DYNAMIC POINTER TRACKING AND ITS APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

by

Kun Zhang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2010

COPYWRITE 2010 BY KUN ZHANG

DYNAMIC POINTER TRACKING AND ITS APPLICATIONS

Approved by:

Dr. Santosh Pande, Advisor
College of Computing
Georgia Institute of Technology

Dr. Jonathon Giffin
College of Computing
Georgia Institute of Technology

Dr. Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Nathan Clark
College of Computing
Georgia Institute of Technology

Dr. Hyesoon Kim
College of Computing
Georgia Institute of Technology

Date Approved: December 10, 2009

ACKNOWLEDGEMENTS

I wish to thank all of the people who support me during the time of pursuing my PhD.

First of all, I would like to give my extraordinary gratitude to my advisor Santosh Pande for his good supervision and valuable advice through my PhD study. He provided me encouragement and support in various ways. His passions for research and his novel ideas nourish my growth as a PhD student.

Thanks to all of my committee members, Dr. Nathan Clark, Dr. Jonathon Giffin, Dr. Hyesson Kim and Dr. Hsien-Hsin S. Lee for advising me in my dissertation work. Their suggestions and comments help me to improve the quality of this dissertation.

I would like to acknowledge my colleagues, Tao Zhang, Xiaotong Zhuang, Tushark Kumar, Romain Cledat, and Jaswanth Sreeram. It is my pleasure to work with them in our compiler team.

Special thanks to my husband and my parents for their constant understanding, unselfish support and persistent confidence in me.

Finally, I would like to thank all my friends at different places who make my graduate study enjoyable.

TABLE OF CONTENT

Page

ACKNOWLEDGEMENTS III

LIST OF FIGURES X

SUMMARY XIII

1. INTRODUCTION 1

1.1 Motivation 1

1.2 Traditional Pointer Analysis Approaches 3

1.3 Our Approach 5

1.4 Contribution Statement 6

1.5 Thesis Organization 7

2. DYNAMIC POITER TRACKING MOTIVATION EXAMPLES 9

2.1 Memory Protection 9

2.2 Program Parallelism 10

2.3 Path Prediction 11

2.4 Dynamic Invariant Detection 12

2.5 Garbage Collection 14

2.6 Summary 15

3. BASIC DYNAMIC POINTER TRACKING ALGORITHMS 16

3.1 Data Structures 17

3.2 Special Issues on Data Structure Design 17

3.2.1 Pointer on the Heap 18

3.2.2 Recursive Function 20

3.2.3 Explicit and Implicit Pointer Array 20

3.2.4 Pointer Parameter and Pointer Return.. 22

3.2.5 Pointer Type Casting.. 23

3.2.6 Function Pointer.. 24

3.2.7 Unions 24

3.3 Data Structure Management 25

3.4 Update Instructions 28

3.4.1 Function Entry and Exit... 28

3.4.2 Malloc and Free 28

3.4.3 Pointer Assignments... 29

3.5 Limitations of the Work 31

3.5.1 Compatibility with Legacy Code 31

3.5.2 Separation of Space... 32

3.5.3 Visibility of Pointers and Pointer Assignments 32

3.5.4 Setjmp/Longjmp 33

3.6 User Support 33

3.7 Possible Optimizations.. 34

3.7.1 Pointer Definition and Use.. 34

3.7.2 Remove Redundant Pointer Chasing Update...................................... 36

3.7.3 Delay Update out of Loop... 39

3.7.4 Delay Update for Special Patterns 41

3.7.5 Remove Update for Pointer Initialization 44

3.7.6 Propagate Update 46

3.7.7 Invariant Removal... 48

3.7.8 On Demand Dynamic Pointer Tracking.. 49

3.8 Dynamic Pointer Tracking Experimental Results 50

3.8.1 Static Pointer Analysis 51
3.8.2 Dynamic Pointer Analysis 53
3.8.3 Recursive Function Analysis 55
3.8.4 Performance Evaluation 56
3.8.5 Dynamic Load/Store 57
3.8.6 Space Cost 58
3.9 Summary 58
4. MEMORY PROTECTION THROUGH DYNAMIC POINTER TRACKING 59
4.1 Introduction of the Problem 59
4.2 Previous Approaches 61
4.2.1 Control Flow Monitoring 62
4.2.2 Non-control Data Attack Detection 64
4.2.3 Bounds Checking 68
4.2.4 Other Approaches 70
4.2.5 Previous Work against Attacks 71
4.2.6 Our Approach 74
4.3 Attack Model 74
4.4 Compiler Analysis and Optimizations 75
4.4.1 Baseline Scheme 75
4.4.2 Compiler Framework Overview 76
4.4.3 Write Range Identification 78
4.4.4 Protection Points Hoisting and Delaying 79
4.4.5 Protection Points Selection 84
4.4.6 Grouping Protection Operations 86
4.5 Data Structure Design 93

4.5.1 Action Table and Special Instruction 93

4.5.2 Points-to Table 95

4.6 Architecture Support 96

4.7 Baseline against Attacks 97

4.8 Evaluation 98

4.8.1 Performance Measurement 99

4.8.2 Security Measurement 100

4.8.3 Space Cost Measurement 104

4.9 Summary 105

5. GARBAGE COLLECTION WITH DYNAMIC POINTER TRACKING 107

5.1 Introduction 107

5.2 Traditional Garbage Collection Techniques 107

5.2.1 Reference Counting Collectors 107

5.2.2 Tracing Collectors 108

5.2.3 Challenges of Garbage Collection for C and C++ 109

5.2.4 Our Contribution 110

5.3 Garbage Collection Algorithms 111

5.3.1 Maintaining Object Table 111

5.3.2 Maintaining Pointer Table 113

5.3.3 Maintaining Pointer Address 115

5.4 Heap Shape Analysis Results 115

5.5 Garbage Collection Results 117

5.6 Summary 118

6. DYNAMIC INVARIANT PREDICATE IDENTIFICATION WITH DYNAMIC POINTER TRACKING 120

6.1 Static Analysis 120

6.2 Dynamic Analysis 121

6.3 Illustration 123

6.4 Summary 127

7. CONCLUSION 128

REFERENCES 130

LIST OF TABLES

Page

Table 1. Example pointer table. 17

Table 2. Updating code for pointer assignments. 31

Table 3. Pointer definition, dereference, and use statements... 35

Table 4. The description of olden benchmark. 50

Table 5. Number of local and global pointers.. 51

Table 6. Olden benchmark parameter. 53

Table 7. Pointer table size. 58

Table 8. Action table.. 94

Table 9. Ref input set of benchmarks 98

Table 10. Space cost measurement. 104

LIST OF FIGURES

Page

Figure 1. Buffer overflow attack motivation example. ... 9

Figure 2. Parallelism optimization motivation example. ... 10

Figure 3. Program slices of Figure 2. ... 10

Figure 4. Path prediction motivation example. ... 11

Figure 5. Dynamic invariant detection example. ... 12

Figure 6. Garbage collection motivation example. ... 14

Figure 7. Static analysis framework overview. ... 16

Figure 8. Pointer on the heap example. ... 18

Figure 9. Smart offset for pointer on the heap. ... 19

Figure 10. Recursive function example in power. ... 20

Figure 11. Implicit pointer array example in mst. ... 21

Figure 12. Pointer parameter example. ... 22

Figure 13. Pointer type cast example in Bisort. ... 23

Figure 14. Union example. ... 25

Figure 15. Pointers on the heap and in recursive functions example in bisort. ... 26

Figure 16. User redefined malloc example. ... 29

Figure 17. Pointer chasing example. ... 31

Figure 18. Redundant update example. ... 36

Figure 19. Algorithm to identify pointer definitions and their reachable dereferences. ... 38

Figure 20. Algorithm to identify redundant pointer definition. ... 39

Figure 21. Function uloop2 in Twolf. ... 40

Figure 22. Linked list example 1 in Health. ... 41

Figure 23. Linked list example 2 in Health. ... 42

Figure 24. Algorithm to delay update for special patterns... 44

Figure 25. NULL initialization example in Perimeter.. 45

Figure 26. Algorithm to remove update for pointer initialization. 46

Figure 27. Treeadd example. 47

Figure 28. Algorithm to propagate update.. 48

Figure 29. Adpcm_coder example.. 49

Figure 30. Gzip example... 49

Figure 31. Static pointer assignment... 53

Figure 32. Dynamic pointer assignments.. 54

Figure 33. Recursive and none recursive functions.. 55

Figure 34. Pointer assignment in recursive and none recursive functions....................... 56

Figure 35. Updating overhead... 57

Figure 36. Cache effect... 57

Figure 37. Attack without control flow tampering. 65

Figure 38. Security hole in AccMon.. 66

Figure 39. Memory attack example. 72

Figure 40. Compiler framework overview.. 77

Figure 41. Write range example.. 78

Figure 42. Algorithm for hoisting protection points... 80

Figure 43. Auxiliary algorithm for hoisting protection points.. 81

Figure 44. An example of hoisting protection points. 83

Figure 45. Algorithm for selecting protection points.. 85

Figure 46. An example of protection operations grouping.. 87

Figure 47. An example of data layout... 88

Figure 48. Pseudo code for the grouping algorithm. 90

Figure 49. An example of variable grouping algorithm. 91
Figure 50. Architecture support overview. 97
Figure 51. Performance degradation.. 99
Figure 52. Security measurement 100
Figure 53. Detected simulated attacks. 102
Figure 54. Buffer overflow in Wu-FTP. 103
Figure 55. Example object table. 112
Figure 56. Cyclic data structure example code. 113
Figure 57. Cyclic data structure graphic view. 113
Figure 58. Garbage collection with pointer casting. 114
Figure 59. Heap object size... 116
Figure 60. Number of heap objects.. 116
Figure 61. Average heap object size.. 117
Figure 62. Garbage objects detection result.. 118
Figure 63. Dynamic invariant predicate identification algorithm................................. 122
Figure 64. Process invariant predicate list. 122
Figure 65. Process maybe invariant predicate list. 123
Figure 66. Process stack of basic blocks marked for removal....................................... 123
Figure 67. Example showing predicates inside a loop.. 124
Figure 68. Control flow graph when path P1 is taken... 125
Figure 69. Control flow graph when paths P1 and P3 are taken................................... 126
Figure 70. Control flow graph when paths P1 and P4 are taken................................... 126
Figure 71. Control flow graph when paths P2 and P4 are taken................................... 126

SUMMARY

Due to the significant limitations of static analysis and the dynamic nature of pointers in weakly typed programming languages like C and C++, the points-to sets obtained at compile time are quite conservative. Most static pointer analysis methods trade the precision for the analysis speed. The methods that perform the analysis in a reasonable amount of time are often context and/or flow insensitive. Other methods that are context, flow, and field sensitive have to perform the whole program inter-procedural analysis, and do not scale with respect to the program size. A large class of problems involving optimizations such as instruction prefetching, control and data speculation, redundant load/store instructions removal, instruction scheduling, and memory disambiguation suffer due to the imprecise and conservative points-to sets computed statically. One could possibly live without optimizations, but in domains involving memory security and safety, lack of the precise points-to sets can jeopardize the security and safety. In particular, the lack of dynamic points-to sets drastically reduce the ability to reason about a program's memory access behavior, and thus illegal memory accesses can go unchecked leading to bugs as well as security holes. On the other hand, the points-to sets can be very useful for other domains such as the heap shape analysis and garbage collection. The knowledge of precise points-to sets is therefore becoming very important, but has received little attention so far beyond a few studies, which have shown that the pointers exhibit very interesting behaviors during execution. How to track such behaviors dynamically and benefit from them is the topic covered by this research.

In this work, we propose a technique to compute the precise points-to sets through dynamic pointer tracking. First, the compiler performs the pointer analysis to obtain the static points-to sets. Then, the compiler analyzes the program, and inserts the necessary instructions to refine the points-to sets. At runtime, the inserted instructions automatically

update the points-to sets. Dynamic pointer tracking in software can be expensive and can be a barrier to the practicality of such methods. Several optimizations including removal of redundant update, post-loop update, special pattern driven update removal, pointer initialization update removal, update propagation, invariant removal, and on demand update optimization are proposed. Our experimental results demonstrate that our mechanism is able to compute the points-to sets dynamically with tolerable overheads. Finally, the memory protection and garbage collection work are presented as the consumers of dynamic pointer tracking to illustrate its importance. In particular, it is shown how different memory properties can be easily tracked using the dynamic points-to sets opening newer possibilities.

CHAPTER 1

INTRODUCTION

In this thesis, we develop a technique called dynamic pointer tracking to help memory protection, memory leak detection, garbage collection, path prediction, and program analysis at runtime. Instead of being a pure static approach or a pure dynamic approach, our solution refines the static pointer information dynamically through the compiler analysis. With the exact dynamic pointer information, a number of program optimizations and software protection techniques are enabled or enhanced. However, pointer tracking simply through a software mechanism can slow down the execution. In order to alleviate this run-time overhead, we develop several optimization mechanisms to achieve dynamic pointer tracking efficiently. Thus, our technique can improve both the static and dynamic program analysis with reasonable performance cost.

1.1 Motivation

Many programming languages in use today, such as C and C++, make extensive use of pointers. Pointers are used in C programs to realize the call by pointer semantics in function calls, to implement polymorphism in object-oriented programming languages via virtual function pointers, to implement complex data structures like list, tree, and array, and to efficiently share and access objects allocated dynamically on the heap through pointer assignments or dereference. The use of pointers is a powerful and convenient mechanism, but it also makes programs hard to understand, hard to reason about, and sometimes prevents an optimizing compiler from making code-improving transformations.

One of the key pointer attributes is the points-to information. A pointer's points-to set enumerates a set of (abstract) memory locations where a pointer may potentially point to. Points-to sets are often computed statically and give an (over) approximation of a

program's memory behavior. The use of points-to information is becoming more and more important, especially for those programming languages where a pointer could point anywhere. The points-to information is critical and is used for many reasons – to improve the performance of programs with heavy use of pointers, to make better branch prediction in architecture, to optimize instruction scheduling in parallelism exploration, to reduce memory access misses, to support multithreaded programs with the precise data dependence information, to assist speculative computation, to help reduce the program slice size in software debugging, to detect memory leaks, to assist garbage collection, and to achieve full software security.

In practice, pointer analysis presents a challenging problem for a compiler or a program analysis tool: it is hard to extract the precise points-to information statically. There has been an amount of research tackling pointer analysis [29][68][7][37]. However, several problems remain to be solved further. First, the analysis must promise accurate points-to sets. Static program analysis could not obtain the precise information due to the dynamic nature of pointers whose targets are frequently modified, and due to the allowed pointer arithmetic in programming languages like C and C++. Without the knowledge of dynamic pointer behavior, conservative assumptions about memory locations accessed through pointers must be made. These assumptions can adversely affect the analysis precision and the optimization effectiveness. One could possibly live without optimizations, but in domains involving memory security and safety, lack of the precise points-to sets can jeopardize the security as shown by Zhang et al.[1]. Thus, how to track pointers dynamically becomes an important problem that has to be addressed. Second, the analysis must be efficient and practical. Dynamic pointer tracking offers benefits in many different settings, but the huge overheads involved make it infeasible. Although a naïve dynamic pointer analysis could show significant slowdown, tremendous optimizations development and smart data structures design could improve the efficiency. On the other hand, driven by different customer requirements and different application

domains, some domain specific techniques based on the features could further speed up the tracking process. In this thesis, we explore the interesting research topic on how to realize dynamic pointer tracking with tolerable performance degradation.

1.2 Traditional Pointer Analysis Approaches

An attractive feature of the C programming language is that it allows the developer manage and control the memory use. This feature is critical for a lot of system level programming environment. It is one of the reasons why C is popular today. Pointer analysis involves a big literature. We discuss some interesting related work in this section.

There is a lot of work [2][29][68][7][37] investigating static points-to set analysis. The analysis could be roughly classified into different categories according to that if it is context sensitive [29][31][74] or context insensitive[68], flow sensitive [17] or flow insensitive [7][11][37], and field sensitive [57] or field insensitive.

Markus Mock et al. [48] provided an empirical study of pointers in C programs. They implemented two static pointer analysis algorithms proposed by Steengaard [68] and Das [24], and an extension of Steengaard's work [64]. They compared the static and dynamic points-to sets. They measured the points-to set size at every executed dereference point in a program (a load instruction or a store instruction), and computed the average static points-to size. The average dynamic points-to size is obtained by considering the execution frequency of every dereference. Their experiments show that for a large number of programs, the static pointer analysis is so conservative that the points-to set size is several times bigger than the actual points-to set size at runtime. They observed that on average 98% of all dynamic points-to sets are singletons, but only 41% of the static points-to sets are singletons. So providing a technique to keep tack of the dynamic points-to sets like ours is critical in program understanding and optimization.

Driven by the results of [48], Mock et al. [49] investigated the application of dynamic points-to data on improving the program slicing accuracy. They expected the dynamic program slice size to become much smaller. However, their experiments showed that the improved points-to information could not reduce the slice size significantly.

Later Zhang et al. [78] further studied the feasibility of using the dynamic pointer information to refine the program slices periodically. They discovered that in a given time period, the precise points-to sets could greatly improve the program slices. Their experiments demonstrate one successful application of the dynamic points-to data on program optimizations.

Rodric M. Rabbah et al. [59] shed light on the importance of the points-to data in speculative computation and program predication. Their work aims at obtaining the data and control flow dependence through compiler analysis. However, the statically unknown or imprecise points-to information could weaken their analysis strength. Orchestrated with our dynamic pointer tracking scheme, their speculation could be improved. Similarly, dynamic pointer tracking could assist data prefetching [5].

There is several type based pointer analysis. CQual [87] is a type-based tool checking C program properties. It is based on extra user defined type qualifiers annotations, and verifies the qualifiers along the program flow paths. CQual's dependence on the user data hinders it from being a completely automatic tool.

CCured [51] designs a type system reasoning about a program's safety. It classifies pointers to three categories, safe pointer, sequence pointer and wild pointer based on the use of pointers. It does null checking on safe pointers, does bounds checking on sequence pointers, and does bounds and tag checking on wild pointers. There are two major differences between CCured and our work. CCured deploys bounds information around the pointer. It is not good for some work like garbage collection and memory security, which require the pointer target to be centralized. Second, CCured provides function calls

for bounds checking, but we provide the compiler generated assembly code, that is more efficient.

Later BLAST [9] extends CCured by plugging in a model checker. It removes redundant runtime checks by examining if there is program safety property violation on all program paths. BLAST provides an error trace as the feedback to the developer. Their work is based on the assumption that a program safety violation must go through an out of bounds pointer dereference. BLAST improves CCured by reducing the runtime checking overhead.

1.3 Our Approach

State-of-art pointer analysis is either too slow or too imprecise for program optimizations in the real world. In this work, we propose a hybrid pointer analysis that tracks the pointers targets dynamically with static analysis, and design special data structures to perform the tracking precisely and efficiently. Our approach has the following advantages.

Portability and Modularity: Our work is implemented at the higher level of the program representation. The pure dynamic pointer tracker is independent of the system. It could be applied to any platform and compiler. This is an advantage over other approaches that are specific to the hardware or compiler.

Compatibility and Flexibility: The core approach is compatible with dynamically linked libraries and unmonitored files. The developer could choose to only track the interesting pointers. Our scheme could be applied to instrumented and uninstrumented files, as well as instrumented and uninstrumented statements. The program compatibility is supported at fine level. Our work preserves most of the flexibilities of the C language on the utility of pointers. The developer is not required to provide any additional information like quantifiers or annotations regarding the pointers.

Scalability and Efficiency: The space cost is scalable with the number of pointers created in a program. The runtime overhead is linear with the number of pointer assignment in the program. The dynamic pointer tracker implementation introduces a linear space and execution overhead.

Our core approach works as follows. First, the compiler analyzes a program, and determines where to insert the necessary instructions for updating the points-to information. This is our baseline. Then, we propose several optimization techniques to reduce the overheads. Finally, we apply our dynamic pointer tracking technique on achieving memory protection and garbage collection. Our experimental results demonstrate that our mechanism is able to efficiently compute the precise points-to sets.

1.4 Contribution Statement

We provide a dynamic pointer tracking technique different from previous limited pointer analysis related work that is mostly static. Dynamic pointer tracking purely in software can be expensive; we propose several optimization techniques to make it tractable. We show how this analysis is essential (a must) for technique involving software security (memory protection); we also show how this powerful runtime analysis enables new optimizations (on-the-fly garbage collection). We have an ambitious goal to provide an algorithm to monitor the pointer behavior in an efficient way. We argue that the pure static pointer analysis cannot achieve precise points-to information, and a dynamic approach like ours is necessary. We further point out that purely runtime information based approaches lack of the ability of understanding the program semantics. In other words, a semantic gap exists. In this work, we take a novel approach to bridge the semantic gap through the compiler analysis and the runtime support.

We propose a framework of applying dynamic pointer tracking on garbage collection. Based on the program semantics, we develop several garbage collection algorithms and illustrate the pointer behavior's impact on garbage collection. Through the

static analysis of programs and the smartly inserted update instructions, the garbage objects could be detected precisely, immediately and aggressively using different techniques.

We are the first to apply dynamic pointer tracking on memory protection. We develop an intrusion detection system through dynamic memory access control. Our scheme predicts the memory access pattern, and monitors the program execution at a very fine granularity. The major concern of the scheme is the performance degradation and the necessary data structures size. So we propose several optimizations based on memory access control to reduce the overheads.

1.5 Thesis Organization

In this introduction, we introduce the problem of dynamic pointer tracking. We discuss why this problem is full of challenges, and is critical for program analysis and optimizations. We briefly discuss previous solutions to this problem, show their limitations, and talk about our approach at a high-level view. The rest of the thesis is organized as follows.

Chapter 2 illustrates the possible applications of our dynamic pointer tracking technique, and further motivates the demand of the precise points-to information.

Chapter 3 focuses on our basic dynamic pointer tracking algorithm design. We show our analysis model and the essential data structures for performing the necessary update. This section further discusses kinds of possible optimization schemes.

Chapter 4 presents the role of dynamic pointer tracking in garbage collection. The first scheme aims at collecting garbage objects for C and C++ programming languages. It takes the impact of dynamic pointer tracking on garbage detection into considerations and makes better garbage collection based on the current context.

Chapter 5 elaborates on the second application of dynamic pointer tracking – memory protection. With the precise points-to sets, we could offer fine level memory protection

Chapter 6 proposes a framework to identify the invariant predicate dynamically with the help of dynamic points-to information.

Finally in chapter 7, we summarize the main contributions of this thesis.

CHAPTER 2

DYNAMIC POITER TRACKING MOTIVATION EXAMPLES

Pointer analysis involves a great amount of work. The technique could be customized and applied to different domains. In this section, we briefly discuss several examples to motivate the need for our dynamic pointer tracking technique. The possible consumers of the precise points-to information are program optimizations and memory security.

2.1 Memory Protection

Dynamic pointer tracking has a wide range of applications. In [1] Zhang et al. proposed a memory tampering detection algorithm that dynamically controls the access permissions for data objects. In their work, every object has an access permission bit. Their work involves the information regarding which object should be accessed by which store instruction. This information is gleaned through static program analysis, and the guards are inserted around the store instructions to check the permissions. For multiple aliased dynamic objects, such a scheme cannot offer full protection unless the precise points-to sets are available at runtime. In that work, the profile based analysis compromises the protection strength.

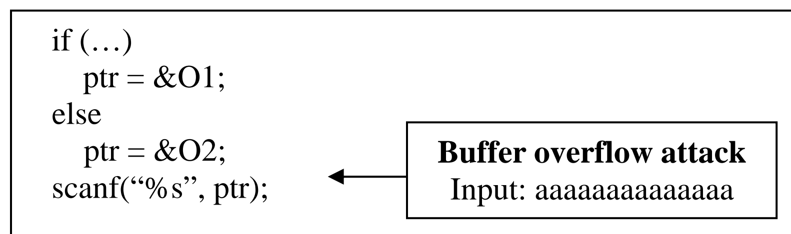


Figure 1. Buffer overflow attack motivation example.

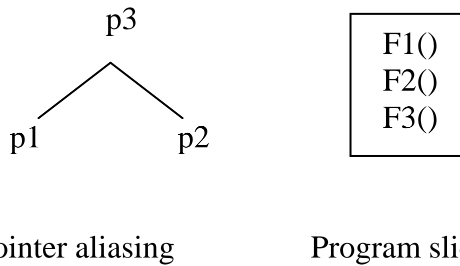
Consider the buffer overflow attack example shown in Figure 1. If in the training runs the pointer ptr always points to the object O1, then the estimated points-to object of ptr is O1. Suppose that in the testing run, the else branch is taken; then ptr points to the

object O2. In this scenario, the store instructions in the scanf function are not be checked, since the current object pointed to by ptr is not as predicted. This problem can be tackled only by determining the dynamic points-to sets. Thus, dynamic pointer tracking is not optional, but a must to achieve full security.

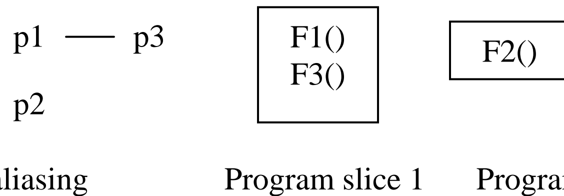
2.2 Program Parallelism

```
main(){
   p1 = malloc();
   p2 = malloc();
   if(…)
      S1: p3 = p1;
   else
      S2: p3 = p2;
   F1(p1);
   F2(p2);
   F3(p3);
}
```

Figure 2. Parallelism optimization motivation example.



(a) Aliased pointers and program slice through static analysis.



(b) Aliased pointers and refined program slices after the statement S1.

Figure 3. Program slices of Figure 2.

Dynamic pointer tracking is very useful in improving the parallelism and reducing the program slice size. Figure 2 shows such an example. In this code segment, there are

three pointers, p1, p2, and p3. Through static pointer analysis, we observe that p3 could be aliased with both p1 and p2; thus the statements through the pointers p1, p2, and p3 in the functions F1, F2, and F3 belong to one program slice, and must be executed sequentially as shown in Figure 3 (a). However, at runtime, only one of the statements S1 and S2 is executed. In other words, at a given time, p3 could only be aliased with one of p1 and p2. If the statement S1 is performed, then p3 is aliased with p1, and the static program slice is split in two parts as shown in Figure 3 (b). The two program slices could be executed simultaneously as two threads to achieve better parallelism. Thus, this example demonstrates that parallelism could benefit from dynamic pointer tracking.

