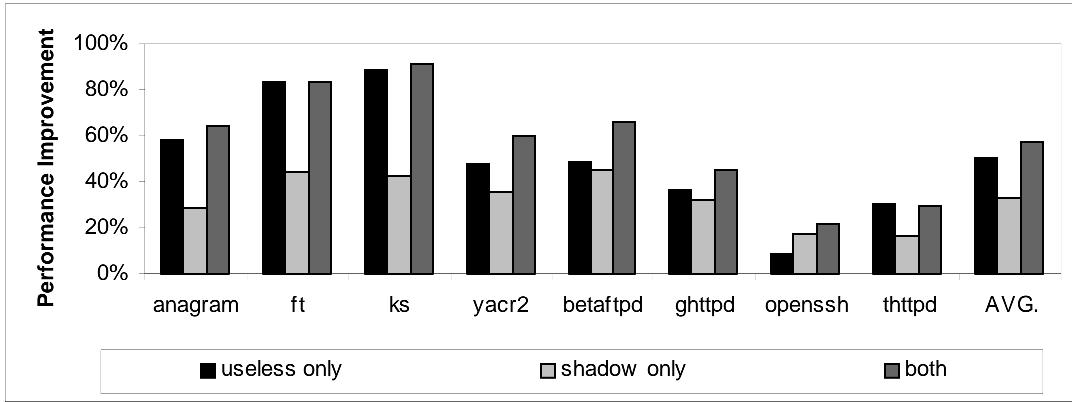

Figure 7.1: Shadow state management performance improvement.

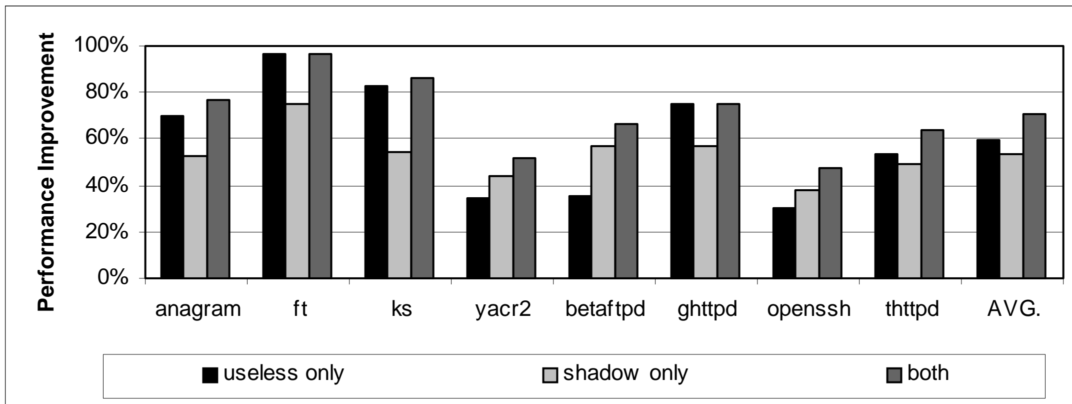

Figure 7.2: Shadow state management dynamic count improvement.

Table 7.2: Shadow state management performance comparison.

	Base line	Unoptimized		Useless Inst. Removed		Shadow State by Name		Both	
		Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
anagram	0.06	3.15	52.50	1.32	22.00	2.24	37.33	1.12	18.67
ft	0.18	5.32	29.56	0.88	4.89	2.95	16.39	0.90	5.00
ks	0.05	3.96	79.20	0.45	9.00	2.28	45.60	0.33	6.60
yacr2	0.12	22.63	188.58	11.87	98.92	14.53	121.08	8.96	74.67
betaftpd	0.07	0.53	7.57	0.27	3.86	0.29	4.14	0.18	2.57
ghttpd	0.52	1.08	2.08	0.69	1.33	0.73	1.40	0.59	1.13
openssh	0.70	1.00	1.43	0.91	1.30	0.83	1.19	0.78	1.11
thttpd	0.15	2.57	17.13	1.78	11.87	2.14	14.27	1.82	12.13

both the instrumentation removal and shadowing state by name.

The average performance improvement for shadowing state by name is 33% and, with the exception of *openssh*, removing useless instrumentation proves to be a better optimization. It is interesting to note that shadowing state by name had a much higher impact on the dynamic instruction count than on performance. This is due to that shadowing state by name only reduces the amount of work within the instrumentation sites; it does not eliminate the calls to the external instrumentation procedures. Unlike useless instrumentation removal, the new approach to shadow state still suffers from expensive function calls and the inability to optimize across the instrumentation calls. This result suggests that further inlining the instrumentation (rather than using function calls) may have good performance benefits.

Combining the two approaches further improves performance by 58% on average and ranged from 22% (*openssh*) to 92% (*ks*). However, the combined impact is not close to the sum of its parts. This not surprising since both techniques target local and global integers in different ways. Variables that take advantage of the new shadow state management likely include many variables that are useless and do not need shadow state. The final column of Table 7.2 shows the degree of slowdown after both optimizations. With the exception of *yacr2*, the benchmarks exhibit slowdowns that are similar to that of other dynamic bug detection tools.

Since the goal of shadowing state of name was to reduce hash table accesses, this metric was measured and is shown in Figure 7.3. This graph shows the percentage of integer shadow state table accesses relative to the unoptimized case. On average, shadowing state

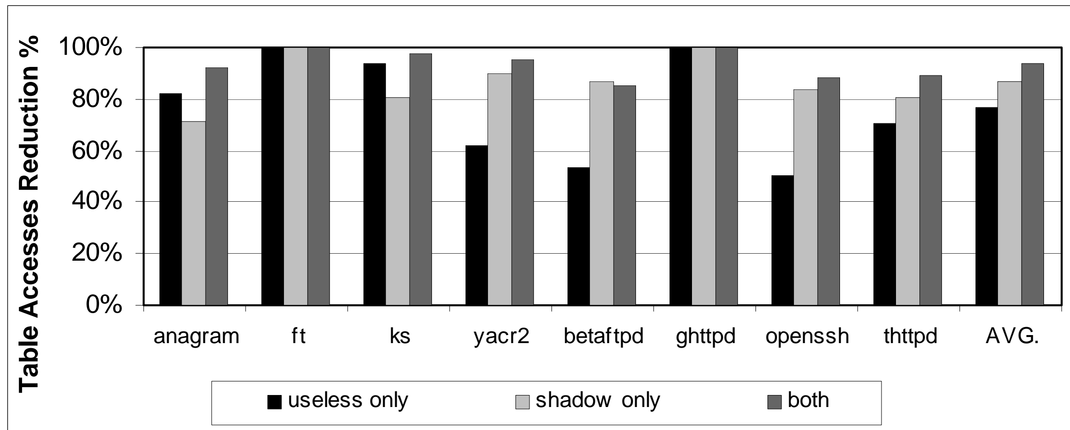

Figure 7.3: Hash table accesses with respect to unoptimized instrumented program.

by name did what is was supposed to do, reducing the amount of accesses by over 85% and the combined effect of both optimizations removed accesses by over 93%. This did not correlate into a similar level of performance improvement since it did not significantly affect the array hash table accesses which dropped slightly (mostly due to useless array references being removed) when both optimizations were applied. For the unoptimized case, there were significantly more integer hash table accesses than array hash table accesses. After both optimizations, the situation is reversed with array hash table accesses outnumbering integer hash table accesses.

7.4 Chapter Summary

This chapter describes our technique for improving the management of shadow state. It eliminates accesses to the shadow state table by shadowing local variables by name. Shadowed state for local variables was kept in temporary variables kept by the compiler. This allows for fast access and maintains compatibility with uninstrumented code since it does not modify the original variable. This optimization reduced the number of shadow table accesses by 85% and improved performance by 33%.

Combining this approach with the useless instrumentation removal optimization resulted in an overall performance improvement of 58% when compared to the unoptimized instrumented program. One downside is these techniques were only applied to local and global integers. Similar techniques can be applied to arrays and pointers to further improve performance.

CHAPTER 8

CONCLUSION

8.1 Summary of the Thesis

There are four main contributions of this thesis: a dynamic approach to detecting input-related software faults, the development of a source code instrumentation infrastructure, a technique for removing unnecessary instrumentation, and a method for improving shadow state management to further improve performance.

Our method for dynamic bug detection looks for software faults caused by improperly bounded program input. Our dynamic approach overcomes many limitations of static analysis while reducing the dependence on the input. Integers and strings are that derived from input are shadowed with additional state representing the bounds (or length in the case of strings) that a variable may hold. At potentially dangerous operations, the entire range of possible values is checked to determine if an error can occur. Since all values are checked, it is not necessary to have the precise input to trigger the error.