2.3 Path Prediction

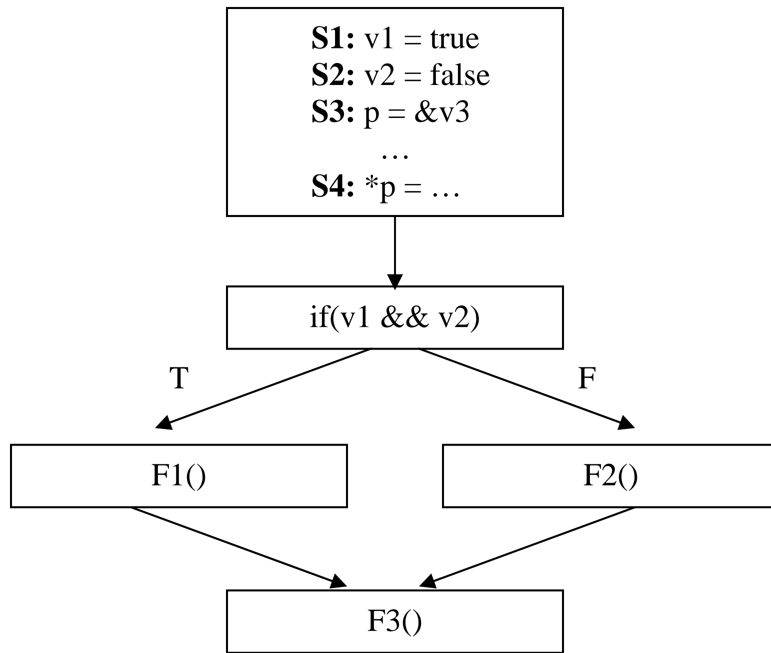


Figure 4. Path prediction motivation example.

Figure 4 shows one example for path prediction using the points-to information. There are three variables involved in the predication, v1, v2, and p. Assume that the pointer p could point to anywhere. We could not predicate which path will be taken statically. At runtime, after the statements S1 and S2 are executed, the definitions of v1 and v2 are available. However, without the information regarding where the pointer p is

pointing, we still could not predict the branch condition due to the unknown target of p. In other words, if we use the static estimated points-to set of p, both v1 and v2 could be pointed by p, so the memory modification statement at S4 is supposed to have a chance to modify v1 or v2. At runtime, after S3 is executed, we know that v1 and v2 will not be modified through a dereference store of p at S4.

In path prediction, dynamic pointer tracking has two roles. First, if a pointer is directly involved in a branch condition, its points-to value could be used to evaluate the condition in a straightforward way. Second, even if a pointer is not directly used in a branch condition, the points-to information could remove the memory disambiguation, thus help the path prediction in another way.

2.4 Dynamic Invariant Detection

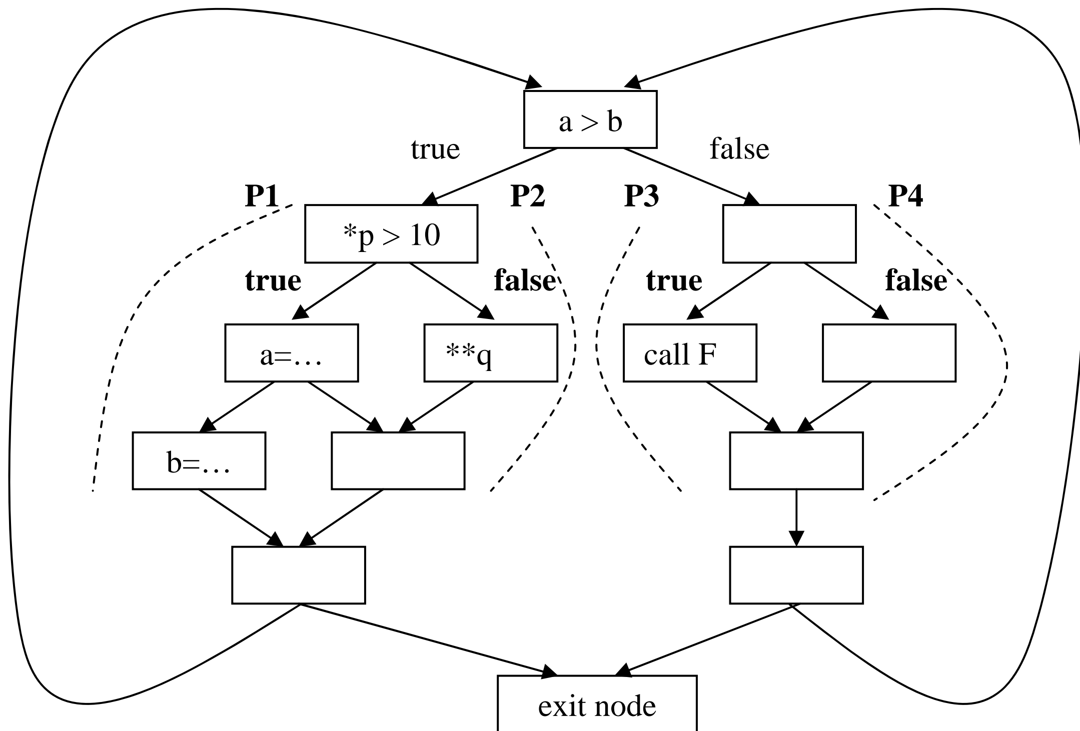


Figure 5. Dynamic invariant detection example.

Figure 5 gives an example control flow graph to show the opportunities of dead code elimination for a loop. Statically it is impossible to do branch elimination for this example due to the unknown value of *p. However, dynamically if one knows that $a > b$ evaluates to true and $*p > 10$ evaluates to false at first, and *q is not aliased with p, then the value of a, b and *p will not be re-evaluated again in the future. They become dynamic loop invariant. Therefore, the memory disambiguation is released with the dynamic pointer tracking, more variables become loop invariant.

Both of the path predication and dynamic invariant detection could be applied to some framework like program migration in mobile computing. One important constitute of application mobility is to migrate a partially executed application. Such a mechanism is especially suited for applications on partially connected mobile computers, such as laptops and palmtops. The program code and file liveness are critical in the practice of application migration. The pure static program analysis is not good enough to speculate towards programs where the program state is small. With dynamic pointer monitoring, we could do a better path prediction as stated in section 2.3, which helps to improve the precision of program reachability analysis dynamically. For the example in Figure 4, if we know that only one branch will be taken and the function will not be called any more in the future, then the code on the untaken path could be discarded during migration. Also, the dynamic invariant detection with dynamic pointer tracking could help eliminate the dead code. Like the example in Figure 5, the program always takes the path P2. The other paths P1, P3 and P4 could be removed from the loop for the future iterations. This is under the assumption that the current function will not be called again. Otherwise, the branch condition $a > b$ has to be evaluated again, and the paths P1, P3, and P4 are possible to be executed. The function call to F is not reachable under the current execution context. If there is no other function calls on F, F can be discarded too. Thus, dynamic pointer tracking could reduce the migration downtime by eliminating more dead code at runtime.

2.5 Garbage Collection

Another application of dynamic pointer tracking is garbage collection for weakly typed languages like C and C++. If the exact points-to information is available at runtime, the garbage objects could be identified quickly, and the memory allocated for them could be reclaimed immediately.

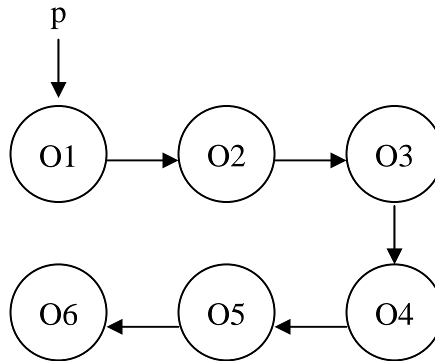


Figure 6. Garbage collection motivation example.

Figure 6 illustrates how dynamic points-to information could help collect garbage objects. There are six objects (O1, O2, O3, O4, O5, and O6) connected in a linked list in the figure. The pointer p points to the head of the linked list. Suppose p is switched from the object O1 to the object O6. At runtime, if we know which pointer is pointing to which object, then O1 could be detected as a dead object. Next, the deadness of O1 results in the pointer to the object O2 being dead. The deadness information gets propagated along the linked list until the last object O6. The garbage detection process is triggered by the pointer assignment instruction. This is the first case how dynamic pointer tracking could help garbage collection. The advantage of our scheme over reference counting is that ours does garbage collection early at the point of pointer switch and not when garbage collection is turned on later. Currently, garbage collection is done selectively at certain points in the program and infrequently since it is so expensive but this can allow garbage collection to be done throughout the program and in an incremental way. On modern multicores this is good. The second case is that even if p remains pointed to O1, but

become unreachable in the current execution context. In this case, reference counting style garbage collection does not work, but our frameworks will. The runtime unreachable code analysis is helpful in the garbage collection.

2.6 Summary

We observe that dynamic pointer tracking plays a key role in kinds of program analysis, system optimizations, and software security. However, to our knowledge, previous pointer analysis techniques could not provide precise points-to sets due to the limitations of static analysis and the lack of runtime support. In this work, we propose an efficient solution to track the pointer behavior dynamically. The implementation issues and optimization opportunities are detailed in the next chapter.

CHAPTER 3

BASIC DYNAMIC POINTER TRACKING ALGORITHMS

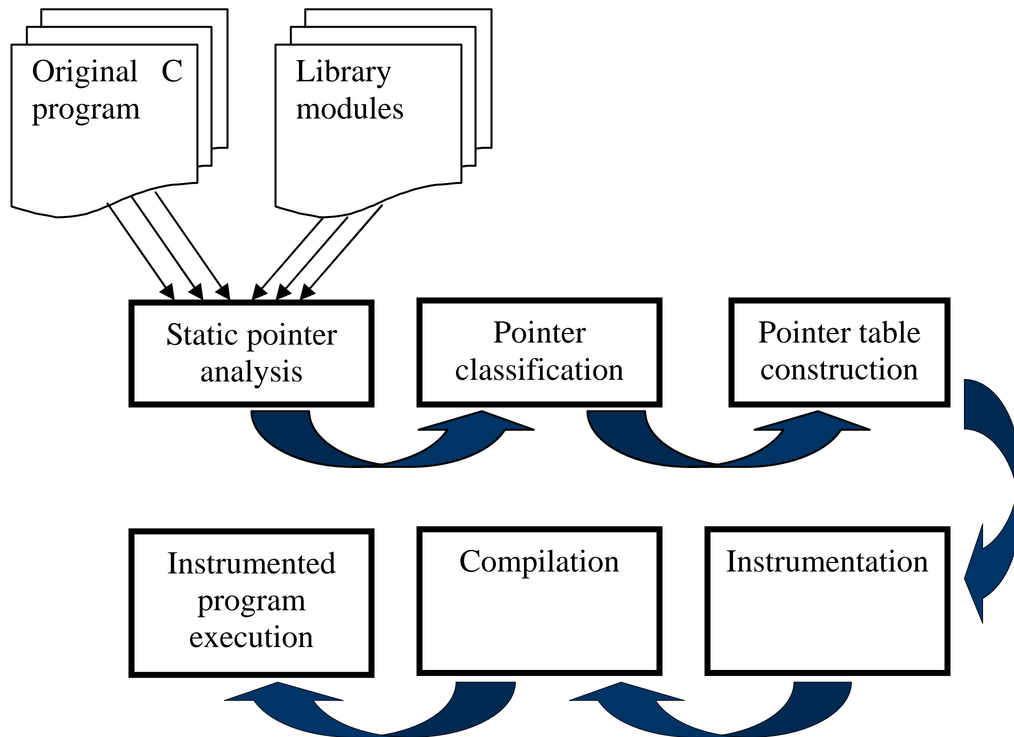


Figure 7. Static analysis framework overview.

This chapter overviews our dynamic pointer tracking technique. Figure 7 shows the static analysis framework. First, we use the pointer analysis in [60] to get the static points-to sets. In this work, we target pointers whose points-to set has more than one object. In the following, we focus our discussion on those pointers. Then, the compiler classifies the pointer into different categories based on their declaration and use. Next, the compiler builds a pointer table indicating which variable a pointer points to. Finally, the compiler instruments the program with some special instructions in the source code to update the pointer table. The instrumented program is compiled with a C compiler. At runtime, the inserted instructions are executed when the program continues, and the

pointer table contains the up to date information. In the following, we elaborate on the data structure design and the special instruction implementation.

3.1 Data Structures

Table 1. Example pointer table.

	Start addr	End Addr	Ptr Addr	Ref Ptr
	0xbeffbef8	0xbeffbff8		p1
ptr_id ⟶	0xbeffe000	0xbeffe060		null
	0xbeffe128	0xbeffe228		p2
	…			

A pointer table is a table using the pointer id as its key. A variable's type information is used to identify pointers. Every pointer has a unique pointer id that is generated by the compiler; thus the table is collision free. One example pointer table is shown in Table 1. A pointer table entry contains four fields – the start address, the end address, the pointer address, and the referent-pointer. The start address and the end address fields represent the memory locations pointed by a pointer. The pointer address indicates where the pointer itself is in memory. The referent-pointer field informs which pointer is pointed by the current pointer. It is especially designed to solve the problem of pointer chasing that is explained in section 3.4.3. The first entry of the pointer table is reserved for the NULL pointer. Note that some of the fields like the referent-pointer, and the pointer address are not indispensable in all applications. In general, the data structure could be customized based on the application domain and the customer requirements.

3.2 Special Issues on Data Structure Design

There are several special issues in building the pointer table. There are a lot of subtleties in C; it is difficult to catch them all. We mainly focus on the problems observed from our empirical study on the benchmarks. We discuss those cases and propose the special handling methods for them in the following.

3.2.1 Pointer on the Heap

There is a big challenge in the pointer table design due to the pointers existing in the heap area. Statically, we cannot know how many such pointers there are since numerous pointers are created in loops, and the number of loop iterations is not predictable. For these pointers, we cannot generate the pointer ids for them at compile time. We call this kind of pointer as *pointer on the heap*.

To solve the problem, a naïve way is to generate a pointer id dynamically after a new heap object is generated, and then creates a table entry for the pointer. However, there is a hard problem in this design. The compiler could generate a pointer id for a statically available pointer. For an assignment to such a pointer, the pointer table is directly accessed using the pointer id. However, for a dynamically generated pointer id, the compiler does not know where its entry is in the pointer table, thus it does not know how to insert the code to update the pointer's target.

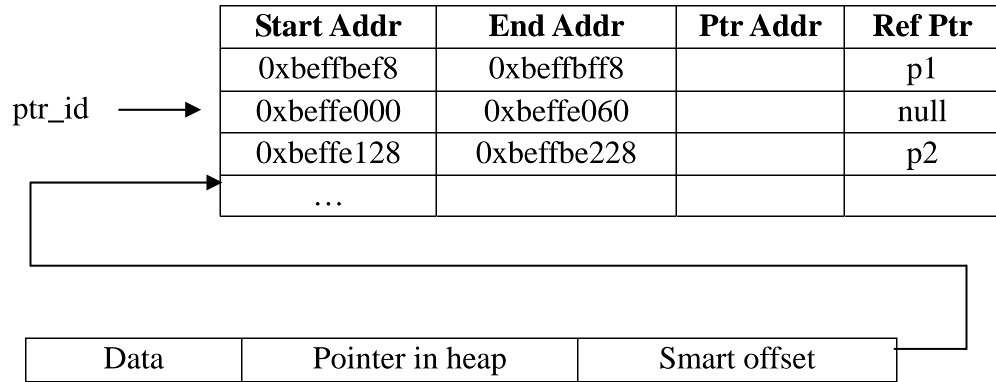
```
struct Node {
   int data;
   Node * next;
} Node;

int CreateObjs(int n) {
   Node * curr, * prev, * head;
   assert(n >= 1);
   prev = (Node *) malloc(sizeof(struct Node));
   head = prev;
   for(i = 1; i < n; i++){
      S1: curr = (Node *) malloc(sizeof(struct Node));
      S2: prev->next = curr;
         prev = curr;
   }
   prev->next = NULL;
   return head;
}
```

Figure 8. Pointer on the heap example.

Figure 8 gives one example illustrating this specific problem. In this example, there are three pointers defined in the function CreateObjs; they are curr, prev, and head. By

analyzing the code, the compiler identifies three more pointers curr->next, prev->next, and head->next. Theoretically, we have six pointers in total. However, before the explicit creation of heap objects, the three pointers on the heap do not exist, so finally the compiler only generates three pointer ids for curr, prev, and head. At runtime, assume that an integer ten is passed as the parameter to CreateObjs. Ten objects are expected to be constructed, and every object has a pointer within it. Thus, we should have thirteen pointers in total (ten pointers on the heap plus curr, prev, and head). The interesting problems are caused by the statement S1, where the malloc function creates a new object, and the new heap object has a pointer within it. At the statement S2, the pointer is assigned a new value. Originally, the compiler inserts special instructions to update the pointer's target. Without the information regarding the pointer's corresponding entry in the pointer table, the compiler cannot proceed doing this. The conclusion from studying the example is that we have to generate the pointer ids for those pointers and map the table entries for them dynamically.



Data layout

Figure 9. Smart offset for pointer on the heap.

The next question is how we use the dynamically generated pointer ids to update the pointer targets. The mapping between pointers and their entries in the pointer table poses challenges. One solution is to add one field in the pointer table recording the address of the pointer; then an assignment to a pointer on the heap triggers a search in the pointer

table and extracts its entry. It is inefficient and could result in huge overheads. To solve the problem, we design a smart offset. A smart offset extends the normal pointer on the heap by adding an offset indicating its offset in the pointer table as shown in Figure 9. If there is an assignment to a pointer on the heap, then its smart offset is used to update its entry.

3.2.2 Recursive Function

```
typedef struct lateral {
    …
} *Lateral;

Lateral build_lateral(int i, int num) {
    register Lateral l;
    ...
    // create a pointer table entry for l
    if (num == 0) return NULL;
    l = (Lateral) ALLOC(0,sizeof(*l));
    S1: next = build_lateral(i,num-1);
    …
    // erase the pointer table entry of l
    return l;
}
```

Figure 10. Recursive function example in power.

Recursive functions pose similar problems as pointers on the heap since all functions called recursively share the same pointer ids for their local pointers. This problem is solved in a similar way. First, the compiler builds the call graph and detects the cycle in the call graph. Then for those pointers in cyclic functions, the compiler inserts the code to create their pointer ids dynamically. One example is shown in Figure 10. At the entry of the recursive function build_lateral, an entry is created for the pointer l. At the function exit, l’s entry is erased from the pointer table.

3.2.3 Explicit and Implicit Pointer Array

A pointer array could be accessed through a variable subscript. Even if an array is statically declared, which pointer in it will be assigned or dereferenced is unknown. For this case, we use the index plus the array's first pointer's id to get the corresponding pointer table entry.

```
typedef struct hash {
  HashEntry *array;
  int (*mapfunc)(unsigned int);
  int size;
} *Hash;

typedef struct hash_entry {
  unsigned int key;
  void *entry;
  struct hash_entry *next;
} *HashEntry;

Hash MakeHash(…) {
  Hash retval;
  int i;
  …
  retval = (Hash) localmalloc(sizeof(*retval));
  //initialize smart offset array
  S1: retval->array = (HashEntry *) localmalloc(size*sizeof(HashEntry));
  for (i=0; i<size; i++)
  S2:   retval->array[i]=NULL;
  retval->mapfunc = map;
  retval->size = size;
  S3: return retval;
}
```

Figure 11. Implicit pointer array example in mst.

Although a pointer is declared as a pointer, it could be used as a pointer array. Figure 11 gives such an example In this example, a pointer "array" is declared in the structure hash. In the function MakeHash, an index i is used to access the pointer hash->array at the statement S2. In our prototype, the pointer hash->array's bounds are transferred to the pointer ent->next. According to the program's semantics, the pointer ent->next's access range is different from hash->array. Driven by this scenario, we determine to handle this based on the program semantics and the type system. If there

exists a subscript to access a pointer and the pointer is allocated using malloc(size*sizeof(type)) like function, where type is a pointer type, then the pointer is determined as a potential array. When the pointer is initialized at the statement S1, every pointer in the array has a smart offset allocated and initialized. Thus, in this case, the bounds information is well stored at a fine level.

3.2.4 Pointer Parameter and Pointer Return

Pointers passed as parameters have to be dealt with carefully. Some functions are not available at compile time. On the other hand, a function could be called by different callers, and different pointers are passed as the parameters. In those scenarios, the pointer id is not available for a pointer parameter. To solve the problem, the compiler modifies the function call interface, and transfers the pointer id together with the pointer parameter as shown in Figure 12.

```
void foo(int * ptr) {
    …
}
// pass the pointer id with the pointer parameter
void foo(int * ptr, int ptr_id) {
    …
}
```

Figure 12. Pointer parameter example.

A pointer used as the return value of a function has a similar problem as pointer parameters. Different pointer could be returned. How to tell the caller which pointer is returned is a problem. We push the return pointer id to the stack together with the return pointer. The function's caller uses the pointer id on stack to get the return pointer's information. Consider the example in Figure 11, the pointer retval's pointer id is returned with it at the statement S3. A side effect of passing and returning the pointer id is to prevent the program from stack smashing [66], which replaces the return address field of

an activation record with the address of a malicious code. Depending on the register pressure, the return pointer id can also be passed through registers.

3.2.5 Pointer Type Casting

How to handle type casting really depends on the application domain. For example, in the garbage collection work, to be conservative, we take the subtype's bounds for both upcasting and downcasting. Also in the memory protection work, to avoid false alarm, both upcasting and downcasting take the bigger bounds.

Most of the time, the type casting is to use a temporary variable to hold an intermediate result, and is finally the data is converted back to the original pointer type data. One example is shown in Figure 13. In this example, two pointers are cast to integers and assigned to f_left.value and f_right.value. Then the two integers are converted to the same pointer type (i.e. HANDLE *) data again, and assigned to two pointers h->left and h->right. The update code is originally inserted after every statement. For this case, the propagation optimization described in section 3.7.6 removes the update at S1 and S2 to get rid of the update for temporary variables. Thus, h->left and h->right are type consistent with the result returned from RandTree.

```
HANDLE* RandTree(int n, int seed, int node, int level){
   future_cell_int f_left, f_right;
   HANDLE *h;
   …
   if (n > 1){
         S1: f_left.value=(int) RandTree((n/2),seed,newnode,level+1);
         S2: f_right.value=(int) RandTree((n/2),skiprand(seed,(n)+1),node,level+1);
         S3: h->left = (HANDLE *) f_left.value;
         S4: h->right = (HANDLE *) f_right.value;
      }
   …
   return(h);
}
```

Figure 13. Pointer type cast example in Bisort.

There is no problem if a pointer is cast to an integer. However, casting an integer to a pointer causes problems. If the pointer is used as a temporary variable, only involved in arithmetic computation and there is no dereference of it, then it could be optimized by copy propagation as we show later. Otherwise, it incurs two issues. First, the integer's pointer table entry is unknown. This can be handled with a smart offset. Second, the integer's access range is unknown. There are several solutions for the second problem. We could use a fake pointer's type and value to update its bounds, or the user could provide annotations indicating how big the integer pointer's target is, or we skip updating the pointer's target.

3.2.6 Function Pointer

One of the C language specific features is the use of function pointers. A function pointer bears type casting. Function pointers often bear type casting, and most of the time they do not have accurate types. Misuse of function pointers could lead to severe program bugs. If a program has an indirect function call, an attacker could tamper the function pointer with the address of the malicious code, and a subsequent call through that function pointer will transfer the control to the attacker.

In this work, we take function pointers as a special kind of pointer. Every individual function is an object. The target of a function pointer is a function. Type casting on a function pointer does not enlarge its access range like downcasting on other pointers. An update for a function pointer assignment depends on the function that it points to.

3.2.7 Unions

Pointer in a union is a special issue. Figure 14 shows one example that has a pointer in a union. The integer type data i in the union a is assigned with a constant number at the statement S1. Then, the pointer type data p is dereferenced at the statement S2. Our scheme inserts the update instruction at S1. Some work like CCured does not support such a declaration. They require that the elements in a union type must agree on types,

say the elements must be all pointers or all non-pointer type data. To reduce the number of rules that would restrict the customers from writing code in the original C language, we decide to allow the user mix pointer type data with non-pointer type data in a union, as long as they agree on the data layout. In some applications like garbage collection, an object could be potentially pointed by the pointer in a union. To be conservative and to be safe, we assume that if a pointer is declared in a union, the assignment to any field in the union triggers pointer update. Thus, we will not collect a live object.

```
Union{
    int i;
    int * p;
} T;
Foo(){
    T a;
    S1: a.i = 0xabcdabcd;
    S2: *(a.p) = …
}
```

Figure 14. Union example.

3.3 Data Structure Management

How to organize the table is interesting due to the dynamically created pointers on the heap and due to the recursive functions. The pointer table size could be increased and decreased at runtime. In the fowling, we discuss how the pointer table is managed.

The first problem is caused by pointer on the heap. There are two types of updates for a pointer on the heap. When a pointer on the heap is created, an entry is constructed. When a pointer on the heap is freed, its entry is destroyed. To prevent the pointer table from fragmentation, we must remember which pointer table slots are allocated and which are freed. In other words, for a new entry allocation, the pointer table has to be scanned to find an available slot. It has been empirically observed that in many applications, the most recently created objects are also those most likely to quickly become dead (known as infant mortality or the generational hypothesis). So we could just record the last entry

in the pointer table as last_entry. If the last entry is disabled; then last_entry is decreased by one. New entry creation request increases last_entry by one. This is a tradeoff between performance and space cost. In this work, the empirical study motivates us with scarifying the space for performance. Note this solution could leave some unused table entries, but it does not impact the correctness of dynamic pointer tracking. The table could be defragmented to get rid of the pointer table holes when necessary.

```
HANDLE* RandTree(int n, int seed, int node, intlevel){
  HANDLE *h;
  …
  if (n > 1) {
       …
       S1: NewNode(h,next_val,node);
       …
       S2: h->left = RandTree((n/2),seed,newnode,level+1);
       S3: h->right = RandTree((n/2),skiprand(seed,(n)+1),node,level+1);
    }
```

(a)

	Start Addr.	End Addr.	Ptr Addr	Ref Ptr
h1				
h1->left				
h1->right				
h2				
h2->left				
H2->right				

(b) Pointer table

	Start Addr	End Addr	Ptr Addr	Ref Ptr
h1				
h2				

(c) Static pointer table.

	Start Addr	End Addr	Ptr Addr	Ref Ptr
h1->left				
h1->right				
h2->left				

(d) Heap pointer table

Figure 15. Pointers on the heap and in recursive functions example in bisort.

The second problem is caused by the pointers defined in recursive functions, which appear alternatively with pointers on the heap. One example function RandTree in the

olden benchmark [88] bisort program is shown in Figure 15 (a). In the function, a pointer h is defined in the recursive function. H is initialized at the statement S1, and two pointers on the heap are created, i.e. h->left, h->right. If the function RandTree is called, a new entry is created for the pointer h. When the statement S1 is executed, two entries are created for the pointers h->left ad h->right. Upon RandTree calling itself, the same process happens again. The pointer table looks like Figure 15 (b), where h1 means the pointer h in the first RandTree's record, ad h2 means the pointer h in the second RandTree's record. When a recursive function returns, its local pointers should be erased from the pointer table. In this case, the two entries in the pointer table for h1 ad h2 are erased. One could see that it leaves some holes in the pointer table that is not desired.

In favor of recursive functions and heap objects, we decide to split the table in two parts. One is the static pointer table containing the pointers declared statically. The other one is the heap pointer table containing the pointers on the heap. The two example pointer tables are shown in Figure 15 (c) and (d). We use two last entry variables (i.e. slast_entry and hlast_entry) to manage the two tables. Dividing the pointer in two parts, we could remove the entries for the pointers in recursive functions easily without leaving the pointer table with fragments.

The difference between our approach and fat pointer in the data structure design is that the points-to information is centralized in one data structure. Most of applications like the multimedia programs and the compressor/decompress do not deploy a lot of pointers on the heap. Only a few smart offsets are generated in those programs. Even if for some heap pointer intensive applications, the heap pointer entries could be garbage collected in a similar way as for other objects. The pointers in the static pointer table are treated as the root pointers, and the heap pointer table is scanned to find the live pointers. In this way, the pointer table entries of those dead pointers could be destroyed and the pointer table could be defragmented.

3.4 Update Instructions

There are three types of positions where a pointer's target is updated. They are at the function entry, when a heap object is allocated or de-allocated, and when a pointer is assigned through other pointer evaluation. Currently, the update instructions are proposed at the pure software level; they could be ameliorated with some hardware support or through specially designed instructions. We discuss the different update code for those three cases in the following.

3.4.1 Function Entry and Exit

Statically, all fields in the pointer table are initialized to zero. At a function entry, the local pointers entries are cleared. We update the pointer table in a lazy way. At the function exit, the local pointers entries are not voided until the next time the same function is called, and its table entries get initialized. For a recursive function, new entries are created and destroyed at the function entry and exit, and last_entry gets updated.

3.4.2 Malloc and Free

There are two cases when a heap object is created through malloc-like function calls. If the newly generated heap object has no pointer in it, then the update instructions are simple. The start and end addresses are set according to the pointer's type and its value. If pointers exist in the new object, then new entries are created in the heap pointer table for them. As discussed in section 3.3, for a free operation, if the pointer has the last entry in the pointer table, the last entry is updated. Otherwise, the table entry is cleared, and the table size is not changed.

C developers sometimes rewrite the malloc and free functions to manage the memory pool by themselves. Consider the example in Figure 16. The developer allocates a big memory pool with a single call to malloc. An allocation for a new object calls

mymalloc1 which returns a part of mempool. According to the program semantics, the pointer dereference at the statement S3 violates the access rule. In our baseline, at the statement S1, the pointer p is enabled to access the whole mempool due to the pointer casting (described in section 3.2.5). The out of bounds dereference is not detected when the program is monitored. Our work is based on the semantics of malloc and free. User manipulated malloc and free like this is out of our control; it is managed at the developer level. Without knowing the program semantics, the parameters 100 and sizeof(int) mean nothing. Suppose we use the parameters to guess that p's target should be within 400 bytes, then the function call to mymalloc2 should also set the pointer q's target with the parameters, but it is not supposed to do so. Therefore, take the typecasting in a conservative way like ours is necessary.

```
char *mempool = (char *)malloc(sizeof(POOL_SIZE)

void *mymalloc1(int size){
   return (void *)mempool[index+size];
}

void *mymalloc2(int size){
   int *x;
   if(size < MIN_SIZE)
      return NULL;
   else return x;
}

Foo(){
   S1: int *p = (int *)mymalloc1(100*sizeof(int));
   S2: int *q = (int *)mymalloc2(20*sizeof(int));
   S3: p[100] = 25;
}
```

Figure 16. User redefined malloc example.

3.4.3 Pointer Assignments

The compiler inserts update instructions right before every pointer assignment and de-allocation statement to keep the points-to sets up to date. The points-to sets are

updated before a pointer assignment since some pointer is set to NULL. For example, the statement “p = p->next” makes p to NULL; p->next information is missing after the evaluation. The pointer assignment statements and the update code for them are shown in Table 2. We discuss them on a case by case basis.

The first pointer assignment case is through pointer arithmetic. The address is computed based on the operands (e.g. p and r) addresses. The operation (e.g. op in “q = p op r”) could be add, subtract etc. How to deal with the out of bounds pointer computation has different options. A pointer goes out of bounds if its value is not between the start address and the end address. In the memory protection work, if an out of bounds pointer is not dereferenced, it will not harm the program. For example, an out of bounds pointer could become within bounds later through pointer arithmetic, and it could also be used to get the object pointed by other pointers. For these scenarios, going out of bounds does not indicate a program error. In the pure dynamic pointer tracking algorithm, we keep the out of bounds pointers. Some applications could decide to trigger an alert upon an out of bounds pointer dereference or upon an out of bounds pointer calculation. The second case is the statement like “q = &v”, where v is a normal variable, not a pointer. The address and the size of v are known after the assignment is executed. The target of q is filled using the information of v. The third case is the statement like “q = &p”, where p is a pointer. The target address of q is filled using p. In this case, the pointer q is directed to the pointer p; thus the ref_ptr of q is set to p. The forth case is the statement like “q = *p”, where the pointer p points to another pointer r as shown in Figure 17. After the statement “q = *p” is executed, q’s target is equal to r’s. P’s referent pointer field is used to get r’s table entry. The last case is through pointer dereference statement like “*q = p”, where the pointer q points to another pointer p. In this case, q’s referent pointer is updated to p. The last two rows in Table 2 are used to demonstrate how complex pointer chasing cases like “q = **p” and “**q = p” (both p and q are pointers) could be dealt with using the address and referent pointer information.

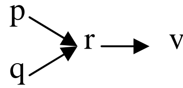


Figure 17. Pointer chasing example.

Table 2. Updating code for pointer assignments.

Pointer assignment	Updating code
q = p op r	q->start_addr = p->start_addr q->end_addr = p->end_addr
q = &v	q->start_addr = v.addr q->end_addr = q->start_addr + sizeof(v)
q = &p	q->start_addr = p.addr q->end_addr = p.addr + sizeof(p)
q = *p	q->start_addr = p->ref_ptr->start_addr q->end_addr = p->ref_ptr->end_addr
*q = p	q->ref_ptr->start_addr = p->start_addr q->ref_ptr->end_addr = p->end_addr q->ref_ptr = p.id
q = **p	q->start_addr = p->ref_ptr->ref_ptr->start_addr q->end_addr = p->ref_ptr->ref_tr->end_addr
**q = p	q->ref_ptr->ref_ptr->start_addr = p->start_addr q->ref_ptr->ref_ptr->end_addr = p->end_addr q->ref_ptr->ref_ptr = p

3.5 Limitations of the Work

3.5.1 Compatibility with Legacy Code

Program modules with dynamic pointer tracking must coexist with program modules without dynamic pointer tracking. Any realistic design for dynamic pointer tracking must allow libraries written without dynamic pointer tracking. A project using dynamic pointer

tracking is very likely to use libraries written in another project, and it is insane to expect that all other libraries would be processed with dynamic pointer tracking or the library source code is available. Making pointers tracked in all code including existing libraries is not reasonable. Those libraries may not be transformed with dynamic pointer tracking, and there may be not necessary to use dynamic pointer tracking.

The impact of coexistent problems is that there is no check and update on the untracked libraries and program components. If the returned pointer from such libraries or program components bears type casting, the points-to information is not accurate. If a pointer on the heap or in a recursive function is passed to an uninstrumented module, the metadata in it has to be stripped. In this work, if a pointer's value comes from a pointer whose points-to information is unknown, its value and base type information is used to do the update. This is under the assumption that the use code does reliable type casting in those cases.

3.5.2 Separation of Space

A pointer table is declared as a global data. It could be stored in the reserved space. A program should not change the space and the smart offsets. If it did, say by overwriting the pointer table used to maintain the points-to information, the pointer tracking scheme might accidentally use the wrong data.

3.5.3 Visibility of Pointers and Pointer Assignments

All pointers should be visible at static time. The declaration of pointers could be used to identify pointers. In other words, a pointer's offset in an object and its declared type are statically available.

A dynamic pointer tracker has to know where the pointers are in a program, and also which kind the pointers are, so that it could determine which pointer should be put in which pointer table. For some hidden pointers, say, they are declared as integers, but they

are cast to pointers and dereferenced, the tracker is fooled into considering the variable as regular integer and does not create an entry for it.

All pointer assignments must be visible to the dynamic pointer tracker. That is, if a pointer value is changed, it must be through a pointer assignment that having both the left hand value and the right hand value as the pointer type. Changing pointers through some obscure assignments like memcpy would conceal the assignment from the tracker, perhaps causing it to do the wrong update.

For the purposes of maintaining the pointer table's safety, consistency and integrity, we assume that all programs are written with a single thread. Multi-threaded programs could be supported by adding locks to the pointer tables.

3.5.4 Setjmp/Longjmp

If a program uses a setjmp/longjmp pair, the program's call stack is broken. In this case, the developer has to take risks to use dynamic pointer tracking mechanism since the pointer table integrity may be not correct.

3.6 User Support

Dynamic pointer tracking is supposed to be applied to any C program. However, the program developer could also help the dynamic pointer tracking framework in different ways. In the following, we describe how programs could be written to improve dynamic pointer tracking.

To improve the precision of dynamic pointer tracking. It will be helpful if the user could avoid type casting like from integer to pointer, or if the user could provide annotations at the type casting places indicating how to update the pointer's bounds. In this way, the pointer's target could be tracked more precisely due to the already known program semantics. In favor of dynamic pointer tracking, it is better to follow the C

idioms like not breaking the known semantics of malloc/free, not returning a pointer to a stack object.

To help the pointer table management. The static pointer table grows due to the pointers defined in recursive functions. The heap pointer table size changes due to the pointers on the heap. If the user could transfer the recursive functions to non-recursive functions, and prefer using static pointers than pointers on the heap, then the static pointer table's size could be predicated in advance, and the heap pointer table's size could be better controlled. Reducing the number of pointers on the heap also helps garbage collection since it will reduce the number of memory scans for live pointers on the heap. If the user could follow the infant mortality rule, and destroy the most recently generated object first, the heap pointer table will have fewer fragments.

To reduce the update overheads. Fewer pointers in recursive functions could eliminate the overhead to create and erase the pointer table entries. If a parameter pointer points to a small object, the user could pass it by reference. On the other hand, the user could choose a set of pointers to be tracked like the tainted data through the user input. The user could also select the places where pointer update is not necessary, and the bounds information could be collected using the pointer’s value and type.

3.7 Possible Optimizations

The goal of this work is to dynamically track the pointer targets, and provide the points-to information as efficiently as possible. Our basic scheme could keep full track of the pointers’ dynamic behavior. However, the large overheads hinder this technique from being practical. How to remove the redundant updates and to reduce the update time are the main concerns of this section. In the following, we first introduce some concepts, and then describe the optimization details.

3.7.1 Pointer Definition and Use

Here are some terms, common to the pointer analysis literature and program analysis.

Definition 1: A pointer *definition statement* is a statement that could modify a pointer's value.

Definition 2: A pointer *dereference statement* is a statement that uses a pointer target's value.

Definition 3: A pointer *use statement* is a statement that uses the pointer's value.

Definition 4: If a statement S defines a pointer and a statement T has a dereference of the pointer, then T is called S's *dereference* statement.

Definition 6: A pointer definition statement S (defining a pointer p) is *reachable* to its dereference statement T there exists a path from S to T if in the control flow graph, and there does not exist other pointer definitions of the same pointer between S and T.

Table 3. Pointer definition, dereference, and use statements.

Definition of q	Dereference of q	Use of q
q = malloc()	v = *q	p = q op r
free(q)	v = **q	
q = p op r	*q = v	
q = &v	**q = v	
q = *p		
q = **p		

Table 3 shows the classification of the definition, dereference, and use statements for a given pointer q. The definition statements shown in the first column are pointer assignments in Table 2 excluding the two statements "*q = r" and "**q = r" since they are classified as dereference statements. The dereference statements are shown in the second column. A dereference could be at either the left side or the right side of a statement. In the dereference statement, "v" represents a normal variable or a pointer. The use statement set contains the pointer arithmetic statement.

3.7.2 Remove Redundant Pointer Chasing Update

There are many positions where a pointer's points-to set could be updated. Some updates need not be dynamically committed at the point where they occur, but can be postponed until the underlying dereference actually needs them – thus, an update could be lazily postponed until the point of use is actually reached. Since in the dynamic sense an update may not reach a dereference point, and may be superseded by another one, redundant and useless updates get factored out.

```
S1:   p = &q
S2:   q = &r
S3:   x = **p
```

Figure 18. Redundant update example.

The redundant update removal algorithm removes redundant updates by identifying the necessary updates that are reachable to a pointer use or dereference, and thus other updates are removed from the baseline. The algorithm is not like the standard dead code elimination due to the pointer chasing cases. For example, in the code segment shown in Figure 18, the pointer p is dereferenced at the statement S3, and it is defined at the statement S1. The interesting issue is that the statement S2 assigns *p to r, and thus the nearest definition of **p is S2. The traditional definition and use analysis proceeds backwards along the control flow graph searching for p's definitions, and thus it could not detect the nearest definition at S2. If a pointer definition exists between a pointer chasing definition and its dereference, it is possible to kill the pointer chasing definition. Whether or not the pointer chasing definition is killed depending on the dereference level and the pointer chasing level. We show the detailed algorithm in the following.

The algorithm is shown in Figure 19. First, a pointer definition and its reachable dereference are identified. The function Identify_Ptr_Def_Deref_in_BB in Figure 19 (a) enrolls a pointer definition's reachable pointer dereference in a given basic block to its dereference list. In the algorithm, the dereference is checked first since a statement could

have both a pointer definition on the left side and a pointer dereference on the right side. If this is the case, and the definition is checked first, then the function returns without enrolling the reference to the dereference list. The function returns false if another statement kills the pointer definition; otherwise, it returns true. The function Identify_Ptr_Def_Deref in Figure 19 (b) collects a pointer definition's reachable dereference in the control flow graph. It starts from a pointer definition statement, looking for its dereference in the same basic block, then it pushes its basic block's predecessors on the stack to further search for its dereference. The search step terminates until it finds another definition along the control flow graph backwards. All dereferences of a pointer definition are stored in the dereference list.

```
//input: BB – basic block
//         S – pointer definition statement
//         p – pointer

Identify_Ptr_Def_Deref_in_BB (BB, S){
   if(S.get_BB() is BB)
      End_Statement = S;
   else
      End_Statement = BB.get_end_statement();
   for (T from End_Statement to BB.get_start_statement()){
     if (T has a dereference of p)
          S.deref_list.insert(T);
     if(T is a pointer definition statement of p)
        return true;
   }
   return false;
}
```

(a)

```
//input :CFG – control flow graph

Identify_Ptr_Def_Deref(CFG){
   for (BB in CFG)
      for (S in BB){
         if (S is not a pointer assignment)
            continue;
         Cur_BB = S.get_BB();
         if (Identify_Ptr_Def_Deref_in_BB(Cur_BB, S, S.LHS) is true)
            continue;
         for(BBP in Cur_BB.get_pred_BBs()){
             Stack.push(BBP);
             BBP.set_visited();
         }
         while(!Stack.is_emptry()){
            Cur_BB = Stack.pop();
            if(Identify_Ptr_Def_Deref_in_BB(Cur_BB, S, S.LHS) is true)
               continue;
            for(BBP in Cur_BB.get_pred_BBs()){
                if(BBP.is_visited())
                   continue;
                BBP.set_visited();
                Stack.push(BBP);
            } // end for
         } // end while
      } // end for
   } // end for
}
```

(b)

Figure 19. Algorithm to identify pointer definitions and their reachable dereferences.

Whether or not a pointer chasing assignment is redundant depends on if there exists a dereference statement reachable from the pointer chasing assignment, whose dereference level is less than the pointer chasing level. A dereference level is how many levels a pointer is dereferenced, which could be got by calculating the * keyword. A chasing level means how many levels a pointer is chased by another, which could be got by calculating the & keyword appearing in the pointer definition and the pointer chasing definition statements. No matter where a reachable pointer dereference is (before or after the pointer assignment), the pointer chasing definition is not redundant since its

information is used by the dereference. The main algorithm for redundant update identification is shown in Figure 20.

```
// input: ptr_chasing_assign_list – pointer chasing assignment list
//           ptr_assign_list – pointer assignment list
//           deref_list – a pointer's reachable dereference statement list

Redundant_Update_Remove(ptr_chasing_assign_list, ptr_assign_list,
   deref_list, post_dom_tree){
   for (S in ptr_chasing_assign_list)
      for (T in ptr_assign_list){
         if ((S is not T) && (T post-dominates S) && (RHS of S is LHS of T)){
            find_deref = false;
            for(D in S.deref_list())
               if(deref_degree(D) < chasing_degree(S,T)){
                  find_deref = true;
                  break;
               }
            if(!find_deref)
               S.set_redundant();
         }//end if
      }//end for
   }//end for
}
```

Figure 20. Algorithm to identify redundant pointer definition.

3.7.3 Delay Update out of Loop

The third technique targets loops. For example, Figure 21 shows a code segment from the twolf benchmark in the SPEC2000 benchmark set. The pointer assignment at the statement S1 is in a loop. There is only one definition of the pointer cellaptr in the loop. The pointer binptr[ablock][abin]->cell is never modified within the loop. Though the target of cellaptr is changed frequently within the loop, the object pointed by cellaptr is always live since it is pointed by other pointers like binptr[ablock][abin]. Thus, this observation inspires us with postponing the updating until the loop is finished. The original update code is postponed as shown in Figure 21. This optimization could be

applied to some framework like garbage collection, where an object's live range is the key.

```
uloop2(){
   while( attempts < attmax ) {
      a = PICK_INT( 1 , numcells ) ;
      acellptr = carray[a]    ;
      if( acellptr->cclass == -1 ) {
        continue ;
      }
      ablock      = acellptr->cblock ;
      ablckptr = barray[ ablock ] ;
      axcenter = acellptr->cxcenter ;
      aorient = acellptr->corient ;
      abin = SetBin( axcenter ) ;
      S1: cellaptr = binptr[ablock][abin]->cell ;
      for( i = 1 ; i <= *cellaptr ; i++ ) {
        …
      }
   }
   //update cellaptr's target
}
```

Figure 21. Function uloop2 in Twolf.

The conditions for this optimization on a loop are

(1) The pointer is defined using one source in the loop.

(2) The pointer definition source is not changed in the loop (including change through pointer chasing and dereference).

(3) The pointer definition source is available out of the loop.

(4) The pointer and the pointer definition source are not involved in a function call in the loop.

(5) The loop does not have return or goto statement.

The algorithm processes every pointer assignment in a loop. If the right hand side of the pointer assignment is not changed, then a candidate is identified. In other words, if the right side is not redefined, then the left side pointer' target remains live; and the update could be delayed out of the loop. If the pointer is dead out of the loop, the update could

be removed. If inter-procedure analysis and function inlining are supported, the restriction (4) could be relieved. The candidate's invariant information gets propagated. For example, there are two pointer assignment "p = q" and "q = r". Although q is redefined, if r remains invariant, both p and q are both loop invariants. The other conditions make sure that the pointer's target is still available and remains the same out of the loop. Finally, the algorithm terminates until it gets a transitive closure of the loop invariant pointer assignments.

3.7.4 Delay Update for Special Patterns

The second redundant update detection algorithm depends on the recognition of computation patterns. Through our experiments, we observe that a lot of pointers are used for the implementation of data structures like linked list, trees. To manipulate the nodes in such data structures, the linked list or trees are often visited.

```
struct Results get_results(struct Village *village, int village_ptr_id){
   struct List              *list;
   struct Patient           *p;

   list = village->returned.forward;
   while (list != NULL) {
      //update p's target
      S1: p = list->patient;
      …
      //upate list's target
      S2: list = list->forward;
   }
}
```

Figure 22. Linked list example 1 in Health.

One example is shown in Figure 22. For this example, our basic dynamic pointer tracking scheme inserts the updates for both statements S1 and S2. S2 is used to traverse the linked list. After the loop, the pointer list is neither defined nor used. The object live ranges are not impacted by the pointer assignment at S2. Thus, removing the update code before S2 shows little influence on dynamic pointer tracking, but it shows significant

savings in update overhead. The algorithm that delays update out of loop cannot be applied to this case since "list = list->forward" is a self definition that breaks the second rule.

```
void addList(struct List *list, struct Patient *patient) {
   struct List *b;

   while (list != NULL) {
      S1: b = list;
      S2: list = list->forward;
   }
   list = (struct List *)malloc(sizeof(struct List));
}
```

(a)

```
void addList(struct List *list, struct Patient *patient) {
   struct List *b;

   if(list != NULL){
      if(list->forward != NULL){
        while (list->forward->forward != NULL) {
            b = list;
            list = list->forward;
        }
       //update b's target
        b = list->forward;
       list = NULL
      }
      else{
       //update b's target
        b = list;
       list = NULL;
      }
   }
  list = (struct List *)malloc(sizeof(struct List));
}
```

(b)

Figure 23. Linked list example 2 in Health.

The difficulty for this solution is that simply deleting the update loses some of the points-to information. For example, in Figure 23, the pointer b's target comes from the pointer list's information. If we remove the update for the statement S2, then the update for b at the statement S1 is not correct. To solve the problem, we transform the while loop as shown in Figure 23 (b). The last iteration of the loop is peeled off so that the update could be inserted out of the loop. On the other hand, if the pointer list is used outside of the loop, the loop also has to be transformed in the similar way. Currently, we only consider one layer peeling. In other words, if there is another statement using b's value, we choose not to optimize the update code. Another observation is that most of the code following the pattern bears no type casting. It allows us to use the pointer's type and value to update its target after the loop, and loop peeling is not necessary.

The algorithm is shown in Figure 24. The major things to consider in the pattern recognition are the availability of the pointer's target information and the liveness of the pointer's target. It restricts the other pointer definitions not to impact the whole list of objects liveness and makes sure that the pointer's target can be seen out side of the loop. First, the loops matching the pattern are identified and the potential pointer assignments are collected in the pointer assignment list. Then, the pointer assignment list is processed to determine if the following conditions are satisfied.

(1) The loop condition is testing if a pointer p or a pointer field of a pointer p is NULL.
(2) In the loop, the pointer p is defined through a statement like p = p->field.
(3) In the loop, the pointer p is only defined once.
(4) The other pointer assignments in the loop must use p as the right hand side.
(5) There is no return or goto statement in the loop.
(6) P is not involved in a function call in the loop.
(7) If p is used to define another pointer q, and q is live out of the loop, then the loop is peeled.

```
//input: ptr_assign_list – pointer assignment list
//          loop information

Delay_Update_for_Special_Pattern{
   for(S in ptr_assign_list){
       p = S.LHS;
       if((L = S->get_loop()) is NULL) || !match_entry_pattern(L->get_entry()
           || (L has return or goto statement) || !match_inner_patttern(S)
           || (L has function call with s as a parameter)
           || ((L has other pointer assignment T) && (T.RHS is not p)
           || (L has more than one definition of p))
           continue;
      condition_satisfied = false;
      for(T in L){
         if(RHS of T is p)
            use_list.insert(T);
      }
      if(!use_list.is_empty()){
         condition_satisfied = true;
         for(T in L){
            for(U in use_list){
               if(RHS of U is LHS of T){
                  condition_satisfied = false;
                  break;
               }
            }//end for
            If(!condition_satisfied)
               break;
         }//end for
         if(condtion_satisfied){
            peel_loop(L);
         }
      }//end if
      else
         S.set_update_delay();
   }end for
}
```

Figure 24. Algorithm to delay update for special patterns.

3.7.5 Remove Update for Pointer Initialization

The update code for initializing a pointer on the heap to null is redundant. This owes to our pointer table design. Initially, all fields in the pointer table are zero. If the last entry

of the pointer table is deactivated, its fields are set zero. For example, a pointer is freed; then its corresponding entry in the pointer table is erased. Thus, it is not unnecessary to initialize the entry as done in the baseline. One example is shown in Figure 25. In the example, when a new heap object is created, its four pointer fields are initialized to NULL. The function is frequently called by the benchmark program. Updating the pointer target on the if branch is not necessary. The algorithm to remove the update for initializing a pointer on the heap to null is shown in Figure 26. The condition for this type of optimization on a loop is There is no other pointer assignment that redefines the pointer between the function entry and the pointer assignments.

```
QuadTree MakeTree(int size, int center_x, int center_y, int lo_proc,
                  int hi_proc, QuadTree parent, ChildType ct, int level) {
   int intersect=0;
   QuadTree retval;
   retval = (QuadTree) malloc(sizeof(*retval));
   retval->parent = parent;
   retval->childtype = ct;

   intersect = CheckIntersect(center_x,center_y,size);
   size = size/2;
   if ((intersect==0) && (size<512)){
         retval->color = white;
         retval->nw = NULL;
         retval->ne = NULL;
         retval->sw = NULL;
         retval->se = NULL;
      }
   …
   return retval;
}
```

Figure 25. NULL initialization example in Perimeter.

```
//input: ptr_assign_list – pointer assignment list

Remove_Update_for_Pointer_Initialization{
   for(every null pointer assignment S in ptr_assign_list){
      find_prev_def = false;
      for(T in ptr_assign_list){
         if((T is not S) && (reachable(S, T, CFG)){
            find_prev_def = true;
            break;
         }
      }
      if(!find_prev_def)
         S.set_redundant_null_initialization();
   }
}
```

Figure 26. Algorithm to remove update for pointer initialization.

3.7.6 Propagate Update

Through the study of the applications, we find that a lot of pointers are used to store the intermediate result that is used soon by later pointer assignments. The update for the intermediate pointers is only used to transfer the points-to information. It neither bears dereference, nor impact the object live range, thus could be removed.

Figure 27 shows such an example. In this example, the two local pointers left and right are defined at the statements S1 and S3. Then they are used at the statement S2 ad S4. They do not have other definitions or uses anywhere else. Our baseline inserts the update code for the pointers left and right immediately after the statements S1 and S3 as shown on Figure 27 (a). The optimized update code is shown in Figure 27 (b), where the updates for the pointer left and right are removed. In this way, four update statements could be removed. Also, the compiler does not need to create entries for the two pointers left and right, and the entry initialization code for them could be eliminated.

```
struct tree *TreeAlloc (int level, int lo, int proc){
    struct tree * new, * right, * left;

    if(level == 0)
         return NULL;
    new = (struct tree *) malloc(sizeof(struct tree));
    new->val = 1;
    S1: left = TreeAlloc(level-1, lo+proc/2, proc/2);
    //update left's target
    S2: new->left = left;
    //update new->left's target
    S3: right = TreeAlloc(level-1, lo, proc/2);
    //update right's target
    S4: new->right = right;
    //update new->right's target
    return new;
}
```

(a)

```
struct tree *TreeAlloc (int level, int lo, int proc){
    struct tree * new, * right, * left;

    if(level == 0)
         return NULL;

    S1: left = TreeAlloc(level-1, lo+proc/2, proc/2);
    S2: new->left = left;
    //update new->left's targete
    S3: right = TreeAlloc(level-1, lo, proc/2);
    S4: new->right = right;
    //update new->right's target
    return new;
}
```

(b)

Figure 27. Treeadd example.

The conditions for this type of optimization are

(1) The definition p = q dominates its use.

(2) The use only has one definition, that is p = q.

(3) P is not dereferenced or used after the use.