We applied our technique to 9 programs and found a total of 17 bugs including 2 security flaws in OpenSSH. The cost of our checker's accuracy is run-time performance, with some programs experiencing more than two orders of magnitude slowdown. As such, in its current form our approach is best targeted to development testing where precision is

more important.

Our instrumentation infrastructure, called MUSE, runs as a part of the GCC C compiler. MUSE is general purpose - a user can create customized instrumentation for a variety of tasks such as bug detection, debugging, profiling, and coverage. Instrumentation is specified using patterns that match against a simplified form of C. Qualifiers further refine the patterns by allowing them to compare against the operands or function name. When a match occurs, a call to an external function is added to the code. Early results show that simplification does not greatly impact the performance but adding instrumentation does due to lost opportunity to optimize across the instrumentation calls.

To reduce the run-time overhead associated with dynamic bug detection, we employed techniques that integrated the checking instrumentation into the compiler allowing static analysis to optimize the instrumentation. Our first approach used static analysis to identify variables that do not need shadow state. Only variables that can contain input derived data and can hold a value that will be used in a dangerous operation need shadow state. Instructions that contain operands that do not need shadow state no longer need instrumentation. This reduced the average number of instrumentation sites by 63% and improved performance by 50%.

To limit the number of accesses to the shadow state table, we shadowed local variables by name by keeping their shadowed state in temporary variables kept by the compiler. This allows for fast access and maintains compatibility with uninstrumented code since it does not modify the original variable. This optimization reduced the number of shadow table accesses by over 85% and improved performance by 33% on average. Combining both

approaches improved performance by 58% over the unoptimized instrumented programs.

8.2 Future Directions

There several avenues for future work. The ideas for future directions have been divided into three main areas: enhancements to MUSE and the instrumentation infrastructure, techniques to improve the quality of dynamic bug detection, and methods for further reducing the impact on performance.

8.2.1 Enhancing the infrastructure

One impediment of our current implementation comes from the fact that two separate tools are used: MUSE and Dflow. In addition, MUSE runs as a phase as GCC, requiring the user to become somewhat familiar with the internals of GCC. A tool that combines both the functionality of MUSE and Dflow would address these drawbacks. The output from such a tool would be an instrumented source code file that would subsequently be used by GCC or any other compiler A primary benefit to this approach is the tighter integration with Dflow and MUSE. Currently, only a list of variables and their derived attributes is passed from Dflow to AST. More sophisticated forms of analysis are possible that can move code and/or instrumentation to improve bug detection and performance. Some of the optimizations described later in this section would need this capability.

MUSE and Dflow can benefit from some changes to increase usability. Some problems are due to complicated hacks so MUSE can get along with GCC. Two areas that can be improved are pointer versus array handling and static variable initialization. The pattern language can be made to more closely match the source language rather than the AST and types of values should be automatically extracted from the program rather than being

specified. The process of inlining code also is unnecessarily complex - another problem that can be mitigated if MUSE were separated from GCC.

8.2.2 Improving dynamic bug detection

Our approach can be refined in order to find new bugs or reduce the frequency of false alarms. One key enhancement would be the addition of symbolic analysis to the relationship between different variables. Currently range analysis is used because it represents a good balance in speed and quality. In order to account for the additional processing required of symbolic analysis, as much of the analysis can be done at compile-time when possible. With knowledge of how the variables relate to each other, false alarms are reduced since a control operation not only narrows the range of the variable involved but all variables that are related to the variable in question. It also can find bugs that we currently do not find. For instance, we are not able to catch all instances where a buffer overflow occurs from concatenating a set of individual strings into one string.

Another common source of buffer overflow errors occurs when strings are processed manually using pointer operations rather than using standard library functions. We are able to catch errors (though we never found any) using our heuristic of guessing a manual string copy has occurred when the last element of an array has been copied. We are also able to find errors when a pointer dereference is out-of-bounds, but is dependent on the precise input. We would like to be able to find errors where pointer dereferences can be out-of-bounds in tests where the pointer accesses are legal but could be illegal if a different input was supplied. Accomplishing this requires analysis of programs where pointer operations are common and to develop heuristics or rules that will catch many of the problems that

can occur with these functions.

Our approach eliminates the dependence on the precise data that is input to the program. However, there is still a dependence on the control path which is directed by input. One idea to reduce the dependence on the input is to move dangerous operation checks to portions of the path that are used in many paths, provided the intervening control operations are independent of the dangerous operation in question. At the expense of false alarms, the movement can be made more aggressive by ignoring the intervening operations when moving the checks.

Before embarking on reducing control dependence, an interesting study would involve looking at where dangerous operations occur in code to see if moving the corresponding check would provide any benefit. Another question is to see what type of level of coverage is necessary for various statements asking questions such as "Would simply executing the statement on any path be enough (statement coverage)?". A very preliminary rudimentary inspection of code showed that many dangerous operations occurred in the main flow within its function (how the function fits into the overall call graph was not looked at) or within loops. Dangerous operation checks that occur in one branch of an if-else rarely could be moved outside of the if-else construct. Continuing this experiment (and making it more scientific) would give valuable insight in how to further tackle the bug detection problem.

Lastly, we would like to implement other checkers in MUSE that look for security-related violations. A possibility is to look at the current permissions of the user and make sure the user does not gain root access in operations that can be exploited.

8.2.3 Reducing instrumentation performance overhead

One way to reduce the instrumentation overhead is to use a static verification system to prove that some of the dangerous operations that occur within a program are safe. Dangerous operations that are proved safe no longer need checking and are no longer considered dangerous. This means that instructions that produce results for the operation may no longer need instrumentation based on the results of the danger propagation algorithm. Besides the performance benefit, combining static with dynamic bug detection will find more bugs than just dynamic bug detection alone. In the future, I personally believe that best bug detection schemes will combine both static and dynamic approaches, allowing static bug detection to find as many bugs (or prove as much about the code) as possible and relying on dynamic detection when static analysis fails.

Removing instrumentation can be difficult at compile-time due to the conservative nature of static analysis. It may be possible to remove instrumentation that is not needed dynamically using approaches similar to those that remove array-bounds checks [11]. Loops are the biggest target as removing instrumentation inside loops will result in a larger benefit. Applying compiler analyses in the style of detecting loop invariants can also be helpful to finding instrumentation to remove though a brief manual inspection of code found little instrumentation that could be moved in this manner.

APPENDICES

APPENDIX A

ELEMENTAL C

The *Elemental C* language is the output of the simplification process. Simplifying the program significantly reduces the complexity of the pattern matching and instrumentation phase. This section presents the grammar of this language. It is based from a development branch of GCC [40] and Hendren *et. al.* [47].

A.1 Statements

```
stmt
: compstmt
| expr ';'
| IF '(' val ')' stmt ELSE stmt
| WHILE '(' val ')' stmt
| FOR '(' val, ')' stmt
| DO stmt WHILE '(' val ')'
| SWITCH '(' val ')' stmt
| LABEL_STMT ':'
| GOTO_STMT ';'
| ASM_STMT ';'
| CASE_LABEL ':'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN val ';'
| decl_stmt ';'

compstmt
: '{' stmtlist '}'
| '{' '}'

stmtlist
: stmtlist stmt
```

```
| stmt
```

Notes:

- Control constructs are simplified by having a test condition of a single variable.
- Scopes have been added (using '{' and '}') for loop bodies, then clauses, else clauses, and each case of a switch statement. This allows additional statements and declarations to be added at any point in the program.

A.2 Declarations

- Same as C but declaration statements do not contain initializers. Initializers are created as separated statements after the declaration.
- Only variable declarations are processed during the simplification process. Type and function declarations are ignored. The compiler provides adequate information for functions and user-defined types.
- Elemental C allows declarations to be mixed with statements. The back-end of the GCC compiler permits this.