The algorithm to propagate update is shown in Figure 28. The algorithm searches the pointer assignment list for a pointer use and its dominator definition. If the pointer use has no other definitions, then the update is delayed to the use point. If the pointer's all updates get propagated to its use sites, then the compiler does not generate a pointer id for it.

```
//input: ptr_assign_list – pointer assignment list

Propagate_Update{
   for(S in ptr_assign_list){
      for(S's use T)
         if((reachable(S, T, dom_tree)) && (T's definition set size is 1)
         && ! ((T' is S's) && (T' is reachable from T)))
            S.set_propagation(T);
      if(all updates for S are propagated)
         ptr_assign_list.remove(S);
   }//end for
}
```

Figure 28. Algorithm to propagate update.

3.7.7 Invariant Removal

Multimedia applications tend to use pointers to access big buffers. Those pointers are initialized once, and then used to visit the buffer data within a loop with auto increment. In such cases, the pointer's target is not changed in the loop. One example is shown in Figure 29. The pointer update pattern is simple arithmetic calculation like plus plus. The updates for those pointers in the loop do not change the information in the pointer table. Removing those updates does not harm the correctness and integrity of the pointer table, and could reduce the overhead significantly.

```
adpcm_coder(indata, outdata, len, state){
    outp = (signed char *)outdata;
    inp = indata;
    for ( ; len > 0 ; len-- ) {
       S1: val = *inp++;
       …
       S2: *outp++ = (delta & 0x0f) | outputbuffer;
    }
}
```

Figure 29. Adpcm_coder example.

```
int longest_match(cur_match)
       IPos cur_match;
{
   do {
   } while (*++scan == *++match && *++scan == *++match &&
     *++scan == *++match && *++scan == *++match &&
     *++scan == *++match && *++scan == *++match &&
     *++scan == *++match && *++scan == *++match &&
     scan < strend);
}
```

Figure 30. Gzip example.

Another example is shown in Figure 30. In the SPEC2000 gzip program, the longestmatch function is called 1.5 million times, and consumes 70% of the total execution time. In that function, two pointer scan and match are self increased in a do loop; both of the pointers targets are not changed within the loop. There are sixteen dereference stores per iteration. If we do not support the dynamic points-to sets, those dereferences will go unchecked, causing big security loss. On the other hand, if dynamic pointer tracking is deployed, the overhead for updating the pointer target is out of the loop, which incurs little slowdown. Tradeoff a little overhead for a big security gain motivates us to develop a dynamic pointer tracker.

3.7.8 On Demand Dynamic Pointer Tracking

In this section, we only propose some general purpose optimization techniques. Actually, a lot of domain driven approaches could be proposed to achieve efficient

dynamic pointer tracking. In some applications, not all pointers are sensitive. Thus, only those sensitive pointers need to be tracked. A sensitive pointer's value could come from a non-sensitive pointer's value. In this case, the sensitive pointer's value and its base type are used to do the update. The advantage of this on demand dynamic pointer tracking is the low overhead. The drawback is that sometimes not all points-to information is available. However, in some domains, tracking some of the pointers is more attractive due to the low cost and the special analysis requirement.

3.8 Dynamic Pointer Tracking Experimental Results

Table 4. The description of olden benchmark.

Benchmark	Description	Data Organization
Bh	Barnes & Hut N-body force computation algorithm	Heterogenous OcTree
Bisort	Forward & backward sort of integers using 2 disjoint bitonic sequences which are merged to get sorted result	Binary-tree
Em3d	Electromagnetic wave propagation in a 3D object	Single-linked lists
Health	Columbian health care simulation	Double-linked lists
Mst	Minimum spanning tree of a graph	Array of single-linked lists
Perimeter	Perimeters of regions in images	Quad-tree
Power	Power pricing system optimization problem solver, leaves are clients asking power and root is power plant returning pricing	N-way tree, single-linked lists
Treeadd	Recursive sum of values in balance B-tree	Binary-tree
Tsp	Traveling salesman problem solver using a particular algorithm and a closet point heuristic	Balance binary-tree
Voronoi	Computes the voronoi diagram of a set of points recursively on the tree	Balanced binary-tree

In this section, we show our experimental results and discuss our observations. We target x86 architecture that is widely deployed on personal PC and some servers in today's world. Currently we focus on programs written in the C programming language. The compiler we use is MACH-SUIF [81] that provides friendly user interface for the compiler development. The C code is converted to MACH-SUIF representation and our

work is implemented based on the MACH-SUIF code. The benchmarks are the olden benchmarks that make intensive use of pointers. The benchmark description is in Table 4. The benchmarks are instrumented to gather the dynamic pointer assignment accounts. In the following, we show the analysis results to demonstrate the opportunities for optimizing the baseline.

3.8.1 Static Pointer Analysis

First, we show the static pointer analysis results. The pointers are divided in two groups – local pointer, and global pointer. The classification is determined by where a pointer is declared. The scope of local pointers is a function; the scope of global pointers is either a file or a program. In other words, global pointers are defined either as static or as global.

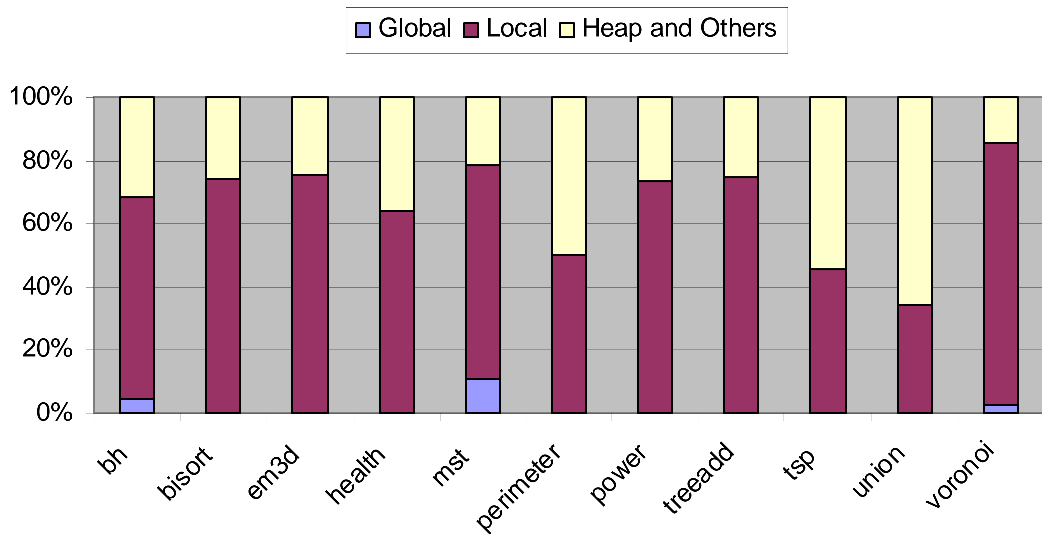
Table 5. Number of local and global pointers.

Benchmark	Glocal pointer	Local pointer
Bh	3	349
Bisort	0	29
em3d	0	53
Health	0	54
Mst	2	70
perimeter	0	18
Power	0	25
Treeadd	0	9
Tsp	0	45
Union	0	25
Voronoi	8	143

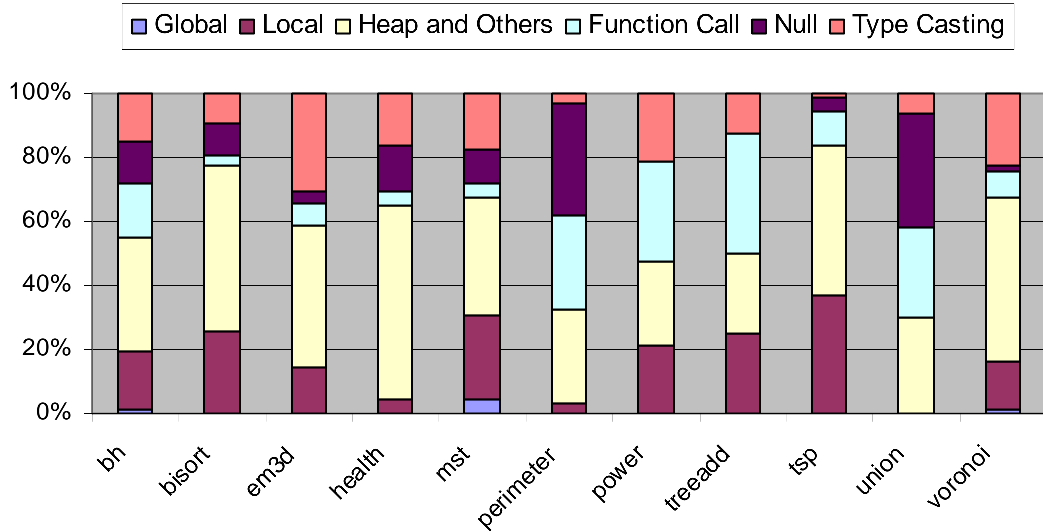
Table 5 reports the number of pointers. The average numbers of global pointers and local pointers are 1 and 68. From this table, we observe that the number of local pointers

is much bigger than the number of global pointers. Most of the local pointers are used to visit some complex structures and store the intermediate value. Thus, focusing on the local pointers for optimization is most beneficial.

Figure 31 shows the number of pointer assignments. Figure 31 (a) gives the pointers defined on the left hand side. The three columns represent the global pointers, local pointers, pointers on the heap and others, where others mean unknown type of pointers like parameter pointers. We could observe that most of the pointer assignments are assigned to local pointers. Figure 31 (b) shows the pointers that are used on the right hand side. Besides the three similar columns in Figure (a), Figure (b) has the function call, null and type casting columns that indicate the pointer's value coming from a function call, a null initialization, and some value through type conversion like pointer assignments through malloc functions. The right hand pointers type is diversified. Pointers on the heap and local pointers are the major things, the others like null and function call are also important.



(a) Defined on the left hand side



(b) Used on the right hand side

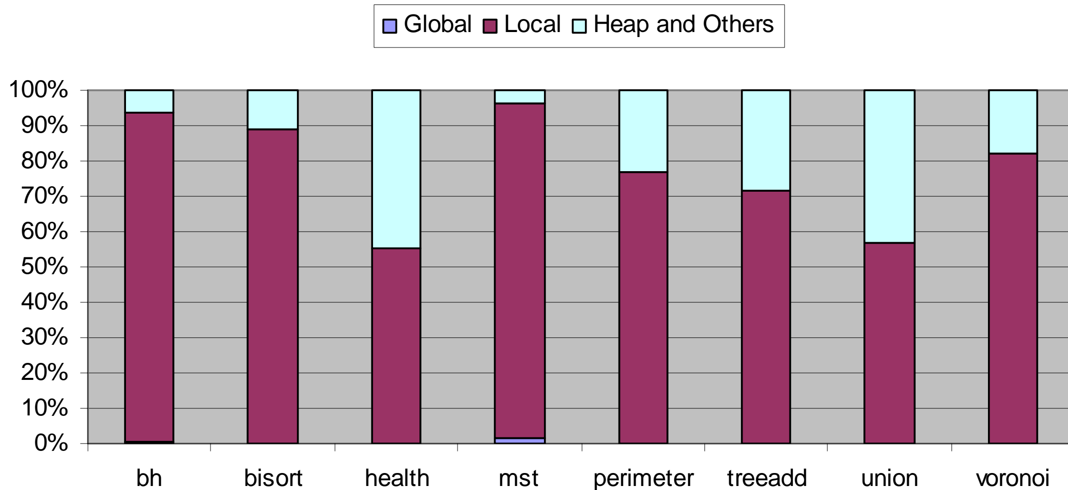
Figure 31. Static pointer assignment.

3.8.2 Dynamic Pointer Analysis

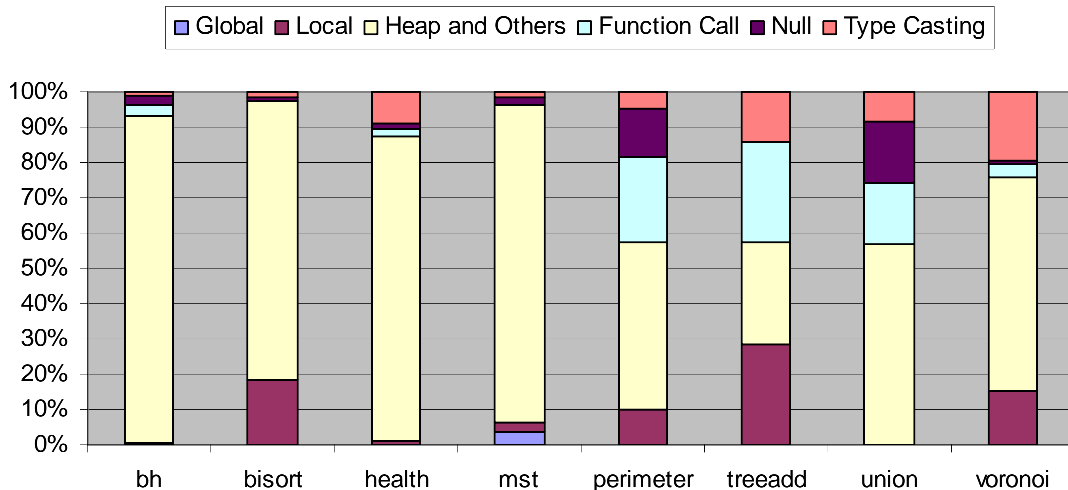
Table 6 shows the parameters to the olden benchmarks. We mainly follow the parameters provided by Trimaran [86]. The pointer chasing cases are rare in those programs. In our experiments, we assume that the pointer table has the start and end address. The dynamic aspect evaluation is based on this assumption.

Table 6. Olden benchmark parameter.

Benchmark	Parameter
Bh	512
Bisort	10000
Health	4 250 1
Mst	128 1
Perimeter	6 1
Treeadd	13
Tsp	20 1
Union	6 4
Voronoi	N/A



(a) Defined on the left hand side.



(b) Used on the right hand side.

Figure 32. Dynamic pointer assignments.

Figure 32 shows the dynamic pointer assignments. Heap pointers and parameter pointers possess almost all right side of pointer assignments. Most of the pointer conversions are used to allocate space for new objects. The function calls involved in pointer assignment are mainly through calling recursive functions. The two benchmarks em3d and tsp use library functions like drand48 and srand48 that are not supported by

SimpleScalar [3], and the power program causes segmentation fault when compiled using MACH SUIF, so their dynamic results are not shown in the thesis.

3.8.3 Recursive Function Analysis

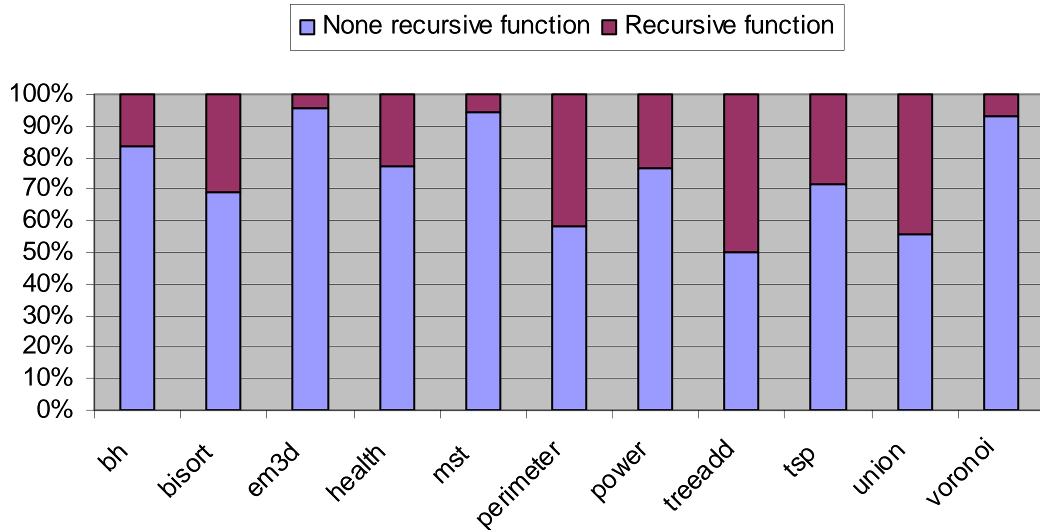


Figure 33. Recursive and none recursive functions.

Figure 33 shows the number of recursive and none recursive functions. On average, 25% of the functions are recursive functions, ranging from 4% to 50%. Figure 34 shows the number of pointer assignments in recursive and none recursive functions. On average, 47% of the pointer assignments are in the recursive functions, ranging from 0% to 97%. Both of the figures demonstrate that dividing the pointer table for the local pointers in recursive and none recursive functions is necessary due to the big number of pointer assignments in the recursive functions. Most of the pointer assignments in those benchmarks are used to create new heap objects or to visit the data structures. The pointers on the heap and the local pointers coexist in and interleave the pointer table. The live range of the pointers on the heap are long, and the live range of local pointers is limited to a function's live record, which could result in a large amount of interspace in the pointer table.

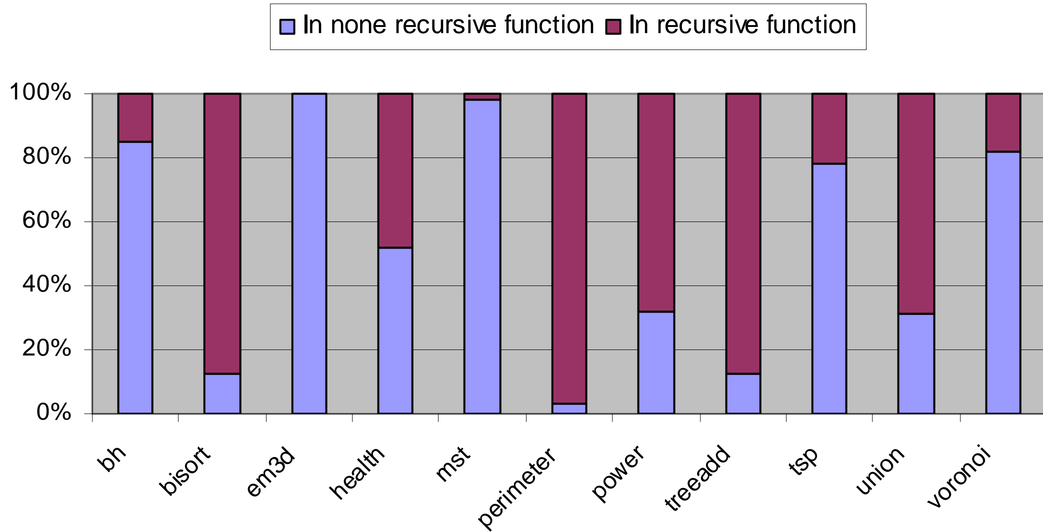


Figure 34. Pointer assignment in recursive and none recursive functions.

3.8.4 Performance Evaluation

The performance measurement is shown in Figure 35. The y-axis is normalized, where 100% represents the original execution time of a program. Every benchmark has three columns. The first column is the original execution time. The second column is the execution time of our baseline. The third column is the execution time with all optimizations turned on. From the results, we can observe that the average performance degradation under the baseline update scheme is 31%, ranging from 2% to 76%. After all optimizations are turned on, the performance degradation finally decreases to 15%. The savings for the health benchmark is mainly due to the points-to set propagation. The update for its intermediate representations is discarded as long as they do not impact the object liveness. The bisort benchmark overhead remains high since it uses two local pointers in a while loop to visit a binary tree and swaps the branches and leaves. The pointer assignment is like "pl=pll", and then "pll=pl->left". The update could change the object liveness. There is no type casting in the loop. If the points-to information is not needed urgently, the pointer's target could be updated outside of the loop with the pointer's value and type. The olden benchmarks make intensive use of pointers. In other

programs like SPEC benchmarks, only a few pointers are used, and the overhead is much smaller.

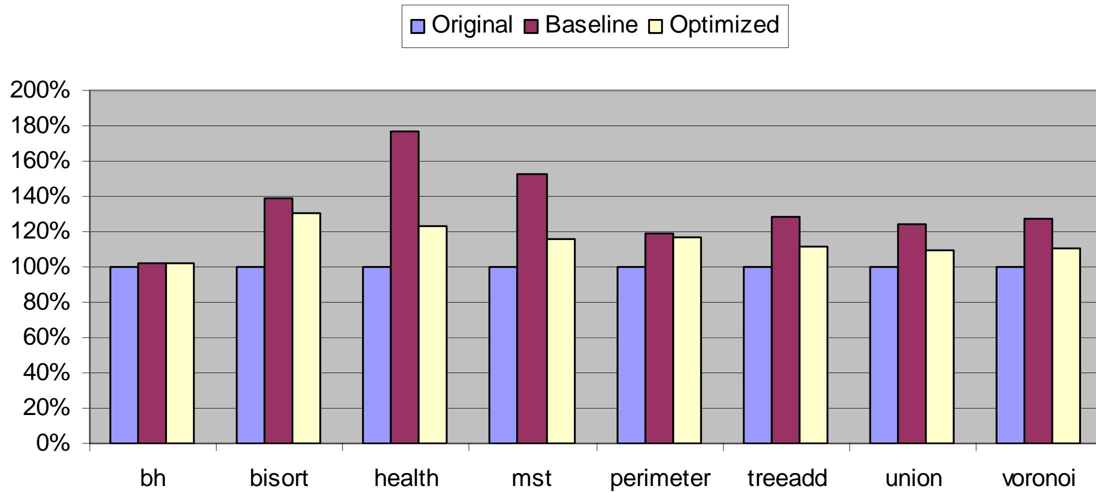


Figure 35. Updating overhead.

3.8.5 Dynamic Load/Store

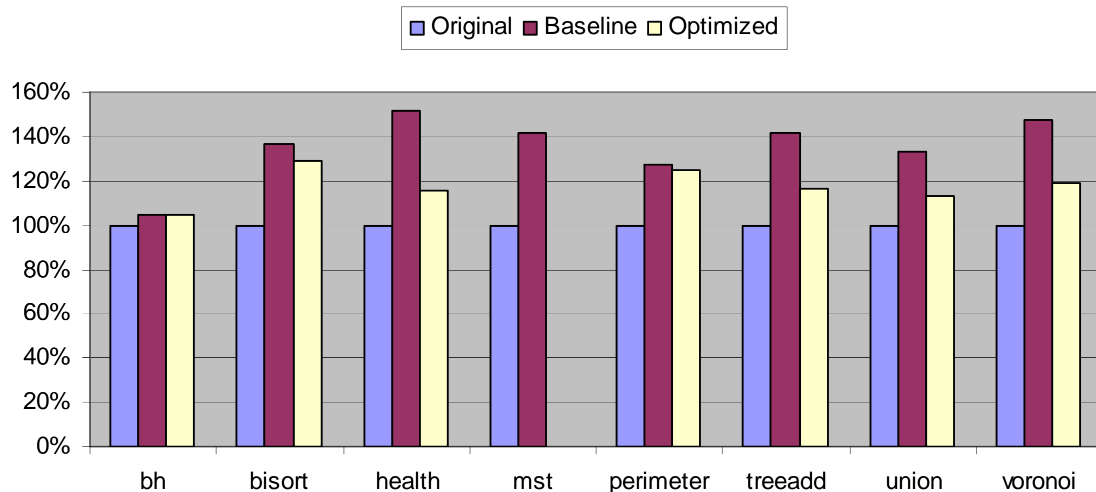


Figure 36. Cache effect.

The number of dynamical load/store is shown in Figure 36. The y-axis is normalized, where 100% represents the original number of loads and stores executed by

the program. On average, our baseline increases the number of memory accesses by 34%, and our optimized algorithm increases the number of cache accesses by 17%.

3.8.6 Space Cost

Table 7. Pointer table size.

Benchmark	Heap pointer table size (byte)	Static pointer table size (byte)	Total pointer table size (byte)
bh	6376	2232	8608
bisort	65552	976	66528
health	169728	464	170192
mst	132096	376	132472
perimeter	10688	352	11040
treeadd	65552	352	65904
union	32144	488	32632
voronoi	472	808	1280

Table 7 shows how much space is needed for the pointer table. From the table, we can observe that the heap pointer table is much bigger than the static pointer table. This is due to the intensive use of pointers in the olden benchmarks, especially involving numerous object allocations. The pointer table can go up to 17 KB with an average of 5KB, that is tolerable for modern computers with large cache. For other normal applications that do not use pointer intensively, the pointer table size is usually small, which can be controlled under several KB.

3.9 Summary

In this chapter, we focus on the dynamic pointer tracking algorithms, the design issues, and possible optimization opportunities. Pointer tracking is generally a difficulty problem since C programming language allows tremendous ways to use pointers. We develop a new data structure to keep the points-to information, and propose several improvements to reduce the overhead. Overall, our study on the compiler analysis and the dynamic tracking could be customized applied to a wide range of applications and research areas.

CHAPTER 4

MEMORY PROTECTION THROUGH DYNAMIC POINTER TRACKING

4.1 Introduction of the Problem

Software could suffer from various types of vulnerabilities, including buffer overflows, format string vulnerabilities, double frees, etc.[19]. Attackers devise clever techniques to exploit these vulnerabilities. If an attacker successfully exploits the vulnerability, the attacker can penetrate the system, and in most cases obtain the complete control over the system. According to the statistics from CERT [80], the number of vulnerabilities reported to CERT increased from 171 to 5990 during the period of 1995 to 2005. Thus, the problem is becoming worse instead of going away.

The fundamental way to solve the problem is to (re-) write secure code. But such efforts meet resistance due to the huge time to the market pressure, and the legacy issues due to the reused code. Thus, released software could contain various vulnerabilities. Since this situation is unlikely to change in the near future, countermeasures to attacks have been extensively investigated.

Intrusion detection system (IDS) is an important technology that is designed to offer memory protection. There are two types of intrusion detection systems. One is network based, which monitors network packets, and matches attack signatures in the packet content. The other kind is host based which resides in the host, and monitors the target system or the target program directly. In this work, we focus on host-based IDS. There are two general approaches proposed for host-based IDS. The first approach is based on signature checking. It detects attacks by using pre-identified intrusion signatures. Such approaches are accurate, but are unable to detect unknown attacks. Also, the signatures have to be constantly updated. Another approach is based on anomaly detection which is

our focus. It assumes the nature of an intrusion is unknown, but the intrusion somehow deviates from the program's normal behavior. Anomaly detection based techniques are able to detect unknown attacks, and thus are more powerful.

However, one important observation is that most of the previous anomaly detection work [33][72][62][32][43][46][35][34][44][77][22][63][14] focuses on monitoring program control flows and program paths. Control flow diversion has been successfully utilized by attackers previously to bypass the critical checking code or divert the execution to the injected malicious code. But with attacks becoming more and more sophisticated, control flow diversion may or may not be used as an attack technique. More fundamentally, though, memory tampering is the most common starting point used in starting and propagating an attack. Thus, one important question is whether monitoring program control flow is the right approach to detect the incidence of an attack. However, through our experiments, we found that a large percentage of memory tampering does not alter program control flows at all. For such memory tampering, monitoring control flows is pointless. In [13], Chen et al. showed several real non-control flow modifying data attacks. We will discuss those attacks in detail later, but the conclusion is that it is critical to detect data tampering that does not alter control flows.

Another solution for memory protection is bounds checking for arrays and pointers [27][42][5][36]. Misuse of pointers and arrays could result in memory corruption, but they are not the root cause of memory tampering. Thus, the important question one may naturally ask is that what the foundation of memory tampering is. To tamper with memory, an attacker must exploit an existing or an inserted store instruction to modify memory as the attacker desires. To detect memory attacks, we have to start from the general store instructions instead of purely relying on some special attack models or aggregate properties.

Store instruction based memory intrusion detection approaches did not really start to become an issue until recently [79][76][14]. This is mainly due to the hardness of pointer

analysis. However, most of these mechanisms cannot offer memory protection at fine level. Thus, how to get the exact points-to set and how to detect memory tampering at fine granularity becomes an important issue that has to be addressed.

In this work, we propose a dynamic access control based intrusion detection system with compiler support to detect data tampering directly. Our system is not only dealing with memory safety, but also is preventing memory intrusions due to attacks. The dynamic pointer tracking technique is a key enabling component in this design to achieve full memory protection at fine level. The basic idea is that the compiler identifies program regions, in which data should not be accessed according to the program semantics. The compiler conveys this information to the hardware, and the hardware checks data accesses based on the information. If the compiler asserts that the data should not be accessed, but there is an attempt to do so at runtime, an attack is detected. To achieve security at fine granularity, we further maintain some data structures to monitor, and control the data access dynamically. Although security carries a performance penalty, our analysis indicates that it is possible to explore optimizations to reduce the cost significantly. The main contributions of this work are:

(1) We design some data structures dynamically tracking the pointer activities.

(2) We offer a secure solution at fine level.

(3) We propose an intrusion detection system to detect a broad class of memory tampering.

In this chapter, we show the design and implementation of our intrusion detection system with dynamic pointer tracking.

4.2 Previous Approaches

Memory intrusion detection involves a great amount of work. In this section, we briefly discuss some memory protection related work to motivate the need for our intrusion detection system. To be effective, we divide the previous work in four

categories – control flow monitoring, non-control attack detection, bounds checking, and other memory protection techniques.

4.2.1 Control Flow Monitoring

First, we discuss the memory protection schemes based control flow monitoring. Unknown software attacks can be detected through anomaly detection techniques by monitoring a program's behavior, and comparing it with the expected behavior. Up until now, most anomaly detection research focuses on monitoring a program's control flow behavior at different levels of granularity.

Forrest et al. [33] found that the system call trace appears to be a good starting point for anomaly detection. System calls are generated as the program interacts with the kernel, examples of which are fopen(), fgets(), and fclose(). A system call trace can be considered as a distilled execution trace of a program leaving many program structures out.

Various system call monitoring based anomaly detection schemes have been proposed. The scheme in [33] tries to detect attacks by checking whether a small segment of the system call trace is normal. Finite State Automata (FSA) based techniques are also proposed to encode the expected system call trace information [46][72][62][67]. To construct the FSA for system call monitoring, every program statement invoking a system call becomes a state on the state machine diagram. The transitions between states are triggered by system calls. Each transition edge in the FSA is labeled by triggering a system call, and the target state is determined by the feasible control flows. The state machine can be easily constructed through a static analysis of the program [72] or by dynamically learning the system call trace and the program counter [62]. The main goal of system call monitoring is to detect how/whether an attacker is bypassing/creating an unusual trace. Although the most common class of attacks they detect is buffer

overflows, they could also detect other attacks like Trojan horses and format string attacks. But these attacks must cause some malicious function calls.

Later work points out that simply checking the FSA for system calls is not sufficient and misses certain attacks. For instance, Wagner et al. noticed that impossible paths may occur at call/return sites that an attacker can exploit [72]. This can cause anomalous control flows along those paths to be undetected. They propose a second model called the abstract stack model, which records the call stack information to solve the problem. In [32], the authors observed that even the abstract stack model can miss important attacks. Thus, in [32] they provide a solution called vtPath to detect certain attacks missed previously by considering more program information. By constructing two hash tables that store not only the possible return addresses but also the so-called virtual paths (i.e. the sequence of procedure entry/exit points and return addresses traversed between two system calls), more attacks can be detected. However, [32] also acknowledges that some attacks are still left undetected.

[44] incorporates system call arguments into the detection model, which is yet another example of incorporating more program information into the detection model. Most recently, [35] categorizes system call based anomaly detection systems into “black box”, “gray box”, and “white box” approaches. Systems that rely not only on the system call numbers, but also on the extra information extracted from the process memory fall into the “gray box” category. [35] further systematically studies the design space of “gray box” approach, and analyzes the importance of monitoring granularity with respect to the accuracy of systems. Their follow up work [34] gives a new “gray box” anomaly detection technique called the execution graph, which only accepts system call sequences consistent with the program control flow graph. However, due to the limitation of monitoring granularity (similar to [32]), execution graph is not able to detect certain attacks.

[73] develops a framework targeting soft timer based attacks. They maintain a database to record the legitimate soft timer interrupt requests. First, the transitive closure of all legitimate software timer interrupt request callback functions is constructed. Then, the symbolic soft timer interrupt request signatures are generated. Finally, the interrupt checker is placed in a secure domain to prevent the attacker from transferring the control from the guest kernel to the shell code. This work still relies on monitoring the soft interrupt triggered control flows.

In summary, monitoring granularity at a system call granularity level is not fine enough to detect many attacks in reality, since an attack could take place between two system calls. Privilege escalation is a type of attack that modifies memory to hijack root privileges, which may not lead to anomalous system call trace, but which does lead to anomalous control flow. Program shepherding [42] is a protection mechanism against control-flow hijacking. It does not rely on system call monitoring, but enforces several rules of control transfers. The attacks detected by program shepherding are quite limited. Recently, Zhang et al. [77] proposed an anomalous path checking technique with the hardware support to monitor segments of dynamic program paths. The idea is similar to the one in [33], but their monitoring granularity is finer, and the detection strength is better. However, their scheme still focuses on control flows solely, and tries to detect attacks by identifying suspicious control flows.

Control flow diversion has been successfully utilized by attackers previously to bypass critical checking code/divert execution to the injected malicious code. But with attacks becoming more and more sophisticated, control flow diversion may or may not be used as an attack technique. For such non-control memory tampering, monitoring control flows is pointless. Next, we show some mechanisms to detect non-control flow data attacks.

4.2.2 Non-control Data Attack Detection

```
FILE * getdatasock( … ) {
      ...
      seteuid(0);
      setsockopt( ... );
      ...
      seteuid(pw->pw_uid);
      ...
}
```

Figure 37. Attack without control flow tampering.

In [13], Chen et al. showed non-control data attacks are very realistic, and should be given a serious attention. Figure 37 gives a piece of code from the Wu-FTP version 2.6 showing a non-control data attack example. It contains a format string vulnerability that allows an attacker to modify any memory location. In normal situations, this piece of code temporarily escalates the user privilege using seteuid(0) to perform the setsockopt operation. Then it restores the original user privilege. But consider a scenario in which pw→pw_uid is tampered with, and is set to 0 through a format string attack. In this case, the second seteuid operation will always grant the attacker with the root privilege, although it is supposed to restore the original user privilege. So the attacker is able to retain the root privilege, and do all damages subsequently. Note that there is no control flow modification involved in this process.

Recently, some new approaches could deal with some kinds of non-control data attacks [1][79][76][14], although they may be originally developed for other purposes.

AccMon [79] is a software debugging tool with the hardware support to detect software bugs leading to memory corruptions. The scheme is based on the observation that a memory location is typically only accessed by a few instructions. In their work, a set of data objects is selected to be monitored. They use profiling to create an invariants table, recording the set of instructions that normally access the selected data object. At runtime, the program is monitored through the compiler inserted operations. If a data object is accessed by an instruction not observed during training, an alarm is raised. Their

scheme relies on profiling/training, so it may have false alarms. Moreover, since it is based on profiling, an instruction is regarded as legal to access a data object when it accesses that data object during any normal program execution, even though under a specific execution, it may be illegal for that instruction to access that data object. This leaves potential holes to be exploited by attackers. Figure 38 shows a simple example. Assume that the program accepts two kinds of user input. If the user input is user_input1, the pointer ptr points to obj1; if the user input is user_input2, ptr points to obj2. Later, the pointer is dereferenced at the instruction deref_inst. Under the AccMon scheme, in profiling (due to the issues of coverage during training), both kinds of user inputs are exercised, and the dereference instruction S1 is regarded as legal to access both obj1 and obj2. But during a specific execution context of the program, the pointer can only either point to obj1 or obj2, but never point to both obj1 and obj2. So if the user input is user_input1, ptr is illegal to access obj2, but AccMon regards it is legal. Thus an attacker could gain illegal access to obj2 and tamper with it without being detected. Their memory protection strength is weakened mainly due to their lack of precise points-to information. Thus, dynamic pointer tracking plays a key role in software protection.

```
ptr = NULL;
…
if ( user_input1 )
   ptr = & obj1;
else if (user_input2)
   ptr = & obj2;
S1: *ptr = …    // deref_inst
…
```

Figure 38. Security hole in AccMon.

Mondrian memory protection [76] is a micro-architectural framework that enables fine-grained memory protection supporting multiple protection domains. They target sharing memory and exporting services. The kernel set the data access permissions. Then, at runtime, the data access policies are checked for every data access. It defines access permissions possibly at the granularity of per word in a permission table. At runtime, the

value in the address register is used to lookup the permission table to see if the domain has appropriate access permissions. There are three limitations in their approach. First, they assume the kernel is trusted, but our approach makes no assumption regarding the kernel. Second, the access permissions are preset by the user, which reduces the scope where their technique is applicable. Third, in their system, the kernel manages the access permissions based on protection domains, but we derive the access permissions information from the program automatically with careful analysis, and the access ranges are updated dynamically based on the current context. The program semantics enable us to protect memory in a better way.

The object level memory tampering detection algorithm proposed by Zhang et al. [1] controls the access permissions for data objects dynamically. They use a table to collect the information regarding which object should be accessed by which store instruction. The table is statically created through the semantic analysis of a program. At runtime, all store instructions are checked regarding their access permissions. Any attempt to modify a read-only object triggers an alarm. There are three major problems in their design. They could not handle pointer dereference store instructions. In their work, they use the profile to estimate the behavior of pointers. However, without the precise points-to information, full memory protection could not be achieved. Dealing with pointer dereference is difficult. Static compiler pointer analysis has a large number of limitations, and in some cases cannot determine an accurate points-to set for a pointer dereference. In their work, before/after a pointer dereferencing instruction, the access permissions of all possible pointed-to objects have to be set, which could increase the protection overhead significantly. If we choose not to handle these cases, we have to give up protecting the dereferencing instructions, and a large number of locations would be exposed to attackers. Thus, such solutions are unacceptable. The other option is to undertake dynamic pointer tracking to discover the accurate pointer targets at runtime. In this work, we choose the latter option.

The profiling based analysis [79] is based on the relation of a store instruction and its referent-object. First, it determines the set of objects that are accessed by a store instruction in the training runs. Only the set of objects are supposed to be legally written by the store instruction. Then, at runtime, if a store instruction attempts to write an object that is not in the set, an alarm is raised. This approach has the same problem as [1] due to their profile based points-to set estimation.

Infoshield [65] proposed a framework to protect the sensitive information against theft with little overhead. The tool recognizes the use pattern of the sensitive data, and instruments the program with verifications on the sensitive data access. At runtime, the address of the sensitive information is enrolled with the instructions that are allowed to access the contents of the address. Every load and store instruction is checked to ensure that they are the legal instructions. Their work has two limitations. First, they predefine the sensitive information, and only focus on that data, which weakens their protection strength. Second, they require the program developers to provide annotations indicating which data is sensitive, which adds the development overheads.

4.2.3 Bounds Checking

Jones and Kelly [40] proposed a referent-object based bounds checking approach. Later, [61] revised Jones and Kelly's work to handle the out-of-bounds pointers in a similar way. A referent-object is an object that is pointed by a pointer. They dynamically maintain an object table to keep the address and the size of all objects. To determine if a new address computed based on a pointer is in-bounds, a checker searches the object table for the referent-object of the pointer. Then, the checker verifies whether the new address falls within the extent of the referent-object. They modify the program to keep the object table up to date. Our approach differs from theirs. We aim at protecting memory; they target bounds checking only for arrays and pointers. The table in their scheme involves huge overheads, since whenever an operation on an array or a pointer is

performed the table has to be searched to get the object that the array or the pointer refers; our data structures are well designed as direct access tables so that the checking process speeds up.

In [51], George Necula et al. proposed a type safe system for C programming language. They used formal methods to design the type system. They derived new pointer representations and new operations on pointers to guarantee the safe pointer dereference. Our work is different from theirs. First, CCured maintains the points-to information (like the start address and the end address) in memory around the pointer; we collect the information in one integrated table (i.e. the pointer table). The start address and the end address of a pointer are security sensitive. If this important data is not well protected, an attacker could modify the address of a pointer, thus tamper with the program. Second, our technique could be extended to provide a hardware based IDS due to the integrated data structure design.

Purify [36] keeps a state for each memory byte. It supports three states, unallocated, allocated and unintialized, allocated and initialized. The transition between states is determined by memory allocation and initialization. The memory access is intercepted to lookup, check, and update the state table. Memory access inconsistent with the state code causes an alert message. There are two big concerns about their work. One is the two-bit state code for every byte in memory, which could consume 25% more space. Second is the function call that is used to trap every memory access, which could cause a factor of 5.5 slowdown over the optimized C code.

Pure software based memory error detection incurs high overhead. MemTracker [71] proposes a similar idea as Purify with the hardware support. They design several events, a programmable state transition table and a dedicated control register to speed up the state table lookup step. It shows that the decrease the performance overhead in SPEC2000 is under 5%. Their work demonstrates that hardware assisted design is critical

to remove the performance barrier. However, the space cost remains the same, which hinders the scalability.

Hardbound [26] improves the fat pointer approach by developing a hardware bounded pointer architectural primitive that enforces special memory safety for C programs with both hardware and software help. The compiler finds where the pointers are defined. The hardware maintains the shadow base and bounds information for every register and memory word in a sidecar shadow space. The hardware propagates the bounds information of a pointer, and checks a pointer dereference against the bounds. The metadata is separated from the pointer and it is encoded to reduce the overheads. Their design has two major advantages. First, the fat pointer metadata is hidden by the hardware, so the memory data layout is not changed, which provides compatibility with uninstrumented code. Second, the hardware speeds up the updating and checking process.

SoftBound [50] proposes similar work as ours. They centralize the metadata in a lookup table. The program is instrumented at pointer assignment places to update the lookup table. They implement two types of lookup tables (hash table and shadow space). Our work differs from theirs at three aspects. First, the attack models are different. They focus on the load and store instructions through pointer dereference; our work is designed as a general solution protecting all store instructions against memory tampering. Second, focusing on the memory protection based on indirect memory store instructions through pointer dereference incurs 54% (for the hash table implementation) and 22% (for the shadow space implementations) overheads. In this work, we develop kinds of optimizations to improve both of the dynamic pointer tracking and the memory protection technique, which controls the performance degradation under 15%. Third, the narrowing of bounds in their work could trigger false alarms, which is not desirable by the program developers.

4.2.4 Other Approaches

There are some papers using code transformation and encryption techniques to protect memory [41][15][28][70][38][51][6][30][66][52][47]. Kirovski et zal. [41] proposed an intrusion detection system (SPEF) with the architectural and compilation support. SPEF inserts encrypted constraints to every block of instructions, and then verifies the constraints at runtime. Crispin Cowan et al. [20] presented PointGuard, a compiler technique to defend against most of buffer overflows by encrypting the pointers when the pointers are stored in memory. These techniques could be breached by smart attackers in a brute-force way.

4.2.5 Previous Work against Attacks

Memory intrusion detection involves a great amount of work. In this section, we briefly discuss three memory tampering attack examples to motivate the need for our intrusion detection system. To be effective, we choose five relatively new memory protection techniques as typical representatives for comparison. They are the bounds checking technique [40], the non-control data protection algorithm [1], the word level memory protection approach [79], the control flow monitoring solution [77], and the profile based intrusion detection mechanism [79].

First, we describe the three memory tampering attacks. Buffer overflow is one of the most popular memory attacks. One example buffer overflow attack is shown in Figure 39, where an attacker is attempting to use a long input to overrun the buffer from the field username to another field dir through the statement S1. Another common kind of attack is double free that exploits the program vulnerabilities through incorrect invocations of the free function. One instance of the double free attack happens at the statement S6. The only thing the attacker needs to do is to lead the program to execute the statements S3 and S6. The object NewUser is freed twice, which could result in program crashes. The third example is the off-by-one problem as shown at the statement S7. Next, we discuss

the five memory protection techniques and show how they cannot detect all above example attacks.

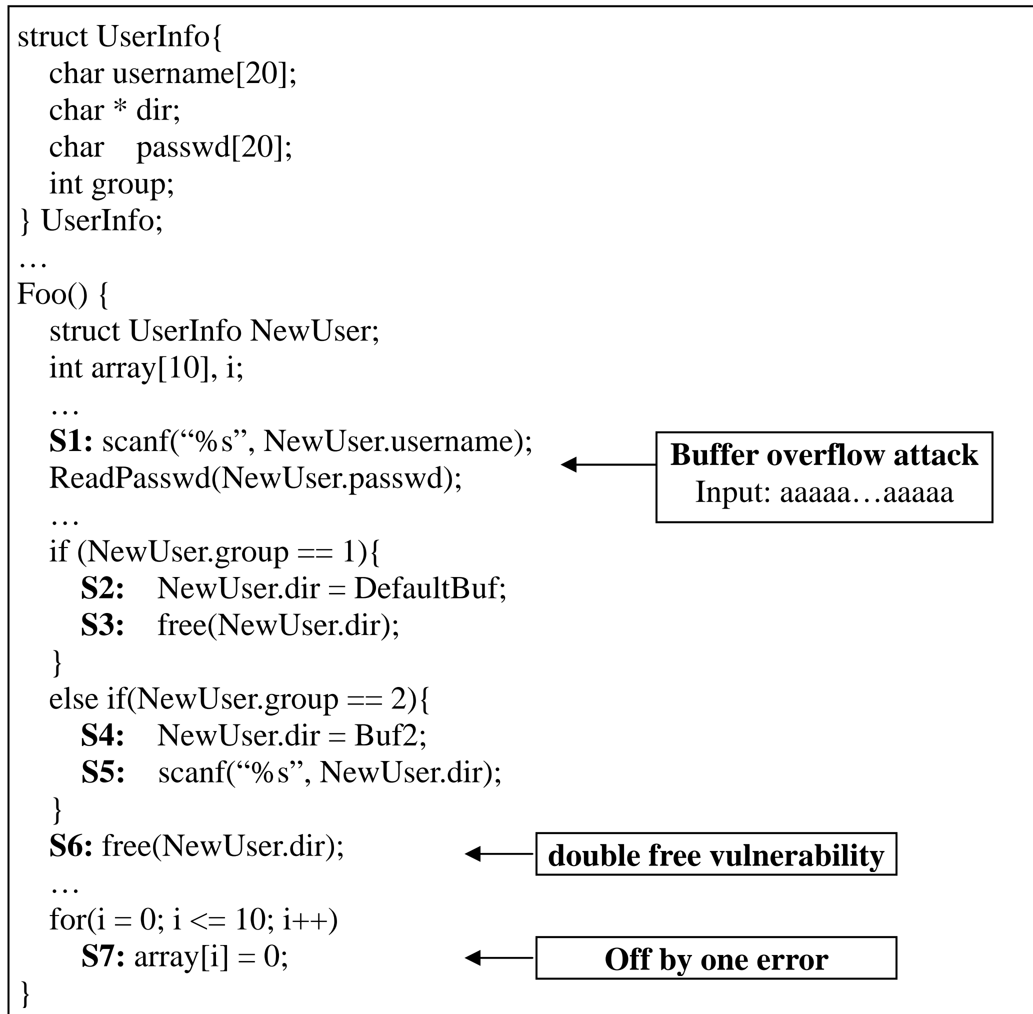


Figure 39. Memory attack example.

One disadvantage of the bounds checking work in [40] is that they do not check the bounds for arrays or pointers defined in a structure. The buffer overflow attack example shown in Figure 39 at the statement S1 cannot be detected, since username is defined within the structure NewUser. Modern programs tend to use complex nested data structures. Their analysis granularity restricts their work from detecting program errors at fine level.

The memory tampering detection algorithm [1] could not protect memory completely due to that their points-to set analysis based on the profile compromises their protection strength. Look at the if-else statement in Figure 39. If in the training runs, the statement S2 is executed more frequently than the statement S4, the estimated points-to object of the pointer NewUser->dir should be DefaultBuf. Suppose that in the testing run, the variable NewUser->group is equal to 2, and the pointer NewUser->dir should point to another buffer other than DefaultBuf. In this case, the store instruction at the statement S5 will not be checked, since the current object pointed by the pointer NewUser->dir does not fall into the estimated points-to set. To solve the problem, at runtime, dynamic program monitor must be proposed to get the exact points-to set.

For the code segment shown in Figure 39, in the Mondrian Memory Protection work [76], consider a scenario that the permission key for the object NewUser is enabled in advance due to some data sharing concerns. When the statement S1 is executed, the Mondrian Memory Protection will let the buffer overflow attack go unchecked.

The limitation of all control flow monitoring based techniques like [77] is that they cannot detect attacks that do not modify control flows. One simple example is that off-by-one problem shown at the statement S7 in Figure 39.

For this example, the profiling based technique [79] would raise a false positive at the statement S5 in a simple scenario. Assume in the training runs, the statement S2 is executed, but the statement S4 is never executed, thus the set of objects that is supposed to be written by S5 is empty. In the evaluation run, the statement S4 is executed. According the implementation in [79], the checker verifies the store instruction at S5, and claims that this store instruction has no right to write the object NewUser->dir since NewUser->dir is not supposed to be a referent-object of the store instruction. This triggers a false positive which is not desired by the developer.

None of these memory protection techniques could detect the example attacks, due to the fact that previous memory protection solutions may only protect software against

certain attacks under some constraints, and due to the hardness of dynamic analysis. In this work, we propose a general solution to protect against all memory attacks.

4.2.6 Our Approach

In this work, we propose and implement an intrusion detection system with the compiler support to detect data tampering directly. Our approach aims at protecting all program data. We tackle the root cause of memory attacks by detecting data corruptions directly.

Briefly, the compiler first determines which memory access should be restricted, and inserts the code necessary for updating the access control dynamically. Then, at runtime, the memory accesses are checked regarding the access permissions. In the design, the dynamic pointer tracking is a key component, since it enables us to extract the precise access range information. This leads to superior attack detection strength. Our static data flow analysis is conservative, thus there is no false positives, which greatly improves the applicability of the scheme. Due to C semantics, the dynamic updates could be frequent, so maintaining these data structures is expensive. We further propose a solution to eliminate the redundant updating operations. Thus, we provide per memory access fine granularity memory protection. There are many implementation issues and optimization opportunities, which are detailed in the following.

4.3 Attack Model

Our attack model is based on memory tampering. We assume that all store instructions are attacking instructions. Our technique is not restricted to some specific kinds of attacks. We do not make any assumption about attack initiation mechanisms (such as whether it is coming from input injection, malicious code injection, memory snooping etc.) except that its end goal is to perform memory tampering.

Scope and Limitations. Currently our compiler transformation is only applied to the user program, the statically linked libraries and some attack sensitive functions in dynamically linked libraries (like scanf, printf, strcpy, recv, free, strcat). The program code is set read-only after it is compiled and linked. The effectiveness of our scheme is determined by the knowledge about memory access instructions that the compiler can obtain statically. Our scheme is most effective when the compiler knows exactly which single variable a memory write instruction is going to access. In those cases, the baseline scheme shown below can provide protection from any memory tampering.

4.4 Compiler Analysis and Optimizations

This section presents our protection scheme. We first introduce some important definitions used in our technique.

Definition 1: A *memory object* is a data object defined in a program and resides in the memory. Memory objects include local stack data objects, static global data objects and dynamically allocated heap data objects. Common memory objects are scalar variables, arrays and aggregated structures etc.

Definition 2: An *access permission level* of a memory object is either writable, or read-only.

Definition 3: A *protection point* is a program point where the access permission levels of some memory objects are changed by setting the corresponding access bits for those memory objects.

Definition 4: A *protection operation* at a protection point is denoted as the action of changing the access permission levels of some memory objects at this point. Generally, there are two types of protection operations – changing from writable to read-only and changing from read-only to writable.

4.4.1 Baseline Scheme

First, we discuss a most basic implementation of our idea. In such a baseline scheme, every memory object has a corresponding access bit containing the access permission level for it. There are two types of protection points – before a store instruction and right after a store instruction. Assume that a memory object "O" is written by the store instruction. The access permission level for O is changed to writable right before the store instruction and is changed back to read-only after the store instruction. Initially, all memory objects access permissions are set to be read-only. During the execution of the program, whenever a store instruction is to be executed, the hardware checks the access bit table to see whether this instruction violates the memory access permission, i.e. writing a memory object whose access permission is read-only. If there is a violation, it alerts the attempted attack. Note that the memory corruption has to be done by some store instructions in a software-based attack. This store instruction could be some existing instruction in the program but writing a memory object that it is not supposed to. A second possibility is that this store instruction could be injected by an attacker. In either case the tampering is detected by our scheme.

The baseline scheme can prevent memory corruption attacks completely. In other words, all memory objects can be protected from being attacked. But it is infeasible to realize this complete protection mechanism due to the very large overhead of setting and checking access permissions. Thus, we propose three compiler optimizations to reduce the performance overhead in a significant way while maintaining the memory protection strength at a fine granularity level. We empirically show how strong protection can be achieved for typical attack models with minimized performance degradation.

4.4.2 Compiler Framework Overview

The compiler's task is to analyze the program, optimize the baseline scheme, collect the information about how to properly set up memory objects access permission and convey this information to the runtime component.

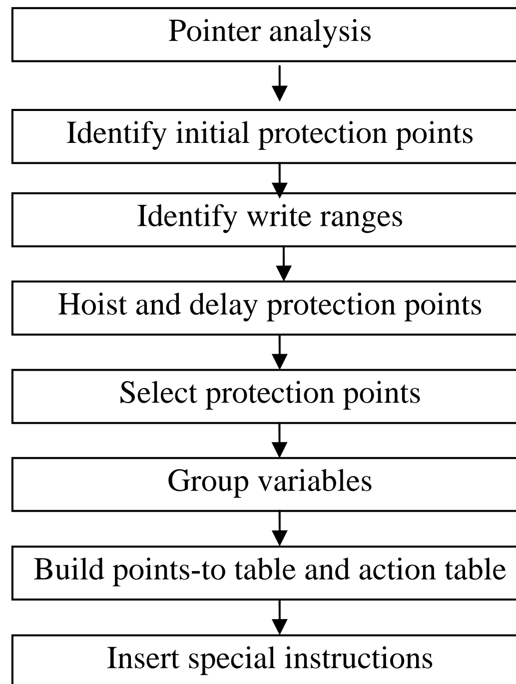


Figure 40. Compiler framework overview.