A.3 Expressions

```
expr
: call_expr
| modify_expr

modify_expr
: lhs '=' rhs

lhs
: id
| indirect_ref
| array_ref
| component_ref
```

```
rhs
: id
| const
| indirect_ref
| array_ref
| component_ref
| unary_expr
| binary_expr
| cast_expr
| ‘&’ addr_expr_arg
| call_expr
| STRING_CST
| VA_ARG_EXPR

indirect_ref
: '*' id

array_ref
: id '[' val ']'
| '*' id '[' val ']'

component_ref
: id '.' id
| '*' id '.' id

unary_expr
: unary_op val

binary_expr
: val binary_op val

cast_expr
: cast_op malloc_call
| cast_op val

addr_expr_arg
: STRING_CST
| id
| call_expr

call_expr
: function_id '(' arglist ')'
| function_id '(' ')'

malloc_call
: malloc_op '(' arglist ')'
```

```
arglist
: arglist ',' val
| val

val
: id
| const

id
: VAR_DECL
| PARM_DECL
| FIELD_DECL
| LABEL_DECL
| FUNCTION_DECL

function_id
: id
| '&' FUNCTION_DECL

const
: INTEGER_CST
| REAL_CST
| COMPLEX_CST
| LABEL_DECL
| RESULT_DECL
```

Notes:

- For performance reasons, structure and array copying are avoided when possible. Pointers to the objects are used instead.

- Structure assignments (such as `a=b` when `a` and `b` are of the same structure type) are decomposed into a series of assignments, one for each field. This includes initialization of structures using a comma-separated list surrounded by braces. However, structures are copied when they are returned from a function.

- Expressions that do nothing (such as '`a+b;`') are eliminated during the simplification process. In effect, expressions will either be assignments and/or call expressions.

- Calls to dynamic memory allocations such as `malloc` that are immediately preceded by a cast operation are kept as an atomic operation so the type information associated with the allocation sticks together.

A.4 Operators

```
unary_op
: UNOP               /* set of C unary operators */

binary_op
: BINOP              /* set of C binary operators */
| RELOP              /* set of C relational operators */
| TRUTH_AND_EXPR
| TRUTH_OR_EXPR
| TRUTH_XOR_EXPR

cast_op
: NOP_EXPR
| CONVERT_EXPR
| FIX_TRUNC_EXPR
| FIX_CEIL_EXPR
| FIX_FLOOR_EXPR
| FIX_ROUND_EXPR

malloc_op
: MALLOC
| CALLOC
| REALLOC
```

Notes:

- The following operators are converted into a functionally equivalent expression: ‘`,`’ , ‘`++`’, ‘`--`’, ‘`->`’, ‘`?:`’.

- The short-circuited operators ‘`&&`’ and ‘`||`’ are converted into if-then-else statements.

APPENDIX B

SPECIFICATION LANGUAGE GRAMMAR

The grammar for the specification language of MUSE is given below. Non terminals that merely expand into a set of keyword tokens are presented as tables with a description of each keyword. The grammar also uses two terminal tokens: `t_STRING` and `t_INTEGER` that represent arbitrary strings and integers respectively.

```
/* StmtList: A list of pattern statements. */
StmtList:
  StmtList Stmt | Stmt

/* Stmt: A statement consists of a pattern and either a before
 * function, an after function, or both. */
Stmt:
  OptionalDefault OptionalIgnGbl Pattern ':' BeforeFunc AfterFunc
| OptionalDefault OptionalIgnGbl Pattern ':' BeforeFunc
| OptionalDefault OptionalIgnGbl Pattern ':' AfterFunc
| `$ignore' t_STRING ';'

/* OptionalDefault: If a pattern is marked with default, it is only
 * executed if no other pattern matches for that identifier. */
OptionalDefault: `$default' | /* nothing */ ;

/* OptionalIgnGbl: If a pattern is marked with ignore_gbl, it is not
 * executed when processing global variables. */
OptionalIgnGbl: `$ignore_gbl' | /* nothing */ ;

/* BeforeFunc: Function that is called before the matched statement. */
BeforeFunc:
  `before:' t_STRING '(' ParmList ')' ';'
| `before:' t_STRING '(' ')' ';'
| `inline before:' t_STRING '(' ')' ';'
```

```
/* AfterFunc: Function that is called after the matched statement. */
AfterFunc:
  ‘after:’ t_STRING '(' ParmList ')' ';'
| ‘after:’ t_STRING '(' ')' ';'
| ‘inline after:’ t_STRING '(' ')' ';'

/* ParmList: A list of parameters to pass to the function. Parms can
 * be one of several macros. */
ParmList:
  ParmList ',' Parm | Parm

/* Parm: Refers to a parameter macro. Macros are divided into groups
 * based on the arguments (if any) they require. */
Parm
: NormalParm
| TypedParm '(' TypeCode ')'
| IntParm '(' t_INTEGER ')'
| TypedIntParm '(' t_INTEGER ',' TypeCode ')'
| t_INTEGER
| '"' t_STRING '"'

/* Pattern: Toplevel pattern rule */
Pattern:
: ZeroOpStmtCode
| OneOpStmtCode PatternOp
| FuncOpExprCode PatternIdent PatternOp
| FuncExprCode PatternExprType PatternIdent
| ZeroOpExprCode
| OneOpExprCode PatternExprType PatternOp
| TwoOpExprCode PatternExprType PatternOp PatternOp
| LhsExpr
| ’$assign’ LhsExpr FuncExprCode PatternExprType PatternIdent
| ’$assign’ LhsExpr ZeroOpExprCode
| ’$assign’ LhsExpr OneOpExprCode PatternExprType PatternOp
| ’$assign’ LhsExpr TwoOpExprCode PatternExprType PatternOp PatternOp
| ’$assign’ LhsExpr ’$rhs_simple’ PatternOp
| ’$assign’ LhsExpr ’$integer_cst’ t_INTEGER

/* LhsExpr: left hand side expression */
LhsExpr:
  LhsOneOpExprCode PatternExprType PatternOp
| LhsTwoOpExprCode PatternExprType PatternOp PatternOp

/* PatternOp: An operand consists of a type and a name. Either or
 * both could be a wildcard. */
PatternOp: PatternType PatternIdent

/* PatternIdent: An identifier */
PatternIdent: BaseIdent ExceptIdent | BaseIdent

/* ExceptIdent: Exceptions to the base identifier */
ExceptIdent: '$except' '(' ExceptIdentList ')'
```

```
/* ExceptIdentList: List of exceptions */
ExceptIdentList: ExceptIdentList ',' BaseIdent | BaseIdent

/* BaseIdent: Can be an identifier, constant, or wildcard. */
BaseIdent
: t_STRING | '$const_val' '(' t_INTEGER ')' | WildCardIdent

/* PatternExprType: Specifies the type of the expression (optional). */
PatternExprType: '(' PatternType ')' | /* empty *

/* PatternType: Matches a type. */
PatternType: BaseType ExceptType | BaseType

/* ExceptType: Types to be excluded. */
ExceptType: '$except' '(' ExceptTypeList ')'

/* ExceptTypeList: List of excluded types. */
ExceptTypeList: ExceptTypeList ',' BaseType | BaseType

/* BaseType: A type consists of a code, an optional name (useful
 * for complex types), and any number of '*' to represent pointer
 * levels. */
BaseType
: TypeCode '(' t_STRING ')' PtrStar
| TypeCode PtrStar
| TypeCode '(' t_STRING ')'
| TypeCode

/* PtrStar: Used to count pointer levels. */
PtrStar: '*' PtrStar | '*'
```

Table B.1: ZeroOpStmtCode: statements that take no operands.

'$expr_stmt'	Statement that is an expression.
'$label_stmt'	Label definition.
'$goto_stmt'	Goto statement.
'$asm_stmt'	Directive to inline assembly code. The assembly code is not analyzed.
'$continue_stmt'	Continue statement.
'$break_stmt'	Break statement.
'$va_start_stmt'	Special statement specific to GCC AST.
'$begin_scope'	When a new scope is created '{'.
'$end_scope'	When a scope ends '}'.
'$any_stmt'	Matches any statement (including the one operand statements below).

Table B.2: OneOpStmtCode: statements that take one operand.

'$if_stmt'	If statement.
'$while_stmt'	While statement. Matches before the statement and at the end of each iteration of the loop (before the condition).
'$do_stmt'	Do-while statement. Matches just before the testing condition at the end of the loop.
'$for_stmt'	For statement. Matches before the statement and at the end of each iteration of the loop (before the condition).
'$switch_stmt'	Switch statement.

Table B.3: FuncExprCode: expressions that take a function name as a parameter.

'$begin_fn'	The start of the function.
'$end_fn'	The end of a function.
'$call_expr'	A call to a function that contains a fixed number of arguments and is not a system function.
'$sys_call_expr'	A call to function that is a system function or a function where the source code is not present.
'$va_call_expr'	A call to a function takes a variable number of arguments.

Table B.4: FuncOpExprCode: expressions that take a function name and an operand.