Figure 40 depicts our compiler framework. There are eight steps. First, the static pointer analysis is used to find which memory objects a pointer points. Second, all store instructions are identified by the compiler. All program points right before/after store instructions are treated as initial protection points as we stated in the baseline. Third, all write range information is collected to determine where to conduct the protection (we will talk about write ranges later). Forth, the hot protection points are hoisted/delayed to cold basic blocks. Then given some performance degradation constraints, the least beneficial protection points are removed. Next, some memory objects are grouped to be protected together whenever such an opportunity exists. An action table recording protection points and corresponding protection operations is created. So is a pointed-to table used to assist the handling of pointer dereferences. Finally special instructions are inserted into the code to inform the processor when to look up the action table. Details about the above steps follow.

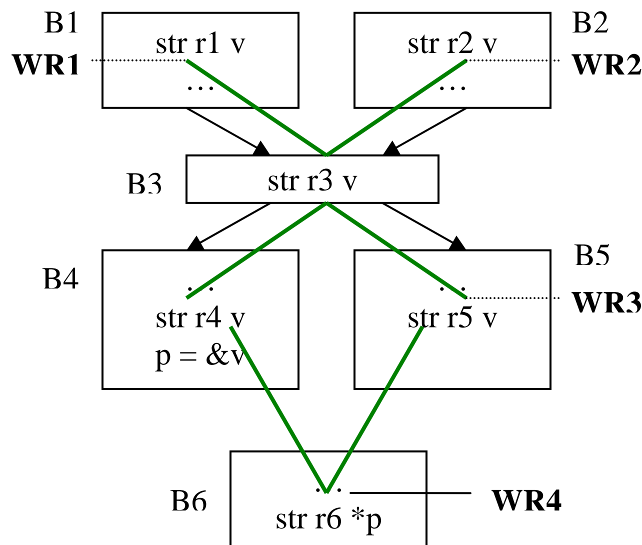


Figure 41. Write range example.

4.4.3 Write Range Identification

A write range is similar to the live range of a variable, except that the starting point of a write range is a store instruction on a memory object, and the ending point is the next closest store instructions on the same memory object. A write range is used to identify protection pairs – given a "set to writable" protection operation, what are its corresponding "set to read-only" protection operations, or on the other hand, what are the related "set to writable" protection operations for a "set to read-only" protection operation. Every pointer assignment is treated as a store instruction. Statically, if a store instruction is through a pointer dereference, and the pointer's points-to objects may be aliased with the variable used in another store instruction, then a write range exists between the two store instructions. The information is collected to assist the later optimization phases.

Figure 41 gives an example to show the write ranges of a variable v. There are five store instructions on v in five basic blocks, resulting in three write ranges – WR1, WR2 and WR3. One from "str r1 v" to "str r3 v", one from "str r2 v" to "str r3 v", and another

from “str r3 v” to “str r4 v” and “str r5 v”. The three write ranges do not include the store instructions themselves. In other words, a write range starts at the point right after a store instruction and ends just before the nearest next store instruction for the same variable. In the block B4, a pointer p points to v. The pointer evaluation determines that the five store instructions are aliased with the pointer dereference store in the B6. A write range WR4 is recognized upon the identification of aliased stores.

We utilize the standard framework of webs to identify write ranges. A write range’s starting and ending points are program points where protection operations should be performed initially. In the above example, the access permission level for v should be set to read-only at the start of WR1, and it should be changed back to writable at the end of WR1. It is the same for WR2, WR3 and WR4. Thus, protection points are the boundaries of write ranges.

4.4.4 Protection Points Hoisting and Delaying

The large overhead of complete memory protection by setting access permissions at all protection points is mainly due to that some store instructions are executed many times. For example, in some multimedia encoding applications, the encoding process is done in a major loop executed a very large number of times; while other code outside the loop body is seldom touched. Hoisting/delaying a protection point out of the loop body would afford us more efficient protection mechanisms. This observation motivates us to come up with an optimization to move the protection operations from a hot basic block to a cold basic block whenever possible.

```
Func:   HoistPP
Input: PP – a variable's protection point set
       PD – a variable's related pointer definition set
       CFG – control flow graph of the program
       Profile – Hot/cold basic block information in the CFG
       Threshold – a threshold to make cost-benefit tradeoff
Output: Optimized protection point set

For each variable v
  For each P∈PP[v] in a basic block B
    If (B is not hot) || (∃Q∈PP[v] in B and Q is before P)
    || (∃Q∈PD[v] in B and Q is before P)
      continue;
    For each preprocessor BP of B
      _push (BP, S)
    EndFor
    HoistSet = ProcStack
    //BP∈HoistSet
    Hoist_benefit = B.freq – ∑ BP.freq
    Hoist_cost = ∑ dynamic stores from BP to P
    If (Hoist_benefit/Hoist_cost > threshold)
      PP[v] = PP[v] – {PB}
      PP[v] = PP[v] ∪ HoistSet
    EndIf
  EndFor
EndFor
```

Figure 42. Algorithm for hoisting protection points.

Figure 42 gives the pseudo code for our hoisting algorithm. Starting from a hot basic block containing a protection point, we go back along the edges in the control flow graph. Its predecessor basic blocks are pushed onto a stack for further processing. For a pointer dereference store instruction, the protection point cannot be hoisted cross the boundary where the pointer is defined. Before a pointer is defined, its points-to information is unknown, and we do not know which object should be allowed to write. We provide PD as the set of pointer definitions where the pointer is dereferenced in the store instruction.

```
Func: ProcStack
Input: PP – a variable's protection point set
       PD – a variable's related pointer definition set
       CFG – control flow graph of the program
       Profile – Hot/cold basic block information in the CFG
       S – stack for processing
       P – the protection point to be hoisted
       BB – basic block where P is
Output: HoistSet – Set of protection points that the original protection
       point is going to be hoisted to

HoistSet = Null
While (B = pop(S) != NULL )
  B.visit = true;
  If(B is BB)
    If(∃Q∈PP[v] in B and Q is after P)
      return Null
  Else
    If B is hot
      If ((∃P∈PP[v]) && (P is in B)) || ((∃P∈PD[v]) && (P is in B))
        return Null
      Else
        For each preprocessor BP of B
          If B.visit = false
            _push(BP, S)
          Else
            return Null
          EndIf
        EndFor
      EndIf
    Else
      HoistSet = HoistSet∪{B}
    EndIf
  EndIf
EndWhile
return HoistSet
```

Figure 43. Auxiliary algorithm for hoisting protection points.

ProcStack in Figure 43 is an auxiliary function to explore the algorithm along backwards edges. It first examines the block on the top of the stack to see if it is hot. A basic block is said to be hot if its execution frequency exceeds some preset threshold. If it is cold, then we get one desired location to insert the protection operation. Otherwise, we should check whether we have reached a boundary – a basic block that contains another

protection point for the same memory object, or a pointer definition for the pointer dereference store. In other words, the algorithm determines if we have reached a program point where the object's access permission level is just changed from writable to read-only, or a point where the pointer is just defined, so we have to stop here and leave the hot protection point for further optimizations. If the current basic block that is being processed is hot and it does not include any protection operation on the same memory object, we continue pushing its predecessor basic blocks onto the stack. A control flow graph may include loops, so we use a flag to indicate whether a basic block has been visited, and also check whether we come back to the same basic block where the store is. If the same basic block containing the protection point P is seem again, we have to check if there exists another store after P. If so, it means that we reach a loop and other stores on the same variable prevent us from hoisting the protection point.

After obtaining the potential locations for hoisting a protection point, we determine whether to actually move the protection point out of the hot basic block based on a cost/benefit analysis. The benefit is denoted as the savings in terms of dynamically executed protection operations. The cost is defined in terms of the security degradation. Hoisting a protection point to a predecessor point means that the object could be corrupted between the two points. So it is not protected in that region. If the ratio of benefit/cost is bigger than a threshold, then the protection point is hoisted. Currently the threshold is determined by the system manually, which can be optimized using some heuristic solutions.

Figure 44 gives a sub control flow graph showing how the hoisting algorithm works. There are two store instructions in blocks B2 and B8, generating four protection points, right before and after the two stores. Assume all blocks in the loop body are hot. It is easy to see that the protection point before the store in B2 can be taken out of the loop to the block B1. That means we can set the access permission level for v1 to writable at the end of B1 instead of in B2. Similarly, if B6 is cold and B8 is hot, then the protection point can

also be lifted from B8 to B6. On the other hand, if B6 is also hot, the hoisting algorithm has to proceed by processing blocks B5, B4, B3 and B2 until it meets another store for v in B2. Thus there are two opportunities of moving protection points from hot blocks to cold blocks in this example.

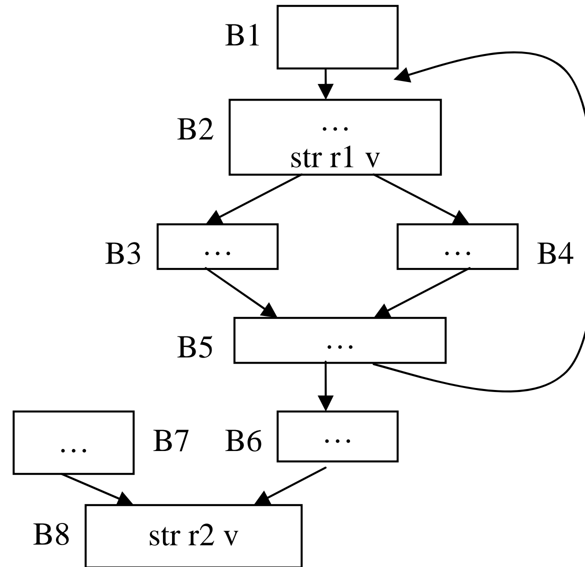


Figure 44. An example of hoisting protection points.

The benefit also comes with penalty – the degradation in security. If a write is only permitted just before the instruction "str r1 v", the variable v is well protected. If the variable v is set to be writable at the end of B1, the attacker can possibly corrupt variable v during the program's execution between the end of B1 and the store instruction. So a tradeoff must be made to balance the benefit and the cost. On the other hand, if there is no other store instruction between the end of B1 to the store instruction to variable v1, the protection remains intact or does not degrade at all. This is because the attacker has to execute a store instruction to corrupt a memory object. Thus, some motions of protection points actually do not result in security degradation if they don't move the protection point past another store (which could be malignant).

The similar algorithm is applied to delay the protection operation of changing the access permission from writable to read-only from a hot basic block to a cold one. Look

at the example in Figure 44 again. Initially, variable v is set to be read-only right after "str r1 v" in B2. It could involve great overhead since B2 is hot. So if B6 is cold, we could delay this protection to B6. However, there is a relatively long path from B2 to B6. So the protection to variable v may be degraded significantly. In this case, the compiler determines whether it should be delayed or not after considering both the benefit and the cost similar to that of hoisting a protection operation (i.e. if benefit/cost is bigger than the threshold, it is decided to be delayed).

4.4.5 Protection Points Selection

Due to the big overhead, completely protecting all memory objects may not be feasible. So how to determine where and which memory object should be protected are the key issues addressed in this section.

We build a cost/benefit analysis model to select protection points based on the profile data. The analysis unit is the set of protection points of a write range. All protection points for a write range have to be analyzed together since they are related operations. In other words, if a "set to read-only" operation for a memory object is removed, the corresponding "set to writable" operations are not necessary any more. On the other hand, if only the "set to writable" operation is deleted, its corresponding writes are not checked. Otherwise, the corresponding writes will be regarded as illegal and there will be false alarms.

The benefit of removing a write range is denoted as the sum of the dynamically executed store instructions within the write range. In other words, the benefit corresponds to the protection offered to the number of stores in a given range and is therefore proportional to the number of stores. Benefit means how many store instructions we could protect if we have the write range. The cost of a write range is defined as the sum of the dynamic execution times of all protection points in this write range. The cost

means how many overheads we have for keeping the protection point. The cost is equal to the sum of the frequencies of basic blocks where the protection points are located.

```
Func: SelectPP
Input: AllWR – write ranges set
          Threshold – performance degradation constraint
Output: Optimized write ranges set

CompCostBenefit
// WR ∈ AllWR
performance_degradation = ∑ WR.cost
While (performance_degradation > threshold)
        WR ∈ AllWR with minimum weight
        AllWR = AllWR – {WR}
        Performance_degradation -= WR.cost
EndWhile
```

(a)

```
Func: CompCostBenefit
Input: AllWR – write ranges set
          Profile – Hot/cold basic blocks information in the CFG
Output: Weight of each write range

For each variable v
   For each WR ∈ AllWR[v]
      WR.benefit = ∑ (dynamic stores within WR)
      WR.cost = ∑ (instruction at protection point in WR)
      WR.weight = tradeoff(WR.beneft, WR.cost)
   EndFor
EndFor
```

(b)

Figure 45. Algorithm for selecting protection points.

Figure 45 shows the pseudo code for the algorithm to choose the most profitable memory objects to be protected. The algorithm first analyzes the cost/benefit for all write ranges. The weight of a write range is based on the cost/benefit tradeoff which indicates its priority in terms of protection. The weight is equal to WR.benefit/WR.cost.

Then, driven by the performance degradation constraint, the algorithm repeats removing write ranges until the degradation requirement is satisfied.

As an example, consider the control flow graph in Figure 44. If the "set to read-only" operation cannot be delayed from B2, we re-evaluate its benefit and cost in this compiler pass to determine if such a protection point should be deleted.

4.4.6 Grouping Protection Operations

In this section, we introduce a compiler technique called protection operations grouping. Normally, each protection point corresponds to one protection operation that incurs certain runtime overhead. However, interesting optimization opportunities exist when multiple protection points performing the same type of operations are clustered together and the memory objects they protect are adjacent in memory. In these cases, instead of incurring one protection operation for each protected memory object, the protection operations can be grouped together as one protection operation for all memory objects at the clustering point. Figure 46 shows a simple example. Assume variables v1 and v2 are adjacent to each other. The start address of v1 is addr1 and the size of it is size1. The start address of v2 is addr1+size1 and the size of it is size2. Instead of setting access permission levels of v1 and v2 separately, we can group the operations and set the access permission level of the memory region with start address as addr1 and with size as size1+size2. The grouped protection operation achieves better performance since it reduces the number of dynamic protection operations and the associated runtime overhead.

Such optimization opportunities occur when there are multiple protection operations clustered together and when the memory objects protected at the clustering point are adjacent to each other. We can easily create clustered protection operations when there are multiple adjacent store instructions as shown in Figure 46. In our scheme, the compiler tries to move store instructions together as long as their dependencies are still

satisfied. Also, after the protection point hoisting/delaying algorithm is performed, many protection operations tend to be grouped at the same program point, such as the beginning/ending of a cold basic block.

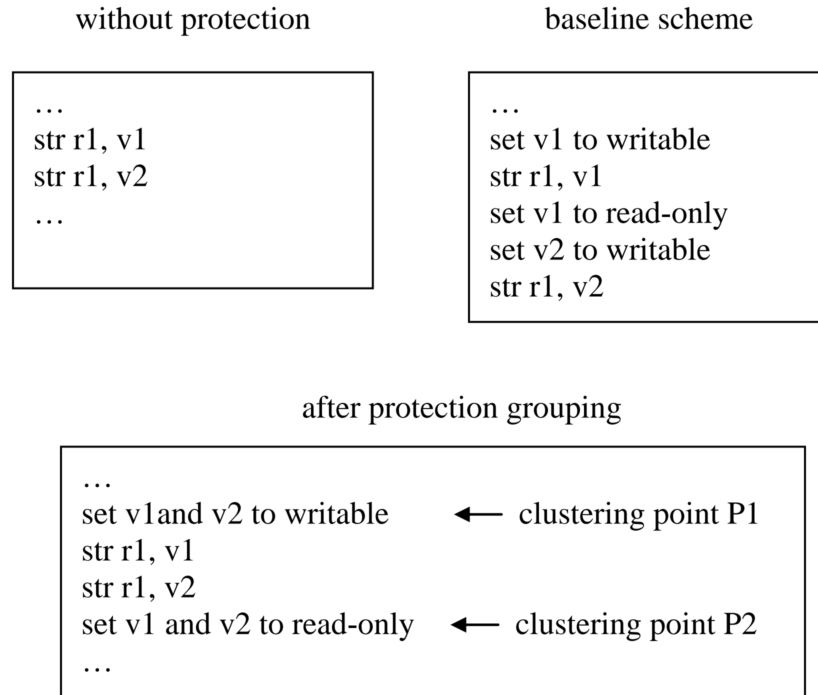
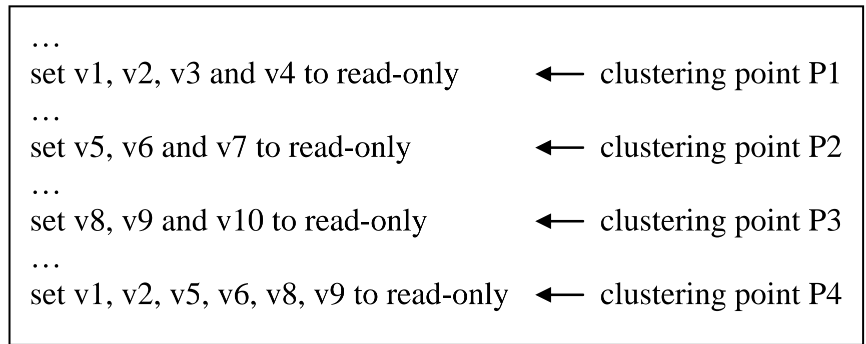


Figure 46. An example of protection operations grouping.

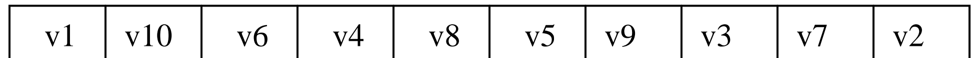
After clustered protection operations are identified, we need to properly lay out the memory objects to exploit the optimization opportunities. Note that the memory objects layout is a global decision rather than a local decision with respect to a given protection point. Different memory layouts may be required to completely exploit protection operation grouping opportunities at different clustering points, but there could be only one memory layout for the program. Our goal is to find a layout of memory objects that can maximally exploit the protection operations grouping opportunities.

Figure 47 gives an example showing how the data layout impacts the protection operations grouping. There are nine variables and four clustering points in this example. Figure 47 (b) and (c) show two possible data layouts of these variables in virtual memory. Data layout 1 scheme only allows us to perform the protections on v5, v8 and v9 together

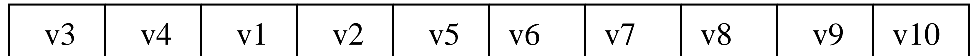
at P4 since only they are adjacent and there are protection operations on them at the same clustering point (i.e. P4). On the other hand, data layout 2 allows us to protect v1, v2, v3 and v4 together at P1; to protect v5, v6 and v7 together at P2; to protect v8, v9 and v10 together at P3; to protection v1, v2, v5 and v6 together at P4; and to protect v8 and v9 together at P4. The two data layouts show substantial difference in grouping protection operations, so our goal is to find out the best way to layout the data so that maximum protection operations can be grouped.



(a) clustering points



(b) Data layout 1



(c) Data layout 2

Figure 47. An example of data layout.

The pseudo code of our algorithm to determine the memory object layout is shown in Figure 48. First, a variable group at a clustering point is identified. All objects with the same type, whose access permissions are changed in the same way at a clustering point form the variable group for the clustering point. Types of an object include initialized global object, un-initialized global object, heap object, and stack object. So there are at most four variable groups at a clustering point. Access permissions can be changed either from read-only to writable or from writable to read-only. The weight of a variable group

is defined as the savings of dynamic protection operations if the protection operations for all variables in the variable group are grouped together at the given point. Then the algorithm starts from the variable group (say CurrVG) with the maximum weight. Note that we only deal with variable groups whose size is bigger than one since one element group does not provide any opportunity for optimization. Let ProcessedVG be the set of variable groups that have been processed. It is initialized to null. CommonVG includes variable groups that have common variables with CurrVG. The motivation behind identifying the set of common variables is to maximize linearize the variables in the group so that more protection operations can be removed. We illustrate these concepts below through an example.

We use the example in Figure 49 to explain our algorithm step by step. In this example, there are four groups VG1, VG2, VG3 and VG4 corresponding to the example in Figure 47. Their weights are decreasing. That means VG1 should be dealt with first, then VG2, VG3 and VG4. In processing VG1, both ProcessedVG and CommonVG are null, so VG1 is included in ProcessedVG. In processing VG2, since Processed VG only contains VG1 and there is no common variable for VG1 and VG2, CommonVG is also empty. VG2 is inserted into VG. Similarly, VG3 is also included in VG. Now VG contains VG1, VG2 and VG3. The last variable group is VG4, which has common variables with VG1, VG2 and VG3, so CommonVG includes VG1, VG2 and VG3.

```
Func:    GroupVar
Input:   AllVG – all variable groups
Output:  Optimized variable groups

ProcessedVG = null
While AllVG != null
   CurrVG = SelectMaxWeightVG
   AllVG = AllVG – {CurrVG}
   If (CurrVG.size > 1)
      CommonVG =    VG’ | (VG’ ∈ AllVG} & (VG’ ∩ CurrVG != null)}
      ProcessedVG = ProcessedVG – CommonVG
      If (CommonVG != null)
         ProcessedVG = ProcessedVG ∪ ArrangeData
      Else
         ProcessedVG = ProcessedVG  ∪ {CurrVG}
      EndIf
   EndIf
EndWhile
```

(a)

```
Func: ArrangeData
Input: CommonVG – variable groups having common element with CurrVG
Output: Arranged variable group

ListGroup = null
For each VG’ in CommonVG
   CommonVar = VG’ ∩ CurrVG
      For each v in CommonVar
      CurrVG = CurrVG – {v}
      ListGroup = ListGroup –VG’
      VG’ = VG’ – {v}
      NewListGroup = CreateListGroup(v, VG’)
      ListGroup = CombineListGroup(ListGroup, NewListGroup)
EndFor
```

(b)

Figure 48. Pseudo code for the grouping algorithm.

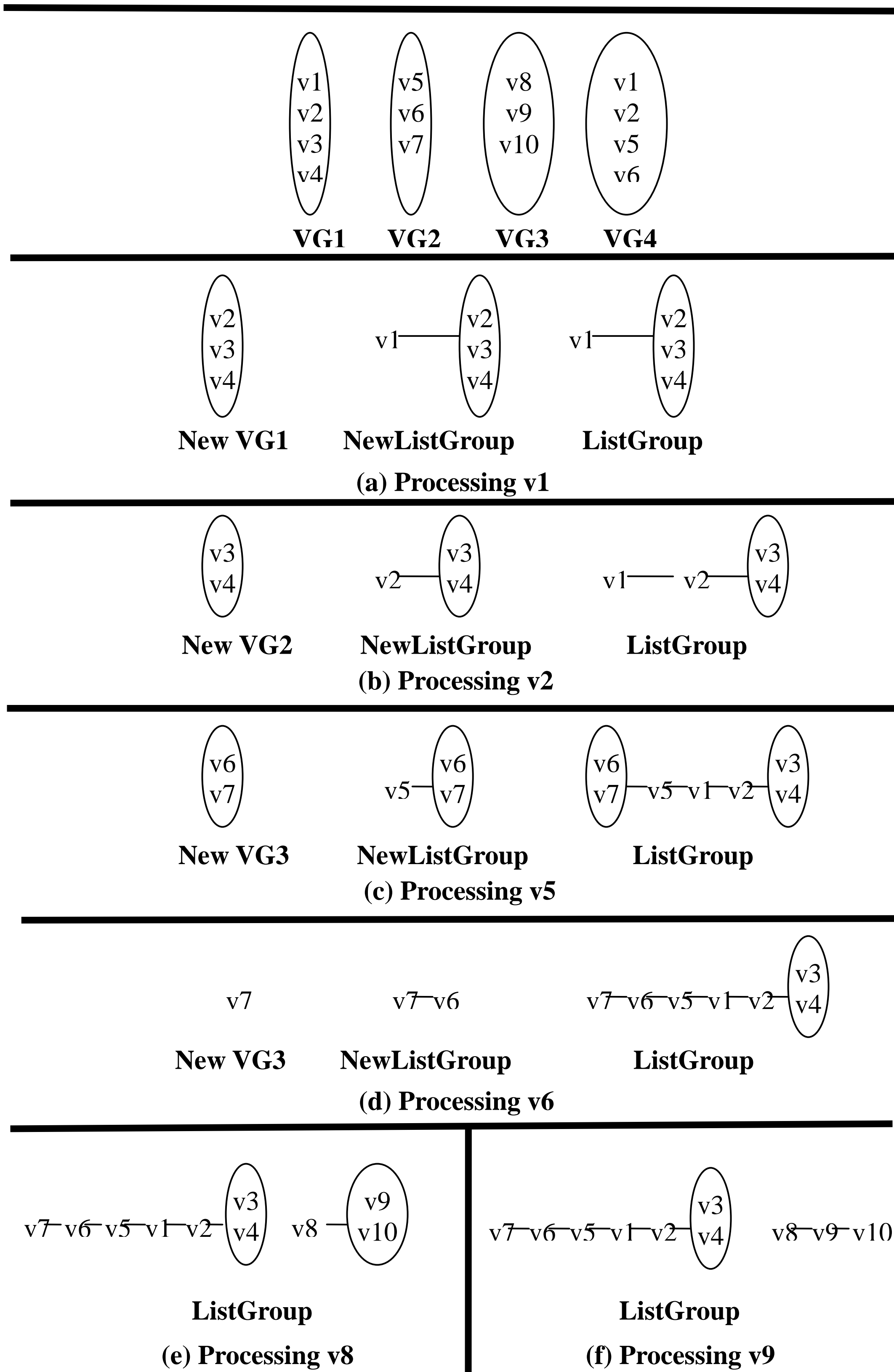


Figure 49. An example of variable grouping algorithm.

Next we show how to use the common variables to clarify the spatial relationship of variables. For every variable group VG' in CommonVG, let CommonVar be the set of common variables of CurrVG and VG'. For each variable v in CommonVar, it is taken out of the variable group first. The remaining variables in the variable group form a new variable group. Next v is connected to the new variable group to create NewListGroup that is in a special form – listgroup. Listgroup is a special structure containing both lists and groups. A group means that variables in it could be grouped together, but it does not tell us the exact layout of the variables, i.e. which variable should be adjacent to which variable in the virtual memory. So the variable can be linearized in many ways. But considering all variable groups, a variable should be preferred to be the neighbor of another variable to optimize overall protection operations. So a list is designed to represent the linear data layout of the variables. Finally NewListGroup is combined with original ListGroup to build current ListGroup.

To make it more clear, look at the example in Figure 49 again. VG1 is taken from CommonVG first. CommonVar includes v1 and v2. Initially ListGroup is set to be empty. v1 is first removed out of VG1. Note that although the variable group has priority based on their weights, variables in the same group have the same priority. That means it does not matter if we deal with v1 or v2 first. Now the new VG1 only has v2, v3 and v4. NewListGroup is created containing two elements, v1 and the new VG1. Then NewListGroup is combined with ListGroup. Since the original ListGroup is empty, current ListGroup is the same as NewListGroup as shown in Figure 49 (a). v1 is taken out as a special element connecting to the new VG1 since VG4 indicates that it is preferable for v1 to be grouped with other variables later, such as v2, v5 to reduce protection operations.

The next variable in Common to be processed is v2. In this case, VG1 is picked from ListGroup. Currently VG1 is the set of v2, v3 and v4, so ListGroup only has v1. Then v2 is taken out of VG1 connecting to the new VG1 creating NewListGroup. Next

NewListGroup and ListGroup are combined as follows. In CombineListGroup, if both NewListGroup and ListGroup have ending variables (such as v2 in NewListGroup and v1 in ListGroup) and they are both in CurrVG, then the ending variables are connected by an edge. An ending variable is a variable in a listgroup that has only one neighbor and not in a group. Thus, by connecting v1 and v2, we get the final ListGroup as shown in Figure 49 (b).

Besides scalar variables, we can also group array accesses. For example, if an array is written/read in a stride of 2, e.g., it accesses elements 1, 3, 5, 7, 9, 11…, we may group array elements 1, 3, 5 as a group and array elements 7,9,11 as a group. Array access analysis is an important topic in compiler optimizations especially for loop parallelization. It has been extensively studied in [53][21][12][23][58]. With array access information, array accesses can be grouped using standard techniques. For array accesses with a stride of 1, the problem is very similar to loop vectorization. For array accesses with a stride greater than 1, loop scatter-gather can be done first to create a loop accessing the same data with a stride of 1. Of course any transformation cannot violate the original program dependencies. All the techniques involved are elaborated in [69]. Being able to handle arrays is important to reduce security cost for the whole program since a large percentage of dynamic memory accesses go to aggregated data structures, most of them being arrays.

4.5 Data Structure Design

4.5.1 Action Table and Special Instruction

We now describe how the compiler actually inserts protection operations to be executed by the hardware component at runtime. The central data structure involved is an action table. The action table records which action should be performed at a given program point on a given memory object. An example action table is shown as Table 8.

The action table is a hash table that uses an instruction's PC address as its key. We always create a hash table without collisions to avoid the overhead of maintaining a spill list. Every table entry has three fields – action, memory object starting address, and memory object size. The action field has one bit – 1 denotes the action of changing the access permission from read-only to writable; 0 denotes the action of changing the access permission from writable to read-only.

A special instruction is inserted at every protection point to inform the processor that at the given program point, some protection operation needs to be done. Whenever the processor encounters such a special instruction, it uses its PC address to index into the action table and to execute the desired action.

Table 8. Action table

Action	Starting address	End address
0	Addr1	4
1	Addr2	8
0	Addr3	30
	…	

Another possibility is to insert instructions to implement the intended protection operations directly instead of recording them in a separate action table. However, the action would require several instructions to implement, which indicates more changes to the processor's standard ISA and more code space. Using a separate action table plus one special instruction appears to be a better solution to us.

The action table requires the start address and the size of the modified memory object for every action. The important problem is that the information may not be available statically. In general, during compilation, we do not know the address of the local stack objects and the heap objects. For some heap objects, we may not even know their sizes. To solve this problem, the action table has to be made writable and the compiler has to insert instructions to fix the action table for stack and heap objects. Such modifications of the action table occur after every dynamic memory allocation and every

function entry point. The inserted code will update the action table with dynamically obtained address and size when necessary.

4.5.2 Points-to Table

Dealing with pointer dereferences is a difficult problem. Static compiler pointer analysis has a lot of limitations and in some cases cannot determine an accurate points-to set for a pointer dereference. If the points-to set of a pointer contains multiple possible memory objects, then before the pointer dereferencing store instruction, the access permission levels of all possible pointed-to memory objects have to be set to writable. Also, after the pointer dereferencing store instruction, the access permission levels of all possible pointed-to memory objects have to be set to read-only. This could increase the protection overhead significantly. Moreover, in some cases, the points-to set can be very big, even including all possible memory objects, which means the compiler cannot derive any useful points-to information for this dereference. If we choose not to handle such cases; then we have to give up protecting the dereferencing store instructions and a large number of locations would remain unprotected which sacrifices security.

In our scheme, we use the pointer table developed in section 3.1. The inserted instructions are described in section 0. It is possible that a store instruction accesses multiple memory variables at different time. The dynamically maintained points-to information enables us to check store instructions regarding a program's current execution state. Thus, we could handle instructions that access different objects.

Not all information is available at compile time, and thus the tables have to be made writable, and the compiler has to insert instructions to fix the tables. However, making the table writable brings some security complications. Ideally, the tables should be put in the reserved address space, and made read-only to the user program, so that malicious corruptions could be avoided from the user program. Now, how to protect the tables becomes an issue. Our solution is to apply the protection mechanism in a hierarchical

way. That is, we use the same method to protect the tables residing in the memory. But in this case, the scale of the problem is much smaller, and the problem is much simpler. We know that the tables should only be modified by those fix-up instructions inserted by the compiler. We can regard one table as one single object and its address is predetermined. Thus, the additional checking table for protecting the original table can be easily constructed. It can be put in reserved address space, thus is not accessible to the user program, but only accessible to the hardware component. So it cannot be corrupted by the attacker. Then the original checking table can be protected properly.

4.6 Architecture Support

Figure 50 illustrates the necessary architecture support for our work. There are three major data structures. Action table and points-to table are hash tables as explained above. They are cached as data in data caches. The access bit table records the access permission level for every memory object. It is in a reserved space and can only be accessed by the hardware component of our protection scheme, thus is protected from tampering. Access bit table is large and accessed frequently. It has to be carefully managed to avoid significant space and performance overhead. A very similar problem exists in the Mondrian Memory Protection system work [76] and it provides an excellent reference on how to manage this large access permission table. We largely follow their design, regarding every memory object as a memory segment in their work. We deploy multi-level permission table with mini-SST entries that are elaborated in [76]. To improve performance, a protection lookaside buffer and sidecar registers are also deployed. Utilizing the design in [76] greatly reduces the space and performance overhead of our access bit table.

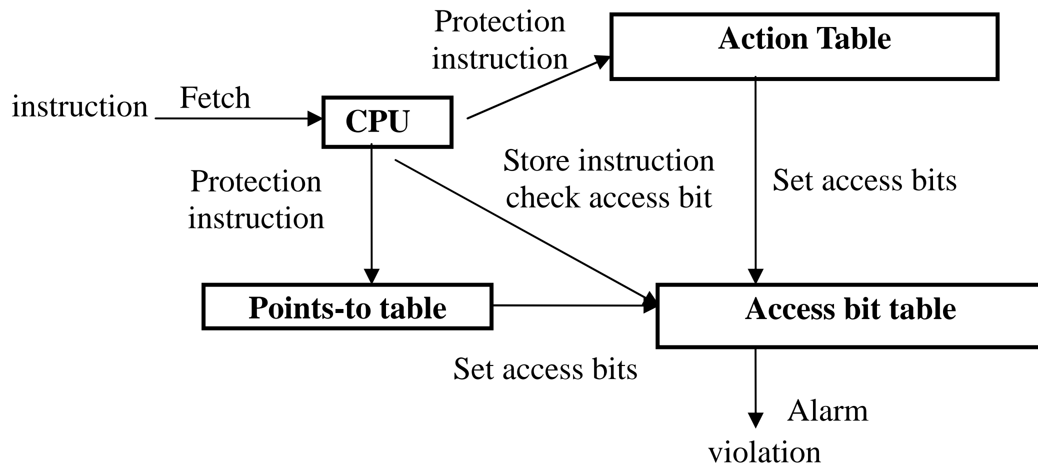


Figure 50. Architecture support overview.

There are three kinds of operations performed by the hardware component. Upon fetching a protection instruction for a none pointer dereference store, the processor uses its PC address as the key to index the action table. Then the access bits for the memory object will be set according to the action, the starting address and the size. The access permission bits are set for protection instructions on pointer dereferenced memory objects based on the points-to table. Another case is when a store instruction is fetched. The processor checks the permission bits. If the instruction violates the access permission, then an alarm is raised indicating the program is under attack.

4.7 Baseline against Attacks

In this section, we revisit the example attacks in section 4.2.5, and show how our scheme could detect all of these attacks successfully. In Figure 39, the buffer overflow attack takes place through the library function scanf. Although the second parameter passed to the function scanf is an array, in the function scanf, the data type of the second parameter is pointer. At the statement S1, the function scanf read the user input, and copies it to the buffer NewUser.username using a while-like statement. In the while-like statement, the pointer NewUser.username is increased by one in one iteration, and its

points-to information is updated using the case “q = p + r”. The details of scanf could be found in the file scanf.c that is contained in the software utility library glib. The basic process is that the data is transferred from the I/O to a buffer by increasing a pointer (say str) that is pointing to the buffer. The essential part of our detection mechnism applied to this scenario is checking the store instruction (say store_inst) that defers the pointer str. We have the start address and the end address information to restrict the pointer dereference. When the instruction store_inst is going to be executed, the memory object is set to writable. The assertion checks the access bits and finds that store instruction is going to access memory out of bounds. Thus, the attack is detected.

Similarly, we could catch the double free attack. In this example, when NewUser.dir is freed once, its begin and end addresses are invalid. Thus, the second free function call on it is claimed as illegal.

The off-by-one bug could also be caught by our solution, since the write instruction exceeds the pointer’s access range. This proves that providing both fine level analysis and dynamic pointer tracking would offer a fine grained memory protection solution.

4.8 Evaluation

Table 9. Ref input set of benchmarks

Applications	Input set
Bzip2	input.program
Gzip	nput.program
Mcf	inp.in
Parser	ref.in
Adpcm(en)	clinton.pcm
Epic	lana.tif
G721(en)	clinton.pcm
Mesa	None
Mpeg(encoder)	bitstream of YUV components
Pegwit	public_key,encryption_noise_file, plain_text

Our experiments are based on x86 Pentium architecture. The compiler work was implemented using the MACH SUIF compiler. The benchmarks are from SPEC2000

integer benchmarks and MediaBench embedded benchmarks. The ref input sets for these benchmarks are shown in Table 9.

4.8.1 Performance Measurement

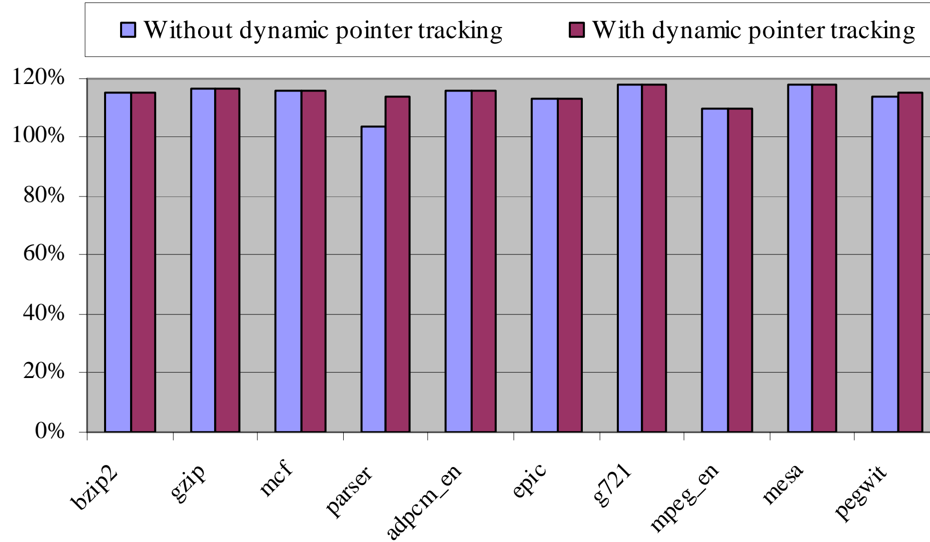


Figure 51. Performance degradation.

Figure 51 shows the performance degradation under our scheme. All hardware support modeling is done inside SimpleScalar targeted to x86. We did two set of experiments. Both of them include the optimizations described in sections 4.4.4, 4.4.5, and 4.4.6. The difference between the two sets of experiments is that the first experiment does not have dynamic pointer tracking that is our baseline; the second one enables dynamic pointer tracking with the possible optimizations. The Performance numbers are normalized to the original program binary without protection. From the results, the average performance degradation under the baseline protection scheme is 14%. The performance degradation mainly comes from the overhead to access the access bit table. Other sources of performance degradation include accessing the action table and the points-to table, update the points-to table, and executing the compiler inserted instructions to fix up the action table. The pointer table overhead is small except for the

parser benchmark, which makes intensive use of pointers throughout the whole program. The average performance degradation with dynamic pointer tracking is 15%. The less than 1% dynamic pointer tracking overhead is tolerable if we could achieve big security improvement as shown in section 4.8.2.

4.8.2 Security Measurement

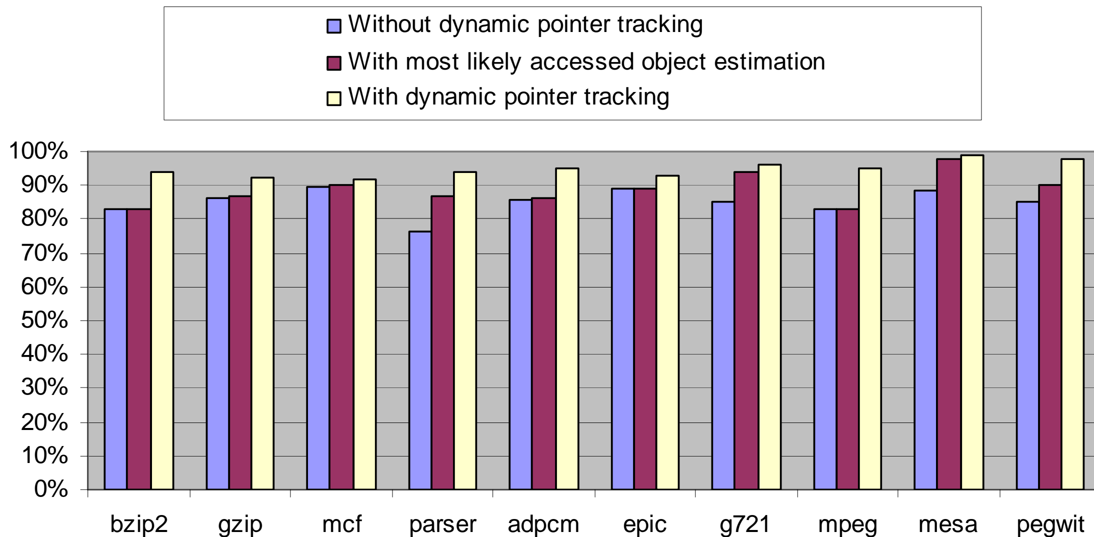


Figure 52. Security measurement

Figure 52 shows the security measurement. Under our baseline protection scheme, the access permission level for a memory object is set to writable just before a store instruction writing to it and set to read-only right after the store. Therefore the baseline protection scheme can detect all memory corruption attacks exploiting the flaws in the program such as the buffer overflows and double free vulnerabilities, since an attack needs a store instruction to do the tampering and that attacking store instruction has to be between the pair of "set to writable" and "set to read-only" operations. This corresponds to the 100% protection shown in Figure 52. Due to the motion and removal of protection points caused by compiler optimizations, some memory locations could be left unprotected. The reason is that another store instruction str2 may be between the "set to

writable" operation and the "set to read-only" operation for str1 now, and str2 could be the attacking store instruction and could possibly tamper the memory object to be written by str1. Also, if a pointer points to an unexpected memory object at runtime, the write will not be checked against the access permission table, thus there could be false negatives too. We count the number of dynamic store instructions like str2, then subtracted those from the total dynamic store instructions to measure the protection strength. As per this calculation, a higher number of dynamic store instructions after subtraction means the higher the protection strength is. The results in Figure 52 indicate that the compiler optimizations do not lead to much security degradation. With dynamic pointer tracking, the average security strength is 92%, which is 7% higher than without dynamic pointer tracking, and 3% higher than the estimated most likely accessed object approach. This is due to the fact that in most of programs, there are much more pointer dereference stores than pointer assignments. In the benchmarks, many pointers are used to access buffers. The auto increment update removal optimization eliminates all updates for such pointers, and save enormous cycles. Even with the optimized dynamic pointer tracking, the security could be maintained at the same level. Those schemes to improve the performance like protection point hoisting, grouping, and removal are not viable since we can not scarify security for performance, and that is why we thought of dynamic pointer tracking and its optimizations. With the points-to sets, we get security enhancement (and no degradation), and we are able to tackle the overheads as well.

It should be noted that the above method to measure protection strength represents the worst case. The above method assumes that every store instruction moved into a pair of "set to writable" and "set to read only" operations could be an attacking instruction or an attacking point. In reality, normally a program only has a few possible attacking instructions. For example, for a buffer overflow attack exploiting the strcpy function, the store instruction in the strcpy function implementation is the possible attacking instruction. As long as our optimizations do not move that possible attacking instruction

into a pair of "set to writable" and "set to read only" operations (which is very unlikely), the security will not be harmed. Thus, our scheme has a much stronger detection strength against real-world attacks than represented by the above worst case.

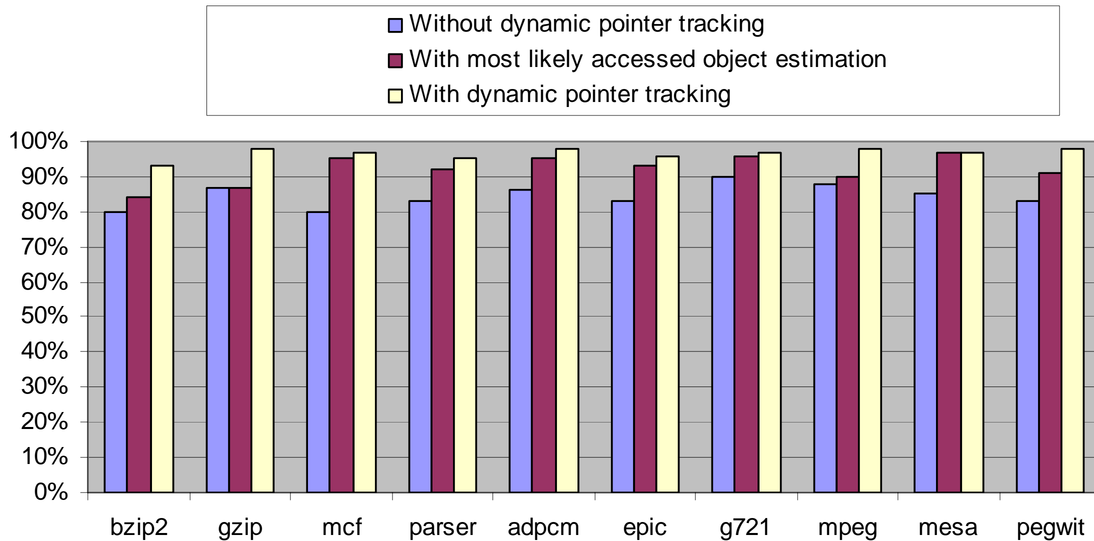


Figure 53. Detected simulated attacks.

We further performed simulated attacks to our benchmarks. We run the benchmarks in the simulator and then randomly tamper a memory location by making a store instruction write to a different memory location to see whether our scheme is able to detect the tampering. For each benchmark, we perform such simulated attacks by tampering at 1000 different memory locations. In our experiments, we assume that every store instruction could be an attacking instruction, so again our results represent the worst case since the possible attacking instructions in a program are very limited as discussed above. Figure 53 shows the percentage of attacks detected. On average, 8% more randomly injected memory tamperings are detected with dynamic pointer tracking than without dynamic pointer tracking, 4% more attacks are detected with dynamic pointer tracking than with most likely accessed object estimation. It shows that our scheme with a dynamic pointer tracker's help is very effective to protect memory tampering attacks.

Attacks including the access control and non-control data attacks against a number of real-world network applications are used to evaluate the effectiveness of the proposed technique in the following discussion. We test real-world attacks against network applications running on SimpleScalar augmented with our protection scheme. Briefly, we evaluate our work focusing on five types of attacks as shown below.

```
int mapping_chdir(char *orig_path){
   int ret;
   char *sl, *path;
   path = &pathspace[0];
   strcpy(path, orig_path);
   //path = abcdefgh
   while ((sl = strchr(path, '/'))) {
      char *dir;
      dir = path;
      if (*dir)
         do_elem(dir);
      }
}
```

(a)

```
Void do_elem(char * dir) {
…
If(!(mapped_path[0] == '/' &&    mapped_path[1] == '0'))
      strcat(mapped_path, "/");
      strcat(mapped_path, dir);
}
```

(b)

Figure 54. Buffer overflow in Wu-FTP.

Buffer overflow against static data. The Wu-FTP server is a popular FTP server. The buffer overflow vulnerability in the function do_elem is a breach in Wu-FTP [45] allowing attackers overwrite memory locations as shown in Figure 54. We designed a long string as the path name representing the series of directories to exploit this vulnerability. In the initialization of the FTP server, the path name (that comes from the user input) is passed to the function do_elem. In this function, the object mapped_path is

overflowed at the statement strcat(mapped_path, dir), since the object pointed by the pointer dir is larger than the size of mapped_path.

Buffer overflow against stack data. In the file ftpd.c in Wu-FTP, there are buffer overflow vulnerabilities in the function skey_challenge, which potentially can be exploited by malicious users to compromise the whole system. The variable *name is never subject to any boundary checking. It is possible to write beyond the buf[] array, overwriting the return address of the function, modifying the path of execution flow through the second sprintf function.

Off-by-one. The off-by-one problem discovered in Wu-FTP [83] was tested against out technique. This bug could be easily detected by our system through the access range checking.

Double free. The double-free bug in MIT Kerberos [82] was detected by our technique successfully since the first free function call set the pointer to null, and the second free function call is checked against null parameter.

4.8.3 Space Cost Measurement

Table 10. Space cost measurement.

Benchmark	Action table size (byte)	Access bit table size (KB)	Points-to table size (byte)	Code size increase (byte)
Bzip2	14016	147	472	7018
Gzip	13216	159	496	11520
Mcf	21248	306	338	29477
Parser	62272	323	1222	45138
Adpcm_en	672	24	30	349
Epic	10784	160	222	5395
G721_en	3200	59	100	1610
Mpeg_en	24864	89	318	12436
Mesa	770976	79	106	385488
Pegwit	40096	77	248	20057

The space cost is shown in Table 10. There are four fields in this table. The first three fields represent the sizes of the action table, the access bit table, and the points-to table. The last field shows the total code size increase of the program due the insertion of protection operations. The additional action table to protect the original action table only requires a little space, so we do not show its size here. We use a multi-level permission table. The average sizes of the action table, the access bit table, the points-to table, and the code increase are 96134 bytes, 142 KB, 355 bytes and 51848 bytes respectively. Some of the access bit table sizes are small, such as the adpcm encoder (adpcm_en) and g721 encoder (g721_en). This is because these applications have only a few objects and these objects are big in size. The access permissions for one object are compacted using the Mini-SST technique, which could save a lot of space. The average size of points-to set for each pointer dereferencing store instruction is 1.58. The results show that the space cost of our protection technique is at most several hundred kilo bytes, which is very acceptable for modern computers.

4.9 Summary

Memory protection at fine level against non control flow data attacks is important, but so far there has not been a solution that offers full protection. Since the pointers important practical languages such as C/C++ allow them to visit any memory location, lack of such a solution leaves the software written in these languages vulnerable. In this work, we tackle the problem of detecting memory attacks with dynamic pointer tracking. We propose a compiler baseline framework that refines the points-to set dynamically to detect memory access violations. Architectural support is designed to optimize the scheme and to reduce the runtime overhead. On modern processors, the additional space overheads due to the protection tables are negligible allowing the performance degradation to be within limits. The other overheads (due to the checking and updating operations) could be minimized through a careful data-structure consideration coupled

with clever placement of operations. Thus, the work shows that on modern processors, it is possible to achieve memory protection at fine level by using a combination of dynamic pointer tracking at a reasonable performance penalty.

CHAPTER 5

GARBAGE COLLECTION WITH DYNAMIC POINTER TRACKING

5.1 Introduction

Garbage collection is the process of detecting useless objects in memory, and reclaiming the resources allocated for these useless objects. It is used to help programs become more stable, to detect memory leaks, to identify program bugs, and to improve the performance. Garbage collection has been studied for a long time, and many programming languages, like Java, C#, Modular-3, and C++, tend to support garbage collection in some environments. Garbage collector is usually implemented with the compiler, the memory allocator, and the runtime system support.

5.2 Traditional Garbage Collection Techniques

There are two major garbage collection techniques. We compare them in the following.

5.2.1 Reference Counting Collectors

Reference counting is a kind of garbage collection technique. It counts the number of references to an object. When a reference to an object is created, its reference count is increased, and when the reference is gone, the reference count is reduced. An object is declared as garbage if its reference count is zero.

The advantages of reference counting collectors are obvious. First, the garbage objects could be reclaimed immediately after they become garbage. Thus, the program is not paused for a long time, which is desired in real-time systems for live services. Second, it could be applied to both user programs and operating systems to detect memory leaks. Third, it requires simple implementation.

Reference counting collector has two disadvantages. First, it cannot deal with reference cycles very well. A naïve reference counting collector can not reclaim cyclic data structures since every object in a cyclic data structure has at least one reference count. Second, the runtime overhead is huge if the reference count is updated frequently. A large number of pointer copies and an inefficient reference counting algorithm could significantely degrade the performance. Third, the space cost is a concern for big applications. In reference counting collector, every object in memory has a tag indicating the number of reference to it. The reserved space could influence the cache performance and memory usage if the number of objects is big.

Reference counting was first invented by Collins [18]. Improvements on the original reference counting technique were proposed in the literature [54][16][7][8][10][55] [25][4][75][39][54]. Baker [7] studied garbage collection and discovered that most of references come from local variables that have short life time. Deferring the update for these references to necessary positions would eliminate a number of overheads. It is critical to keep the reference counts in a consistent state by deferring the update. Levanoni and Petrank [46] developed a framework aiming at multi-processor systems. They removed the barrier of operation synchronization by providing a snapshot of the heap.

5.2.2 Tracing Collectors

Tracing garbage collector is another kind of garbage collector. It relies on object reachability analysis. First, a tracing garbage collector identifies root objects, which are usually the global objects, and the objects on the call stack and registers. Then, the collector starts from the root objects and get a transitive closure of all live objects. Finally, the objects that are not reachable from the root objects are determined to be garbage.

The basic algorithm is called mark-and-sweep, which performs garbage collection after a certain period or when the system is running out of memory. Every object in memory has a tag indicating if it is being used or not. Between two garbage collections, the tag is cleared. Note that the tag is usually one bit that is smaller than the reference count used in the reference counting collector. In garbage collection, the program or the system is suspended. Then, it sweeps the root objects, and set their flags. Next, the algorithm continues mark objects reachable from the root objects. Finally, it scans the memory again to reclaim the space occupied by the objects that are not flagged. Thus, it requires several memory scans.

The advantage of tracing collectors is that it is more efficient compared with reference counting collectors since there is not runtime update. But they have two main problems. First, the program has to be paused during garbage collection, which is not tolerable for some programs and systems, though it could be improved by only collecting newly created objects[56]. Second, it requires at least one flag for one object.