'$call_expr_arg'	Argument for a function call where called function takes a fixed number of arguments and is not a system function. Invoked once for each argument.
'$sys_call_expr_arg'	Argument for a function call where called function is a system function or a function where the source code is not present. Invoked once for each argument.
'$va_call_expr_arg'	Argument for a function call where called function takes a variable number of arguments. Invoked once for each argument.
'$return_stmt'	Matches a return statement that contains one operand. Does not match a return statement that returns nothing.
'$birth_parm'	Matches when a parameter is created at the beginning of a function. Invoked once for each parameter.
'$death_parm'	Matches when a parameter goes out of scope at the end of a function or just before a return statement.
'$call_field'	A field within a structure is returned. Invoked after control has transferred back to the caller function.
'$return_field'	A field within a structure is returned. Invoked just before the return statement in the callee function.

Table B.5: ZeroOpExprCode: zero operand expressions.

'$malloc_expr'	A call to `malloc` that is immediately preceded by a cast.
'$calloc_expr'	A call to `calloc` that is immediately preceded by a cast.
'$realloc_expr'	A call to `realloc` that is immediately preceded by a cast.
'$string_cst'	A string constant that occurs in the code.
'$init_string_cst'	A string constant that occurs in the global variable initializers outside of any function.

Table B.6: OneOpExprCode: single operand expressions.

'$rval'	Any time a variable is read.
'$birth_local'	A local variable is created on the program stack.
'$birth_heap'	A variable is created on the heap.
'$birth_global'	A global variable is created at the beginning of the program.
'$death_local'	A local variable goes out of scope.
'$negate_expr'	Negation operator.
'$abs_expr'	Absolute value operator.
'$bit_not_expr'	Bitwise not operator.
'$sizeof_expr'	Size of operator.
'$any_unary_op'	Matches any unary arithmetic or logical operation (the four previous identifiers).
'$nop_expr'	Casting operator (specific GCC AST expression).
'$convert_expr'	Casting operator (specific GCC AST expression).
'$fix_trunc_expr'	Casting operator (specific GCC AST expression).
'$fix_ceil_expr'	Casting operator (specific GCC AST expression).
'$fix_floor_expr'	Casting operator (specific GCC AST expression).
'$fix_round_expr'	Casting operator (specific GCC AST expression).
'$any_cast_op'	Any of the above casting operations.
'$indirect_ref'	Dereference operation (*a) that appears on the right hand side.
'$addr_expr'	Address-of operator.
'$va_arg_expr'	Variable argument expression (specific GCC AST expression).

Table B.7: TwoOpExprCode: binary operand expressions.

'$plus_expr'	Addition operation.
'$minus_expr'	Subtraction operation.
'$mult_expr'	Multiplication operation.
'$trunc_div_expr'	Division operation (rounding mode is truncate).
'$ceil_div_expr'	Division operation (rounding mode is ceiling).
'$floor_div_expr'	Division operation (rounding mode is floor).
'$round_div_expr'	Division operation (rounding mode is round to zero).
'$trunc_mod_expr'	Modulus operation (rounding mode is truncate).
'$ceil_mod_expr'	Modulus operation (rounding mode is ceiling).
'$floor_mod_expr'	Modulus operation (rounding mode is floor).
'$round_mod_expr'	Modulus operation (rounding mode is round to zero).
'$min_expr'	Minimum operation.
'$max_expr'	Maximum operation.
'$lshift_expr'	Left shift operation.
'$rshift_expr'	Right shift operation.
'$lrotate_expr'	Rotate left operation.
'$rrotate_expr'	Rotate right operation.
'$bit_ior_expr'	Bitwise or operation.
'$bit_xor_expr'	Bitwise exclusive-or operation.
'$bit_and_expr'	Bitwise and operation.
'$lt_expr'	Less than operator.
'$le_expr'	Less than or equal to operator.

Table B.7: TwoOpExprCode: binary operand expressions.

`'$gt_expr'`	Greater than operator.
`'$ge_expr'`	Greater than or equal to operator.
`'$eq_expr'`	Equality test operator.
`'$ne_expr'`	Not equal operator.
`'$truth_and_expr'`	Logical and operator (may not be possible in *Elemental C*).
`'$truth_or_expr'`	Logical or operator (may not be possible in *Elemental C*).
`'$truth_xor_expr'`	Logical exclusive-or operator (may not be possible in *Elemental C*).
`'$any_binary_op'`	Any arithmetic, logical, or relational binary operator.
`'$component_ref'`	Component selection '`a.b`'.
`'$ind_component_ref'`	Component selection with a dereference '`(*a).b`'.
`'$bitfield_ref'`	Component selection for a bitfield.
`'$ind_bitfield_ref'`	Component selection for a bitfield via a dereference.
`'$array_ref'`	Array reference '`a[b]`'.
`'$ind_array_ref'`	Array reference with a dereference '`(*a)[b]`'.

Table B.8: LhsOneOpExpr: LHS single operand expressions.

`'$lval'`	Anytime something is written to.
`'$lhs_simple'`	Matches when the left hand side of an assignment is a lone variable.
`'$lhs_indirect_ref'`	Matches when the left hand side of an assignment is a dereference operation (`*a`).

Table B.9: LhsTwoOpExprCode: LHS binary operand expressions.

`'$lhs_component_ref'`	Component selection '`a.b`' on the left hand side of assignment.
`'$lhs_ind_component_ref'`	Component selection with a dereference '`(*a).b`' on the left hand side of assignment.
`'$lhs_bitfield_ref'`	Component selection for a bitfield on the left hand side of assignment.
`'$lhs_ind_bitfield_ref'`	Component selection for a bitfield via a dereference on the left hand side of assignment.
`'$lhs_array_ref'`	Array reference '`a[b]`' on the left hand side of assignment.
`'$lhs_ind_array_ref'`	Array reference with a dereference '`(*a)[b]`' on the left hand side of assignment.

Table B.10: TypeCode: type identifiers.

`'$void_type'`	Void type.
`'$integer_type'`	All integer types.
`'$real_type'`	All floating point types.
`'$enumeral_type'`	Enumeration types.
`'$pointer_type'`	All pointer types.
`'$array_type'`	Array type.
`'$record_type'`	Structure type.
`'$union_type'`	Union type.
`'$function_type'`	Function type.
`'$bitfield_type'`	Bitfield type.
`'$any_type'`	Wildcard to represent any type.

Table B.11: WildCardIdent: wildcards to represent classes of variables.

'$any_value'	Any variable or constant.
'$any_var'	Any variable.
'$any_const'	Any constant.
'$any_fn'	Any function or function pointer.
'$any_fn_ptr'	Function call must be a function pointer, only useful for function calls.
'$any_fn_const'	Any function but does not match a function pointer.

Table B.12: NormalParm: parameter macros that take no additional arguments.

'$file_name'	Name of the current file.
'$line_num'	Current line number.
'$smpl_line_num'	Current simple statement number
'$id'	Identifier that will be unique each time this parameter is used.
'$tree_code'	Code that represents the current operation.
'$elt_offset'	Offset between the given function argument or field and the first function argument.[1]
'$fn_name'	Name of function for patterns that use a function name qualifier (such as `$call_expr`).
'$expr_type'	Type of the expression.
'$expr_size'	Size of the type of expression.
'$expr_obj_size'	Size of object that is being pointed to by the result of an expression.
'$expr_lb'	Lowest possible value, based strictly on the expression type.
'$expr_ub'	Highest possible value, based strictly on the expression type.

1. This is used instead of passing the argument number as fields of a structure will have different offsets but the same argument number.

Table B.13: IntParm: parameter macros that take an integer argument .

'$fn_arg_name'	Name of argument.
'$fn_arg_type'	Type of argument.
'$fn_arg_size'	Size of argument.
'$fn_arg_obj_size'	Size of object that is being pointed to by a pointer argument.
'$fn_arg_addr'	Address of the argument.
'$fn_arg_lb'	Lowest possible value, based strictly on the argument type.
'$fn_arg_ub'	Highest possible value, based strictly on the argument type.
'$op_name'	Name of the operand.
'$op_type'	Type of the operand.
'$op_size'	Size of the operand.
'$op_obj_size'	Size of object that is being pointed to by a pointer operand.
'$op_addr'	Address of the operand.
'$op_lb'	Lowest possible value, based strictly on the operand type.
'$op_ub'	Lowest possible value, based strictly on the operand type.

Table B.14: TypedParm: parameter macros that take a type argument.

`'$expr_value'`	Value of the expression.

Table B.15: TypedIntParm: parameter macros that take a type and integer arguments.

`'$fn_arg_value'`	Value of the argument.
`'$op_value'`	Value of the operand.

APPENDIX C

INPUT CHECKER PATTERN FILE ANNOTATED

This appendix displays the pattern file used for the input checker. Before each set of related patterns, a brief description describing what the patterns match and a summary of what actions are taken within the instrumented model.

These patterns capture the beginning and the end of the program. At the beginning, the checker is initialized by creating the hash tables that store the shadow state. In order to get a better understanding of the instrumentation and the program, several statistics are tracked during the program and printed out when the program exits.

```
$begin_fn main:       after:  init_checker();
$end_fn main:         before: print_stats();
$sys_call_expr exit:  before: print_stats();
$sys_call_expr fatal: before: print_stats();
```

These patterns are used to propagate state for integers when passing parameters into a function and when returning from a function. The first two patterns occur before the function is called and stores the shadow state in a list that is indexed by offset assuming the parameters are laid out in sequential order in memory. The offset is used as it allows for integers within structures to be treated in the same manner as normal integer arguments. The `$birth_parm` pattern is called for each integer parameter when the called function

starts. The shadow state is obtained by looking up its state from the table. The `$return_field` and `$call_field` patterns are used to propagate state for integers within structures that are returned and utilize the same routines as passing parameters into functions. The last three patterns are used to propagate state when the return value is an integer.

```
$call_expr_arg $any_fn $integer_type $any_value:
inline before: store_int_fn_arg();

$va_call_expr_arg $any_fn $integer_type $any_value:
inline before: store_int_fn_arg();

$birth_parm $any_fn $except(main) $integer_type $any_var:
inline after:propagate_int_fn_arg();

$return_field $any_fn $integer_type $any_value:
inline before:store_int_fn_arg();

$call_field $any_fn $integer_type $any_var:
inline after:propagate_int_fn_arg();

$return_stmt $any_fn $except(main) $integer_type $any_value:
inline before: store_int_return_value();

$assign $lval $integer_type $any_var
$call_expr $any_fn:
inline after:propagate_int_return_value();

$assign $lval $integer_type $any_var
$va_call_expr $any_fn:
inline after:propagate_int_return_value();
```

These patterns refer to cases where integers are being copied from one memory location to another. In these cases, the source shadow state is accessed and copied into the destination shadow state.

```
$assign $lval $integer_type $any_var
        $integer_type $any_value:
```

```
inline before:process_int_copy();

$assign $lval $integer_type $any_var
        $any_cast_op $integer_type $any_value:
inline before: process_int_copy();

$assign $lval $integer_type $any_var
$component_ref $any_type $any_var $any_type $any_var:
inline before:process_int_copy();

$assign $lval $integer_type $any_var
$array_ref $any_type $any_var $any_type $any_value:
inline before:process_int_copy();

$assign $lval $integer_type $any_var
$indirect_ref $any_type $any_var:
inline before:process_int_copy();

$assign $lval $integer_type $any_var
$ind_component_ref $any_type $any_var $any_type $any_var:
inline before:process_int_copy();

$assign $lval $integer_type $any_var
$ind_array_ref $any_type $any_var $any_type $any_value:
inline before:process_int_copy();

$assign $lval $integer_type $any_var
$va_arg_expr $any_type $any_value:
inline before:process_int_copy();
```

These two patterns are used for the processing of integer operations. The wildcards are used for the patterns but the external instrumentation functions are separate for the operations (though classes of similar operations are grouped together). The internal instrumentation function within GCC will look at the current operation and make the appropriate instrumentation call.

```
$assign $lval $integer_type $any_var
        $any_unary_op $integer_type $any_value:
inline before: process_int_unary_op();

$assign $lval $integer_type $any_var
        $any_binary_op $integer_type $any_value $integer_type
$any_value:
```

```
inline before: process_int_binary_op();
```

These patterns indicate situations where shadow state should be unconditionally removed from the hash table. With the exception of the last pattern, these are cases when integer data is derived from other non-integer types. The `$ignore_gbl` designation suppresses these patterns from firing during the global initialization phase since it is unnecessary to remove anything at the beginning of the program since no variables have any shadow state to remove. The last pattern refers to system calls (or any call to a function where source code is not present) that do not have a specific pattern devoted to them.

```
$ignore_gbl $assign $lval $integer_type $any_var
        $any_cast_op $any_type $except($integer_type) $any_value:
inline before: remove_int_state();

$ignore_gbl $assign $lval $integer_type $any_var
 $any_unary_op $any_type $except($integer_type) $any_value:
inline before:remove_int_state();

$ignore_gbl $assign $lval $integer_type $any_var
 $any_binary_op $any_type $except($integer_type) $any_value
$integer_type $any_value:
inline before:remove_int_state();

$ignore_gbl $assign $lval $integer_type $any_var
 $any_binary_op $integer_type $any_value $any_type
$except($integer_type) $any_value:
inline before:remove_int_state();

$ignore_gbl $assign $lval $integer_type $any_var
 $any_binary_op $any_type $except($integer_type) $any_value
               $any_type $except($integer_type) $any_value:
inline before:remove_int_state();

$default $assign $lval $integer_type $any_var
 $sys_call_expr $any_fn:
inline after:remove_int_state();
```

These statements refer to the various control constructs. In each case, the run-time variable (only a single conditional variable is allowed in simplified form) controlling the loop

is analyzed. The appropriate bounds are narrowed based on whether the value is true or false.

```
$if_stmt $integer_type $any_var:
inline before:process_if_stmt();

$while_stmt $integer_type $any_var:
inline before:process_loop_stmt();

$for_stmt $integer_type $any_var:
inline before:process_loop_stmt();

$do_stmt $integer_type $any_var:
inline before:process_loop_stmt();
```

These patterns mark the creation of an array. The `$birth_heap` directive refers to cases where arrays are created within an allocation such as an array within a dynamically allocated structure[1]. There are separate patterns for dynamic memory allocation routines such as `malloc` listed later in the file. The last three patterns refer to string constants that appear in the program as they also reside in the array hash table.

```
$birth_local $array_type $any_var:
after:process_array_birth($file_name, $line_num, $smpl_line_num,
$op_name($src0),
$op_addr($src0),
$op_size($src0),
0);

$birth_global $array_type $any_var:
after:process_array_birth($file_name, $line_num, $smpl_line_num,
$op_name($src0),
$op_addr($src0),
$op_size($src0),
1);

$birth_heap $array_type $any_var:
after:process_array_birth($file_name, $line_num, $smpl_line_num,
$op_name($src0),
```

1. In order for MUSE to detect this, the call to `malloc` or `calloc` must be immediately casted to the appropriate type.

```
$op_addr($src0),
$op_size($src0),
0);

$assign $lval $pointer_type $any_var
$string_cst:
after:process_string_cst($file_name, $line_num, $smpl_line_num,
$op_name($dest),
$op_value($dest, $pointer_type));

$init_string_cst:
after:process_string_cst($file_name, $line_num, $smpl_line_num,
$op_name($src0),
$op_value($src0, $pointer_type));

$assign $lval $array_type $any_var
$string_cst:
after:process_string_assign($file_name, $line_num,
$smpl_line_num,
$op_name($dest),
$op_value($dest, $pointer_type));
```

These two patterns refer to the two situations where arrays can be deleted: free system call and when locally declared arrays go out of scope. In either case, the array is removed from the hash table.

```
$sys_call_expr free:
before:remove_array_state($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$death_local $array_type $any_var:
before:remove_array_state($file_name, $line_num, $smpl_line_num,
$op_addr($src0));
```

These patterns correspond to cases where arrays are copied. The only two places where this can occur is when an array is within a structure that is passed as a parameter and when an array is within a structure that serves as the return value of a function. These functions behave similarly to the integer parameter passing functions.

```
$call_expr_arg $any_fn $array_type $any_var:
```

```
before:store_array_fn_arg($file_name, $line_num, $smpl_line_num,
$op_addr($src0),
$elt_offset);

$va_call_expr_arg $any_fn $array_type $any_var:
before:store_array_fn_arg($file_name, $line_num, $smpl_line_num,
$op_addr($src0),
$elt_offset);

$birth_parm $any_fn $except(main) $array_type $any_var:
after:propagate_array_fn_arg($file_name, $line_num,
$smpl_line_num,
$op_name($src0),
$op_addr($src0),
$elt_offset);

$return_field $any_fn $array_type $any_var:
before:store_array_fn_arg($file_name, $line_num, $smpl_line_num,
$op_addr($src0),
$elt_offset);

$call_field $any_fn $array_type $any_var:
after:propagate_array_fn_arg($file_name, $line_num,
$smpl_line_num,
$op_addr($src0),
$elt_offset);
```

The following patterns are for operations involving pointers. Due to the limited number of operations allowed on pointers, all unary operations are accounted for in one external function and all binary operations are accounted for in two external functions. The difference between the two binary operator functions is that one works the situation where one operand is a variable and the other is a constant. The other function works on instances where both operands are variables.

```
$assign $lval $pointer_type $any_var
        $any_unary_op $pointer_type $any_var:
before: process_ptr_unary_op($file_name, $line_num,
$smpl_line_num,
$tree_code,
$op_name($dest),
$op_value($src0, $pointer_type));
```

```
$assign $lval $pointer_type $any_var
        $any_binary_op $pointer_type $any_var $pointer_type
$any_const:
before: process_ptr_binary_op_const($file_name, $line_num,
$smpl_line_num,
$tree_code,
$op_name($dest),
$op_value($src0, $pointer_type),
$op_value($src1, $pointer_type),
1);

$assign $lval $pointer_type $any_var
        $any_binary_op $pointer_type $any_const $pointer_type
$any_var:
before: process_ptr_binary_op_const($file_name, $line_num,
$smpl_line_num,
$tree_code,
$op_name($dest),
$op_value($src1, $pointer_type),
$op_value($src0, $pointer_type),
0);

$assign $lval $pointer_type $any_var
        $any_binary_op $pointer_type $any_var $pointer_type
$any_var:
before: process_ptr_binary_op($file_name, $line_num,
$smpl_line_num,
$tree_code,
$op_name($dest),
$op_value($src0, $pointer_type),
$op_value($src1, $pointer_type));
```

The array reference patterns are below. Different patterns are used depending on whether the array reference occurs on the left hand side or right hand side and whether the array is accessed via an array, pointer, or pointer dereference (such `(*a)[x]`). Note that the AST in GCC distinguishes between arrays and pointers. All of the array reference functions do the same thing: access the shadow state for both the array and the index and make sure the bounds of the array cannot be exceeded.

```
$array_ref $array_type $any_var $any_type $any_var:
inline before: process_array_ref_via_array();
```

```
$lhs_array_ref $array_type $any_var $any_type $any_var:
inline before: process_array_ref_via_array();

$array_ref $pointer_type $any_var $any_type $any_var:
inline before: process_array_ref_via_ptr();

$lhs_array_ref $pointer_type $any_var $any_type $any_var:
inline before: process_array_ref_via_ptr();

$ind_array_ref $any_type $any_var $any_type $any_var:
inline before: process_array_ref_via_ind_array();

$lhs_ind_array_ref $any_type $any_var $any_type $any_var:
inline before: process_array_ref_via_ind_array();
```

These patterns merely detect when zero is written into an array, thereby setting the `known_null` flag. The last pattern is for indirect references.

```
$assign $lhs_array_ref $array_type $any_var $any_type $any_value
$rval $integer_type $const_val(0):
before:process_zero_assign($file_name, $line_num, $smpl_line_num,
$op_addr($dest_op0));

$assign $lhs_array_ref $pointer_type $any_var $any_type $any_value
$rval $integer_type $const_val(0):
before:process_zero_assign($file_name, $line_num, $smpl_line_num,
$op_value($dest_op0, $pointer_type));

$assign $lhs_ind_array_ref $any_type $any_var $any_type $any_value
$rval $integer_type $const_val(0):
before:process_zero_assign($file_name, $line_num, $smpl_line_num,
$op_value($ind_dest_op0, $pointer_type));

$assign $lhs_indirect_ref $any_type $any_var
$rval $integer_type $const_val(0):
before:process_zero_assign_via_deref($file_name, $line_num,
$smpl_line_num,
$op_value($dest_op0, $pointer_type));
```

These patterns are used to approximate situations that be problematic when the left hand side contains a dereference. If the last non-null character is copied using one of these patterns, the checker assumes that a string copy by hand has occurred and checks to see if the

source can fit into the destination.

```
$assign $lhs_indirect_ref $any_type $any_var
$indirect_ref $any_type $any_var:
inline before: process_indirect_assign();

$assign $lhs_indirect_ref $any_type $any_var
$array_ref $array_type $any_var $any_type $any_value:
inline before: process_indirect_assign_array();

$assign $lhs_indirect_ref $any_type $any_var
$array_ref $pointer_type $any_var $any_type $any_value:
inline before: process_indirect_assign_array();

$assign $lhs_indirect_ref $any_type $any_var
$ind_array_ref $any_type $any_var $any_type $any_var:
inline before: process_indirect_assign_array();
```

The family of dynamic memory allocation functions is matched using this set of patterns. `$malloc_expr` (and related identifiers) is a special simplified C statement that combines a call to `malloc` with an immediate cast. This allows MUSE to know the type of the allocation. Calls to `malloc` without an immediate cast will instead match "`$sys_call_expr malloc`".

```
$assign $lval $pointer_type $any_var
$malloc_expr:
inline after:  process_malloc();

$assign $lval $pointer_type $any_var
$sys_call_expr malloc:
inline after:  process_malloc();

$assign $lval $pointer_type $any_var
$calloc_expr:
inline after:  process_calloc();

$assign $lval $pointer_type $any_var
$sys_call_expr calloc:
inline after:  process_calloc();

$assign $lval $pointer_type $any_var
$realloc_expr:
```

```
inline after:  process_realloc();

$assign $lval $pointer_type $any_var
$sys_call_expr realloc:
inline after:  process_realloc();
```

The following functions are used to match the string and input functions we are interested in. Refer to Section 3.2 for a description of the functionality behind many of these instrumentation functions.

```
$sys_call_expr strcat:
before: process_strcat($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$sys_call_expr strncat:
before: process_strncat($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type),
$fn_arg_value(2, $integer_type));

$sys_call_expr strcmp:
before: process_strcmp($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$sys_call_expr strcpy:
before: process_strcpy($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$sys_call_expr strncpy:
before: process_strncpy($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type),
$fn_arg_value(2, $integer_type));

$assign $lval $integer_type $any_var
$sys_call_expr strlen:
inline before: process_strlen();

$assign $lval $pointer_type $any_var
$sys_call_expr strchr:
after:  process_strchr($file_name, $line_num, $smpl_line_num,
$op_name($dest),
```

```
$op_value($dest, $pointer_type),
$fn_arg_value(0, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr strrchr:
after:  process_strchr($file_name, $line_num, $smpl_line_num,
$op_name($dest),
$op_value($dest, $pointer_type),
$fn_arg_value(0, $pointer_type));

$sys_call_expr strpos:
before: process_strpos($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type));

$sys_call_expr strrpos:
before: process_strpos($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type));

$sys_call_expr strspn:
before: process_strspn($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$sys_call_expr strcspn:
before: process_strspn($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$assign $lval $pointer_type $any_var
 $sys_call_expr strpbrk:
after:process_strstr($file_name, $line_num, $smpl_line_num,
 $op_name($dest),
 $op_value($dest, $pointer_type),
 $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$assign $lval $pointer_type $any_var
 $sys_call_expr strrpbrk:
after:process_strstr($file_name, $line_num, $smpl_line_num,
 $op_name($dest),
 $op_value($dest, $pointer_type),
 $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr strstr:
after:process_strstr($file_name, $line_num, $smpl_line_num,
 $op_name($dest),
 $op_value($dest, $pointer_type),
        $fn_arg_value(0, $pointer_type),
```

```
        $fn_arg_value(1, $pointer_type));

$assign $lval $pointer_type $any_var
 $sys_call_expr strtok:
after:process_strtok($file_name, $line_num, $smpl_line_num,
 $op_name($dest),
 $op_value($dest, $pointer_type),
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));

$assign $lval $pointer_type $any_var
 $sys_call_expr strsep:
before:process_strsep_before($file_name, $line_num,
$smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));
after:process_strsep_after($file_name, $line_num, $smpl_line_num,
 $op_name($dest),
 $op_value($dest, $pointer_type),
        $fn_arg_value(0, $pointer_type));

$sys_call_expr strtod:
before: process_strtod($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$sys_call_expr atof:
before: process_strtod($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$assign $lval $integer_type $any_var
$sys_call_expr strtol:
inline after:process_strtol();

$assign $lval $integer_type $any_var
$sys_call_expr strtoul:
inline after:process_strtol();

$assign $lval $integer_type $any_var
$sys_call_expr atoi:
inline after:process_strtol();

$assign $lval $integer_type $any_var
$sys_call_expr atol:
inline after:process_strtol();

$sys_call_expr strcoll:
before: process_strcoll($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $pointer_type));
```

```
$sys_call_expr strxfrm:
before: process_strxfrm($file_name, $line_num, $smpl_line_num);

$assign $lval $pointer_type $any_var
$sys_call_expr strdup:
after:process_strdup($file_name, $line_num, $smpl_line_num,
$op_name($dest),
$op_value($dest, $pointer_type),
$fn_arg_value(0, $pointer_type));

$assign $lval $pointer_type $any_var
 $sys_call_expr memchr:
after:process_memchr($file_name, $line_num, $smpl_line_num,
 $op_name($dest),
 $op_value($dest, $pointer_type),
 $fn_arg_value(0, $pointer_type),
 $fn_arg_value(2, $integer_type));

$sys_call_expr memmove:
before: process_memcpy($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type),
$fn_arg_value(1, $pointer_type),
$fn_arg_value(2, $integer_type));

$sys_call_expr memcpy:
before: process_memcpy($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type),
$fn_arg_value(1, $pointer_type),
$fn_arg_value(2, $integer_type));

$sys_call_expr memccpy:
before: process_memccpy($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type),
$fn_arg_value(1, $pointer_type),
$fn_arg_value(3, $integer_type));

$sys_call_expr bcopy:
before: process_bcopy($file_name, $line_num, $smpl_line_num,
$fn_arg_value(1, $pointer_type),
$fn_arg_value(0, $pointer_type),
$fn_arg_value(2, $integer_type));

$sys_call_expr memset:
before: process_memset($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
        $fn_arg_value(1, $integer_type));

$sys_call_expr bzero:
before: process_bzero($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(0, $pointer_type),
```

```
        $fn_arg_value(1, $integer_type));

$assign $lval $integer_type $any_var
$sys_call_expr fgetc:
inline after:process_getc();

$assign $lval $integer_type $any_var
$sys_call_expr getc:
inline after:process_getc();

$assign $lval $integer_type $any_var
$sys_call_expr getchar:
inline after:process_getc();

$sys_call_expr fgets:
before: process_fgets($file_name, $line_num, $smpl_line_num,
    $fn_arg_name(0),
$fn_arg_value(0, $pointer_type),
$fn_arg_value(1, $integer_type));

$sys_call_expr gets:
before: process_gets($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$sys_call_expr_arg scanf $integer_type * $any_var:
before:process_scanf($file_name, $line_num, $smpl_line_num,
$elt_offset,
$op_value($src0, $pointer_type),
 0,
 0);

$sys_call_expr_arg fscanf $integer_type * $any_var:
before:process_fscanf($file_name, $line_num, $smpl_line_num,
$elt_offset,
 $op_value($src0, $pointer_type),
 0,
 0);

$sys_call_expr_arg sscanf $integer_type * $any_var:
before:process_sscanf($file_name, $line_num, $smpl_line_num,
$elt_offset,
 $op_value($src0, $pointer_type),
0,
0);

$sys_call_expr_arg sprintf $integer_type * $any_var:
before:process_sprintf_arg($file_name, $line_num, $smpl_line_num,
$elt_offset,
1,
 $op_value($src0, $pointer_type));
```

```
$sys_call_expr sprintf:
before:process_sprintf($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$sys_call_expr_arg snprintf $integer_type * $any_var:
before:process_snprintf_arg($file_name, $line_num,
$smpl_line_num,
$elt_offset,
2,
$op_value($src0, $pointer_type));

$sys_call_expr snprintf:
before:process_snprintf($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type),
$fn_arg_value(1, $integer_type));

$sys_call_expr read:
after:process_read($file_name, $line_num, $smpl_line_num,
$fn_arg_value(1, $pointer_type),
1,
$fn_arg_value(2, $integer_type));

$sys_call_expr recv:
after:  process_read($file_name, $line_num, $smpl_line_num,
        $fn_arg_value(1, $pointer_type),
        1,
        $fn_arg_value(2, $integer_type));

$sys_call_expr fread:
after:process_fread($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type),
$fn_arg_value(1, $integer_type),
$fn_arg_value(2, $integer_type));
```

This section of patterns refer to other system calls that need to be accounted for in order for proper operation. The first three patterns look at `argc`, `argv`, and `envp` which are considered inputs to the system. Many of the other functions listed either null terminate a string or return a new string that is treated as a string constant.

```
$birth_parm main $integer_type argc:
after:birth_argc($file_name, $line_num, $smpl_line_num,
$op_value($src0, $integer_type));
```

```
$birth_parm main $pointer_type argv:
after:birth_argv($file_name, $line_num, $smpl_line_num,
$op_value($src0, $pointer_type));

$birth_parm main $pointer_type envp:
after:birth_envp($file_name, $line_num, $smpl_line_num,
$op_value($src0, $pointer_type));

$sys_call_expr getnameinfo:
after:process_getnameinfo($file_name, $line_num, $smpl_line_num,
$fn_arg_value(2, $pointer_type),
$fn_arg_value(4, $pointer_type));

$sys_call_expr getcwd:
after:syscall_assign_null($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$sys_call_expr strftime:
after:syscall_assign_null($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$sys_call_expr realpath:
after:syscall_assign_null($file_name, $line_num, $smpl_line_num,
$fn_arg_value(1, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr ctime:
after:process_string_cst($file_name, $line_num, $smpl_line_num,
$op_name($dest),
        $op_value($dest, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr strerror:
after:process_string_cst($file_name, $line_num, $smpl_line_num,
$op_name($dest),
$op_value($dest, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr crypt:
after:process_string_cst($file_name, $line_num, $smpl_line_num,
$op_name($dest),
$op_value($dest, $pointer_type));

$sys_call_expr gettimeofday:
after:process_gettimeofday($file_name, $line_num, $smpl_line_num,
$fn_arg_value(0, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr __ctype_b_loc:
after:process_ctype_b_loc($file_name, $line_num, $smpl_line_num,
```

```
$op_value($dest, $pointer_type));

$sys_call_expr glob:
after:process_glob($file_name, $line_num, $smpl_line_num,
$fn_arg_value(3, $pointer_type));

$sys_call_expr getopt:
after:process_getopt($file_name, $line_num, $smpl_line_num);

$assign $lval $pointer_type $any_var
$sys_call_expr getpwuid:
after:process_getpwuid($file_name, $line_num, $smpl_line_num,
$op_value($dest, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr getpwnam:
after:process_getpwuid($file_name, $line_num, $smpl_line_num,
$op_value($dest, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr getgrgid:
after:process_getgrgid($file_name, $line_num, $smpl_line_num,
$op_value($dest, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr getgrnam:
after:process_getgrgid($file_name, $line_num, $smpl_line_num,
$op_value($dest, $pointer_type));

$assign $lval $pointer_type $any_var
$sys_call_expr getenv:
after:process_getenv($file_name, $line_num, $smpl_line_num,
$op_addr($dest));

$assign $lval $pointer_type $any_var
$sys_call_expr getspnam:
after:process_getspnam($file_name, $line_num, $smpl_line_num,
$op_value($dest, $pointer_type));
```

BIBLIOGRAPHY

[1] Aleph One. Smashing The Stack For Fun and Profit. Phrack Volume Seven, Issue Forty-Nine, July 2003.

[2] L.O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, May 1994.

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

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

[5] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C programs. Proceedings of the Conference on Programming Languages Design and Implementation, June 2001.

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

[7] T. Ball and S. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. Proceedings of the SPIN 2000 Workshop on Model Checking of Software, August 2000.

[8] T. Ball and S. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Microsoft Research Technical Report MSR-TR-2002-09, 2002.

[9] T. Ball and S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. Invited talk at the 29th Annual Symposium on Principles of Programming Languages, January 2002.

[10] T. Ball and S. Rajamani. SLIC: A Specification Language for Interface Checking. Microsoft Research Technical Report MSR-TR-2001-21, 2001.

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

[12] G. Brat, K. Havelund, S. Park and W. Visser. Java PathFinder - A second generation of a Java model checker. Workshop on Advances in Verification. July 2000.

[13] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. Proceedings of the International Symposium of Software Testing and Analysis (ISSTA '02), July 2002.

[14] C. Boyapati, A. Salcianu, W. Beebee Jr., M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. Proceedings of the Conference on Programming Language Design and Implementation, June 2003.

[15] J. Burch, E. Clarke, D. Long, K. McMillian, and D. Dill. Symbolic Model Checking: 10^{20} states and beyond. Information and Computation, June 1992.
[16] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software Practice and Experience, July 2000.
[17] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using Symbolic Execution for Verifying Safety-Critical Systems. Proceedings of the 9th International Symposium on Foundations of Software Engineering, Sept. 2001.
[18] J. Colbeigh, L. Clarke, and L. Osterweil. FLAVERS: A finite state verification technique for software systems. IBM System Journal, Vol. 41, No. 1, 2002.
[19] W. Chan, R. Anderson, P. Beane, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model Checking Large Software Specifications. IEEE Transactions on Software Engineering, July 1998.
[20] Checker. http://www.gnu.org/software/checker/checker.html
[21] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. Proceedings of the Conference on Computer and Communications Security, November 2002.
[22] B. Chess. Improving Computer Security using Extended Static Checking. Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002.
[23] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. Proceedings of the 22nd International Conference on Software Engineering, June 2000.
[24] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proceedings of the 7th USENIX Security Conference, January 1998.
[25] R. DeLine and M. Fahndrich. Enforcing High-Level Protocols in Low-Level Software. Proceedings of the Conference on Programming Languages Design and Implementation, June 2001.
[26] D. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. Saxe. Extended Static Checking. Research Report 159, Compaq Systems Research Center, December 1998.
[27] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. Proceedings of the Conference on Programming Languages Design and Implementation, June 2003.
[28] J. Dvorak. Magic Number: 30 Billion. PC Magazine, August 2003.
[29] J. Dykstra. Software Verification and Validation with Destiny: A parallel approach to automated theorem proving. Crossroads: ACM Student Magazine, January 2002.
[30] The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology report, prepared by RTI (project 7007.011), May 2002.
[31] T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming: Introduction. Communications of the ACM, October 2001.
[32] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. Proceedings of Operating Systems Design and Implementation, September 2000.
[33] D. Engler, D. Chen, S. Hallem, A. Chou, B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. Proceedings of the 18th Symposium on Operating System Principles, October 2001.

[34] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE Transactions on Software Engineering, February 2001.
[35] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. IEEE Software, Jan./Feb. 2002.
[36] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended Static Checking for Java. Proceedings of the Conference on Programming Languages Design and Implementation, June 2002.
[37] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. Proceedings of the Symposium on Principles of Programming Languages, January 2002.
[38] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. Proceedings of the Symposium on the Principles of Programming Languages, January 2001.
[39] J. Forrester and B. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. Proceedings of the 4th USENIX Windows System Symposium, August 2000.
[40] Free Software Foundation. SSA for Trees. <http://www.gnu.org/software/gcc/projects/tree-ssa> 2002.
[41] G. Freidman, A. Hartman, K. Nagin, and T. Shiran. Projected State Machine Coverage for Software Testing. Proceedings of the International Symposium on Software Testing and Analysis, July 2002.
[42] R. Ghiya and L. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. Proceedings of the 23rd Symposium on Principles of Programming Languages, January 1996.
[43] P. Godefroid. Model Checking for Programming Languages using VeriSoft. Proceedings of the 24th Symposium on Principles of Programming Languages, January 1997.
[44] R. Gupta. A Fresh Look at Optimizing Array Bound Checks. Conference on Programming Language Design and Implementation, June 1990.
[45] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. Proceedings of the 1992 Winter Usenix Conference, January 1992.
[46] E. Haugh and M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. Proceedings of the 10th Network and Distributed System Security Symposium, Feb. 2003.
[47] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, August 1992.
[48] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proceedings of the Symposium on Principles of Programming Languages, January 2002.
[49] G. Holzmann. Marktoberdorf Lecture Notes on Software Model Checking. Nato Summerschool, August 2000.
[50] G. Holzmann. The Spin Model Checker, IEEE Transactions on Software Engineering, May 1997.
[51] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
[52] iSec Security Research: Vulnerabilities 2004. http://www.isec.pl/vulnerabilities04.html

[53] D. Jackson. Aspect: An Economical Bug-Detector. 1991.
[54] D. Jackson. Alloy: A Lightweight Object Modelling Notation. MIT Laboratory for Computer Science Technical Report 797, February 2000.
[55] D. Jackson and M. Vaziri. Finding Bugs with a Constraint Solver. Proceedings of the International Symposium on Software Testing and Analysis, August 2000.
[56] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. Proceedings of the USENIX Annual Technical Conference, June 2002.
[57] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. Proc. of the 3rd International Workshop on Automated Debugging, May 1997.
[58] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-base programming environment for the C language. Proceedings of the Summer Usenix Conference, 1988.
[59] D. Keen, F. Lim, F. Chong, P. Devanbu, M. Farrens, P. Sultana, C. Zhuang, and R. Rao. Hardware Support for Pointer Safety in Commodity Microprocessors. UC Davis Tech Report CSE-2002-1, January 2002.
[60] T. Kremeneck and D. Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. Proceedings of the Static Analysis Symposium, June 2003.
[61] D. Lacey, N. Jones, E. Van Wyk, C. Fredericksen. Proving Correctness of Compiler Optimizations by Temporal Logic. Proceedings of the Symposium on Principles of Programming Languages, January 2002.
[62] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. Proceedings of the 2001 USENIX Security Symposium, 2001.
[63] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. Proceedings of the 12th USENIX Security Symposium, Aug. 2003.
[64] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. Proceedings of the Conference on Programming Language Design and Implementation, June 1995.
[65] K. Lhee and S. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. Proceedings of the 11th USENIX Security Symposium, Aug. 2002.
[66] D. Lie, A. Chou, D. Engler, and D. Dill. A Simple Method for Extracting Models from Protocol Code. Proceedings of the 28th International Symposium on Computer Architecture, July 2001.
[67] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
[68] Microsoft TechNet Security.http://www.microsoft.com/technet/Security/default.mspx
[69] R. Moore. A Universal Dynamic Trace for Linux and other Operating Systems. UKUUG Linux Developers Conference 2001.
[70] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug finding. Invited talk at the Workshop on Software Model Checking, July 2003.
[71] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. Proceedings of the Conference on Operating System Design and Implementation, December 2002.
[72] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. Proceedings of Operating Systems Design and Implementation, October 1996.

[73] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. Proceedings of the Symposium on Principles of Programming Languages, January 2002.
[74] G. Nelson. Techniques for Program Verification. Ph.D. thesis, Stanford University, 1980.
[75] R. Netzer and B. Miller. Improving the Accuracy of Data Race Detection. Proceedings of the Symposium on Principles and Practice of Parallel Programming, April 1991.
[76] Parasoft Corporation. Insure++: An Automatic Runtime Error Detection Tool. Technical Report PS961-INS1.
[77] B. Perens. Electric Fence. <http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz>
[78] Perl v5.6 Documentation: perlsec. <http://www.perldoc.com/perl5.6/pod/perlsec.html>
[79] A. Pnueli. The Temporal Logic of Programs. Symposium on Foundations of Computer Science, 1977.
[80] Pointer-Intensive Benchmark Suite <http://www.cs.wisc.edu/~austin/ptr-dist.html>
[81] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accesses Memory Regions. Proceedings of the Conference on Programming Languages Design and Implementation, June 2000.
[82] U. Sammapun, R. Sharykin, M. Delap, M. Kim, and S. Zdancewic. Formalizing Java-MaC. Proceedings of the Third Runtime Verification Workshop, July 2003.
[83] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. Proceedings of the 10th USENIX Security Symposium, Aug. 2001.
[84] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. Proceedings of the Conference on Programming Languages Design and Implementation, June 1994.
[85] K. Rustan, M. Leino, G. Nelson, and J. Saxe. ESC/Java User's Manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.
[86] M. Tikir and J. Hollingsworth. Efficient Instrumentation for Code Coverage Testing. Proceedings of the International Symposium on Software Testing and Analysis, July 2002.
[87] F. Tip. A survey of program slicing techniques. Journal of Programming Languages, September 1995.
[88] Valgrind. http://valgrind.kde.org.
[89] D. Wagner, J. Foster, E. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed Security Symposium, February 2000.
[90] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. Proceedings of the 14th Symposium on Operating System Principles, June 1993.
[91] H. Wasserman and M. Blum. Software Reliability via Run-Time Result-Checking.
[92] J. Whittaker and A. Jorgensen. Why Software Fails. ACM Software Engineering Notes, 1999
[93] Y. Xie, A. Chou, D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to

Detect Memory Access Errors. Proceedings of the 11th International Symposium on the Foundations of Software Engineering, Sep. 2003.
[94] A. Zeller. Automated Debugging: Are We Close? IEEE Computer, November 2001.
[95] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. September 2001, in submission.
[96] J. Zhu. Symbolic Pointer Analysis. Proceedings of the 2002 International Conference on Computer Aided Design, 2002.
[97] L. Zuck, A. Pnueli, and R. Leviathan. Validation of Optimizing Compilers.K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002.