5.2.3 Challenges of Garbage Collection for C and C++

One big problem in implementing the mark sweep collector for programs written in programming languages like C and C++ is how to locate the root pointers. Unlike Java, C and C++ allow a programmer to define and manipulate pointers freely. The compiler could identify pointers easily based on the type information. However, at runtime without the type knowledge, a conservative way has to be developed to tell if the data in memory is a pointer or not. The state of art garbage collectors for C and C++ could not solve the problem completely. Most of the approaches either make some assumptions regarding the representation of pointers or generate some rules to locate pointers.

A Boehm-Demers-Weiser garbage collector [84] was proposed by HP lab. They implemented a mark-sweep style collector as a library. They determine whether the data is actually the address of a heap object by three filtering passes. First, the candidate

pointer is checked against the heap bounds. Then, the candidate pointer value is used to identify the page where it exists and its offset within that page. The page address is looked in a table to return either an object descriptor or a value indicating that if it is not an actual pointer. Next, the object descriptor is used to locate the object in memory. Their approach is based on the assumption that the data whose value falls in the range of collectable objects should be a pointer. However, the data could happen to be the value within the range while it is not actually a pointer.

5.2.4 Our Contribution

Garbage collection seems very promising to future programs developed in C/C++. With the wide spread of the 64-bit architectures and the fast growth in use of very large heaps, especially for game development, the traditional reference counting and tracing algorithms developed for type safe languages like Java are not sufficient to develop efficient garbage collectors. Dynamic pointer tracking provides a novel solution in garbage collection. It is not necessary to scan the whole memory searching for live heap objects as previous work does. The points-to information in the pointer table could help us exact the live objects. With further optimizations, the dynamic pointer tracking technique could be polished ameliorated for garbage collection.

Our approach has several obvious advantages over both reference counting and mark sweep garbage collectors. Compared with reference counting garbage collectors,

(1) We could deal with cyclic data structure since we use the global/static pointers and pointers on the stack as the root to get a transitive closure of live objects.

(2) Our runtime overhead is less than reference counting. In our baseline, for one pointer assignment, we have two operations (update the pointer’s points-to object start and end addresses). In reference counting, one pointer assignment (i.e. the pointer is switched from an old object to a new object) results in more than four operations (decrease the old object’s counter, increase the new

object counter, and checks if the old object's counter is zero; if so, the object is deleted), which includes one conditional jump.

Compared with traditional tracing collectors,

(1) We have less pause time since not all memory need to be scanned.

(2) We identify pointers through the pointer declaration, not by the garbage collector based on some manually defined rules.

(3) Our dynamic pointer tracking solution is compatible with previous tracing collectors.

5.3 Garbage Collection Algorithms

Our garbage collection is divided to two steps. First, the user selects places in the program where to mark garbage objects. Another choice is to enable the mark process when new objects are created. Second, the system notifies the collector when the system has idle time for reclaiming the garbage memory. In this section, we only focus on the marking step in mark-sweep style collectors. We introduce three garbage collection algorithms using the customized dynamic pointer tracking scheme.

5.3.1 Maintaining Object Table

The straightforward way to do garbage collection is that for every object, maintain the whole transitive closure of pointers pointing to it. In this case, the reachability information is important. One naïve solution is that the full reachability information is stored in an object table. One table entry corresponds to one object including the pointers pointing to the object and the pointers contained in the object. If there does not exist any pointer pointing to the object, it is determined to be garbage.

Figure 55 shows one example object table. The table is indexed with an object id. Every object has a unique object id. One entry contains two fields, in pointer and out pointer, both of which are represented using linked lists. The in pointer list contains the

pointers that are reachable to the object; the out pointer list contains the pointers that exist in the object. We have two methods to update the in pointer list. The first one lives without the pointer table. For a pointer assignment p = q, it scans all in pointer lists looking for the pointers p and q. Then it removes p from the in pointer list, and appends it to the in pointer list that contains q. This design is awkward and slow. The second algorithm seeks help from the pointer table. The pointer table remembers the pointer's target object. When a pointer is assigned, the pointer table is looked up to get the pointer's target object in the object table, then the pointer is removed from the old object's in pointer list and inserted in the new object's in pointer list. Using two tables could speed up the updating process. The out pointer list could be generated statically by the compiler for global and local variables. The pointers in dynamically allocated object could be obtained after the object is created.

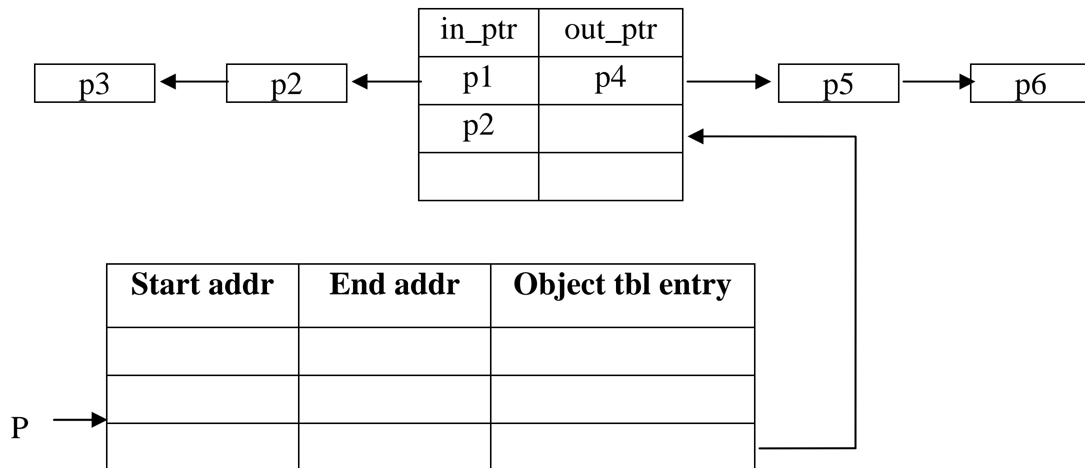


Figure 55. Example object table.

The out pointer list is used to aggressively collect garbage objects. Consider the example shown in Figure 56, when the function foo returns, the pointer head, curr_node, and prev_node are destroyed, and their points-to information are invalid. The data relationship is shown in Figure 57. The deadness of the pointer head results in the

deadness of the object O1. Claiming the object O1 as garbage voids the pointer in it. Thus, the other objects in the chain could be recycled recursively.

```
foo () {
        Node * head, * prev_node, * curr_node;
        head = (Node*)malloc(sizeof(Node));
        prev_node = head;
        for(i = 0; i < 5; i++) {
                curr_node = (Node*)malloc(sizeof(Node));
                prev_node.next = curr_node;
                prev_node = curr_node;
        }
}
```

Figure 56. Cyclic data structure example code.

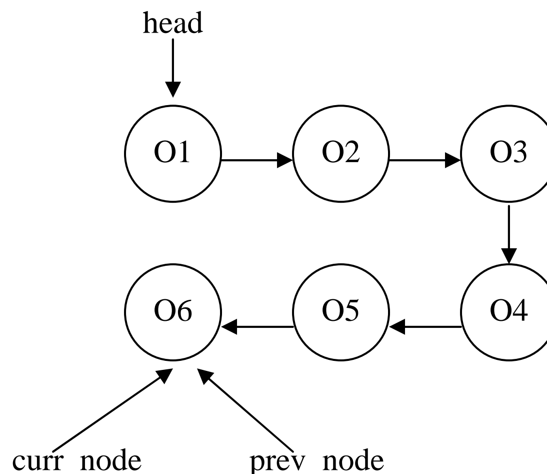


Figure 57. Cyclic data structure graphic view.

This design could be considered as another form of reference counting. The advantage of this scheme is that the garbage object could be detected instantly and the memory location it occupies could be reclaimed immediately. The disadvantages include the tremendous overheads and the disability to handle circles. In the example shown in Figure 57, suppose that there is an arrow from the object O6 to the object O1. Even if the program exits in the function foo, and the references through the pointers head, curr_node, and prev_node are removed, the cyclic data structure remains live.

5.3.2 Maintaining Pointer Table

The second algorithm is using the pointer table to determine if an object is live or not. The pointers are classified in two categories – class one contains pointers on the heap; class two includes the other pointers like global pointers, static pointers and pointers on the stack. The pointer table could be used in mark-sweep style garbage collectors. At the collection cycle, the pointer table is scanned starting from the pointers of class one (root pointers). Then, using the referent pointer information in the pointer table, other pointers that are reachable to the root pointers are identified. The union of the pointers access ranges is the live memory locations; other allocated memory space could be recycled.

The advantage of this approach is that the pointers are located at compile time. It removes the barrier of locating pointers and object bounds using some manually determined rules. Through smart insertion of the updating code, we could update the points-to sets efficiently.

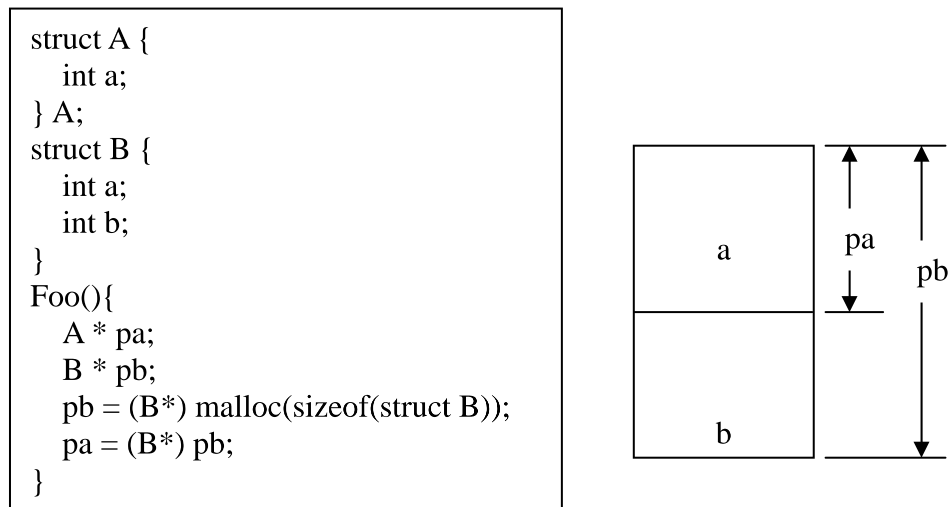


Figure 58. Garbage collection with pointer casting.

Pointer casting caused problems in garbage collection as shown in Figure 58. A pointer pa is declared with the base type A, but it is cast and points to an object of type B. In this case, to achieve garbage collection safety, the dynamic pointer tracking algorithm assumes that the object pointed by pa is of the type B. In other words, down casting enlarges the data pointed by a pointer.

5.3.3 Maintaining Pointer Address

The last algorithm co-operates with the garbage collector from HP labs. The main idea is to collect the root pointers in a program. To identify where a pointer is, we design a table only remembering the address of pointers. The pointer table is still indexed by a pointer id. The statically and dynamically declared pointers are managed in a similar way as shown, before except that a pointer table entry contains a pointer's address, not the pointer's target.

The advantage of this algorithm is that the runtime overhead of tracking the pointer location is very little. At the places where a pointer is created/destroyed (usually through malloc-like and free-like functions), the pointer table is updated. The little overhead could improve the garbage collector significantly. First, the root pointer information is available in the pointer table, thus the step of scanning the whole stack and the registers is bypassed, which speeds up the marking process. Second, recording the root pointers in the pointer table is more accurate and safer than using some rules or based on the pointers value.

This garbage collector is conservative to achieve memory safety. We consider the pointers in the pointer table as the candidates to do garbage collection. Other pointers that are not tracked, we assume that the objects pointed by them are live. In other words, only the objects that are pointed by the dead pointers in the pointer table are collected. This is a conservative way to achieve safety. The specification of dynamic pointer tracking safety is a promise to program analyzer and optimizer. If all program's actions are legal, then dynamic pointer tracking will remain transparent to the program. Dynamic pointer tracking is transparent if it does not change the program semantics, and it does not make invalid change to the program data. Coexistent problem could result in undetected dead objects pointed by untracked pointers.

5.4 Heap Shape Analysis Results

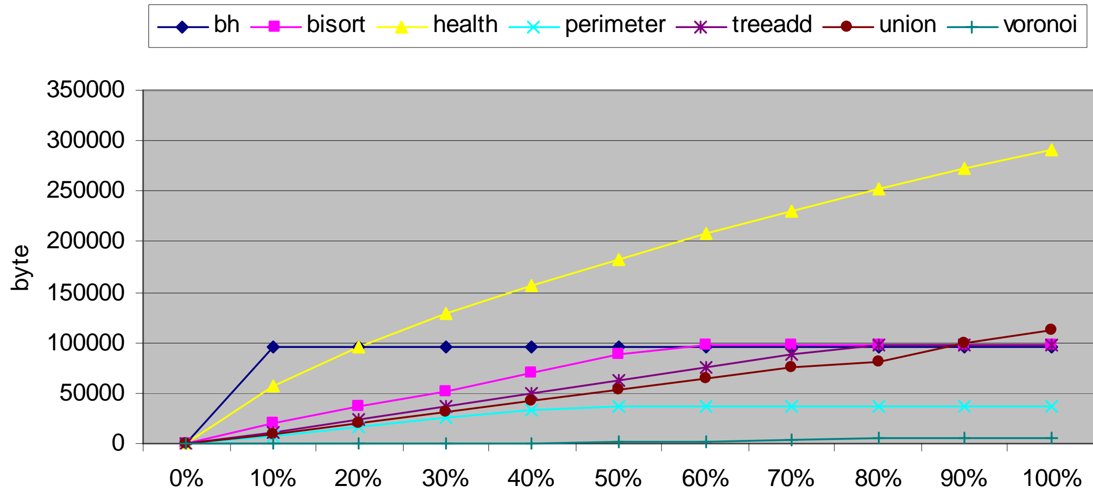


Figure 59. Heap object size.

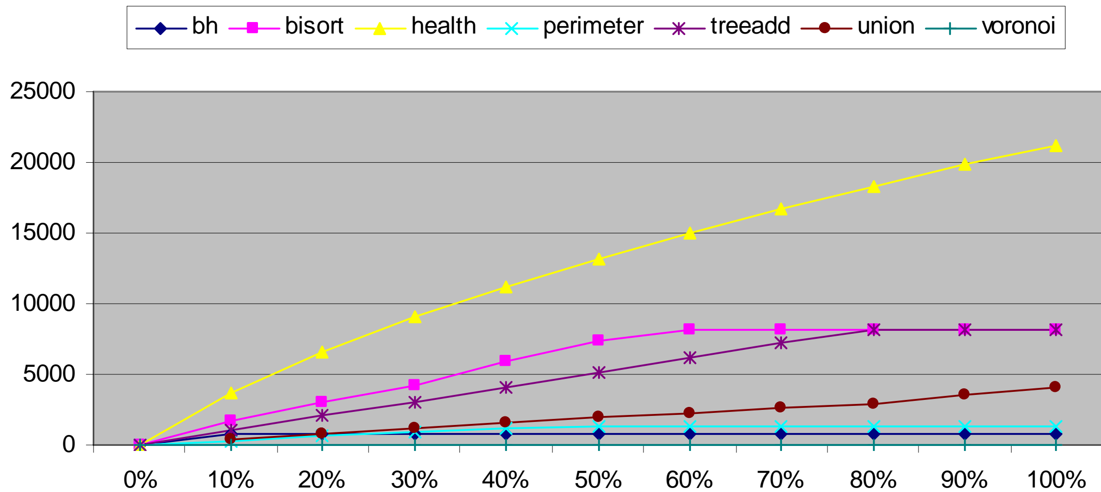


Figure 60. Number of heap objects.

Figure 59 to Figure 61 reports the heap shape analysis results. The heap objects are the user declared heap objects. Most of the heap objects are live throughout the whole program. It is not hard to have following observations:

(1) The heap object size increases steadily as the program executes as shown in Figure 59. The bh program gets to it peek heap object size at the early stage of the program. The health and union programs almost linearly creates heap objects as it executes. The voronoi benchmark creates a small number of heap objects, which

is not noticeable in the chart. The other benchmarks stop allocating space for heap objects in the middle of their execution.

(2) These applications have a big number of heap objects as shown in Figure 60, ranging from 59 to 21216.

(3) The average heap object size is almost constant per application (except voronoi) as shown in Figure 61. This is due to the fact that most of those applications create heap objects of the same type. The heap object size is relatively small.

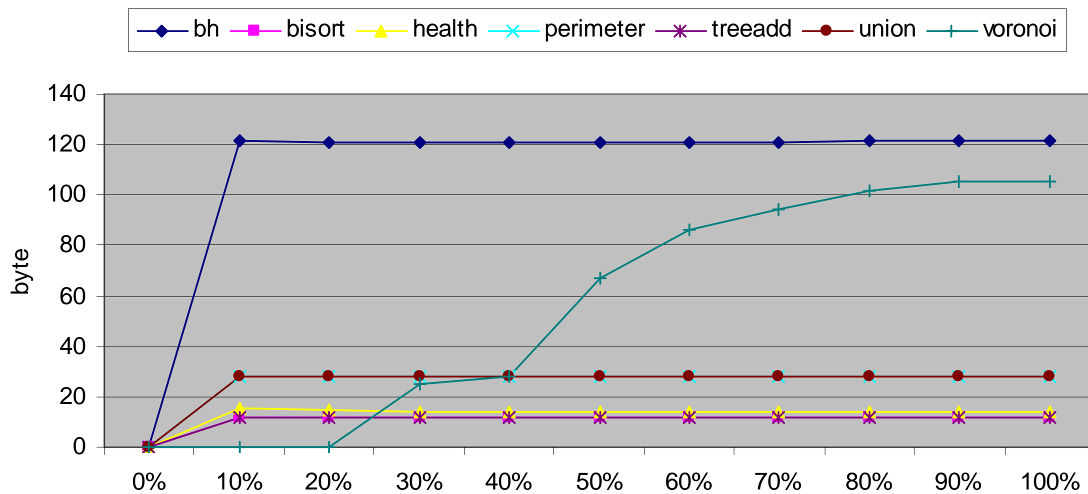


Figure 61. Average heap object size.

5.5 Garbage Collection Results

There are not good garbage collection benchmarks in C. The only one we find is the artificial benchmark from HP Lab [85]. It builds binary trees by allocating a numerous heap objects, and set the tree root to null, that results in all nodes in the tree being dead. The number of heap objects grows linearly with the program's execution time. This is a good study case to test the ability of garbage collectors.

We implemented two algorithms described in section 5.3.2 and 5.3.3. The garbage collector though maintaining the object table as shown in section 5.3.1 involves enormous performance degradation (more than 3X) on the artificial garbage collection

application, so it is not the focus of this section. The number of dynamically executed instructions is increased by 35% and 5%. The space cost of the garbage collector by maintaining the pointer table is dominated by the pointer table size. Every entry has eight bytes storing the start and end addresses. The program does not have pointer chasing, so the third entry is not necessary. The original program without garbage collection or user written free function calls consumes 490.7M bytes heap space. The pointer table size is 13.6M bytes and 6.8M bytes, that is 2.8% and 1.4% of the heap space occupied by the program. The other optimizations except update propagation are not applied to the benchmark since the conditions are not satisfied. This application is selected only to see the feasibility of our garbage collection, not the performance aspect. Note this is an extreme case, in most of applications like the spec200 benchmarks, only a few pointers are deployed, thus the pointer table size is relative small. Figure 62 shows the number of garbage objects detected by both of the garbage collectors. We could detect all garbage objects in the artificial garbage collection benchmark.

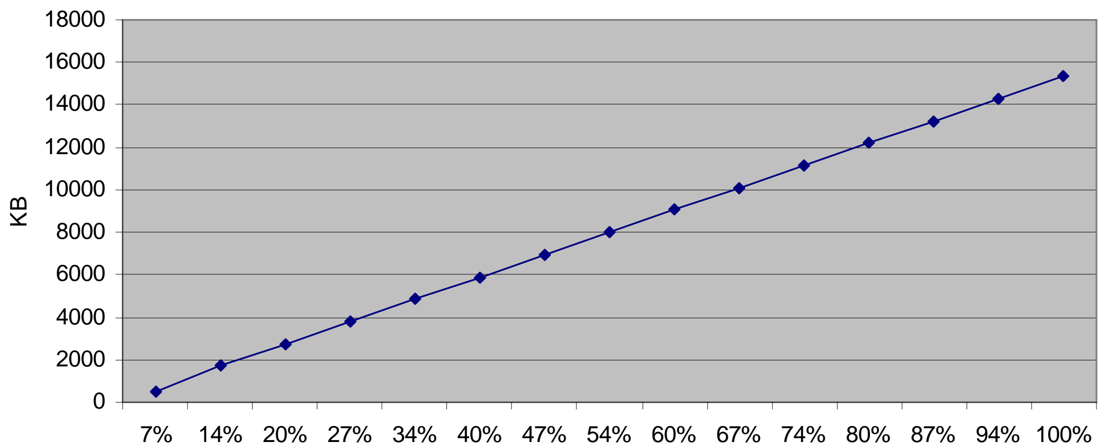


Figure 62. Garbage objects detection result.

5.6 Summary

In this work, we propose several garbage collection algorithms based on dynamic pointer tracking. This demonstrates the feasibility of applying dynamic pointer tracking

in real world applications. It also shows that the pointer information at different levels could be crafted and customized according to different requirements. It proves the flexibility and portability of our tracker.

CHAPTER 6

DYNAMIC INVARIANT PREDICATE IDENTIFICATION WITH DYNAMIC POINTER TRACKING

In this section, we present our work regarding dynamic invariant identification. In particular, we propose a framework that aims at identifying more invariant variables based on the runtime information, and thus recognizing more invariant predicates. Significant opportunities exist at runtime to identify invariants that static analysis is unable to tackle. To achieve the goal, we perform two types of analysis – Static and Dynamic. Each of the analysis is explained below.

6.1 Static Analysis

The goal of static analysis is to identify potential candidates for dynamic analysis. This is done to reduce the dynamic analysis overhead. Specifically, for each loop, we first analyze the predicates (branch conditions) present in them. The outcome of this analysis is used to classify the predicates into two groups: Invariant and Maybe Invariant.

A predicate is classified as invariant if all of its operands do not have loop carried definitions. In other words, all definitions of the variables used in a predicate are generated outside the loop, which include those definitions through pointer dereferences. If there exists at least one path inside the loop; along the path all of a predicate's operands are not defined, then the predicate is said to be maybe invariant. In this case, all predicate operand definitions are guarded by invariant predicates. The main idea of this classification is to expose a secondary effect of the invariant predicate evaluation. In case paths holding definitions of the maybe predicate operands become dead, the maybe invariant predicate might become dynamically invariant and further opportunities for removing dead paths are exposed. We call this as the secondary effect. The classification of predicates is performed by analyzing reacheable definitions of each variable involved

in a predicate. Then for each maybe invariant predicate, we store the predicate operands definitions along with the corresponding basic blocks where they are defined.

At the end of the static analysis, we have a list of invariant and maybe invariant predicates. For each invariant and maybe invariant predicate, we generate the code in the loop pre-header evaluating each predicate. The code generated, would be executed at runtime and a dynamic analysis would be performed based on the predicate evaluation.

6.2 Dynamic Analysis

In the dynamic analysis, we first perform the standard constant propagation based on the parameter value (note that many function parameters are input only value and serve as dynamic constants in an execution instance of the function). Then the interface function is invoked at runtime. Next the interface function, present at the loop pre-header is evaluated as in Figure 63. The invariant predicates are evaluated to identify dead branches. Then we check if a maybe invariant predicate becomes an invariant one. This is accomplished by executing the algorithm given in Figure 64, Figure 65 and Figure 66.

In maybe invariant checking, we first evaluate branch conditions. Based on the evaluation results, all edges along which the execution cannot proceed are collected as NotTakenPaths. If the block B has zero predecessors, then it is marked as not executed block. We repeat the above process on B's successors. This iterative procedure is accomplished using a stack. _push() and _pop() are the corresponding functions used for storing and retrieving the basic blocks that need to be processed.

Such recognition of dead paths may in turn lead to other predicates becoming invariant, which is called the secondary effect. This might happen when the definition blocks of predicate operands are in fact on dead paths. In order to take advantage of the secondary effect, we keep a list of maybe invariant predicates along with the set Def(p), that contains definition blocks of operands in a predicate p within a loop. All marked blocks are removed from the Def(p) set. If this removal leads to Def(p) becoming an

empty set, then the maybe invariant predicate p becomes an invariant one and is added to the invariant predicate list. The analysis terminates when there is no invariant predicate to be processed anymore. Finally the invariant predicate list contains all invariant branch conditions.

```
function: DynamicInvarIdentification()
Input: InvarList, MaybeInvarList
        InvarList is the list of invariant predicates
        MaybeInvarList is the list of maybe invariant predicates
Output: List of dynamic invariant predicates

ProcInvarList()
While(InvarList)
   ProcMaybeInvarList()
EndWhile
```

Figure 63. Dynamic invariant predicate identification algorithm.

```
function: ProcInvarList()
Input: CFG, InvarList, S(Stack)
        CFG is the control flow graph of a given function
        InvarList is the list of invariant predicates
        S is the stack for blocks to be processed
Output: Basic blocks in CFG marked as dead

For i in InvarList do
   NotTakenPaths = _eval(i)
   For e(Bp;B) in NotTakenPaths
     _push(B, S)
   EndFor
   ProcBBStack()
EndFor
```

Figure 64. Process invariant predicate list.

```
function: ProcMaybeInvarList()
Input: CFG, InvarList, MaybeInvarList, Def(P)
        CFG is the control flow graph of a given function
        InvarList is the list of invariant predicates
        MaybeInvarList is the list of maybe invariant predicates
        Def(P) is the set of definition blocks of operands
        in each maybe invariant predicate P
Output: Modified InvarList

For i in MaybeInvarList
   For B in Def(i)
      If (_marked(B))
         _remove_def_block(B, Def(i))
      EndIf
   EndFor
   If (Def(i) is NULL)
      add_to_invarlist(i)
   EndIf
EndFor
```

Figure 65. Process maybe invariant predicate list.

```
function: ProcBBstack()
Input: CFG, S(Stack)
        CFG is the control flow graph of a given function
        S is the stack of blocks to be processed for removal
Output: Basic locks marked as dead

While ((B = pop(S)) != NULL )
   If ( num_pred(B) = 0)
      _mark(B)
      For Bs in succ(B)
         If(!_marked(Bs))
            _push(Bs, S)
         EndIf
      EndFor
   EndIf
EndWhile
```

Figure 66. Process stack of basic blocks marked as dead.

6.3 Illustration

We now illustrate our dynamic invariant predicate identification algorithm with the help of an example. Figure 67 gives a motivation example that is a typical control flow graph of a loop with branch conditions in it. The static analysis on this example determines that the predicate a > b is invariant. The branch condition *p < 5 is classified as a maybe invariant. Statically, the set Def(*p < 5) includes the definitions of *p in the basic blocks B5 and B12. The other predicates d < e and f > lim are not classified under either of the predicate lists. This is so because their operand definitions are loop generated on all branches (B6, B7, B8, and B9).

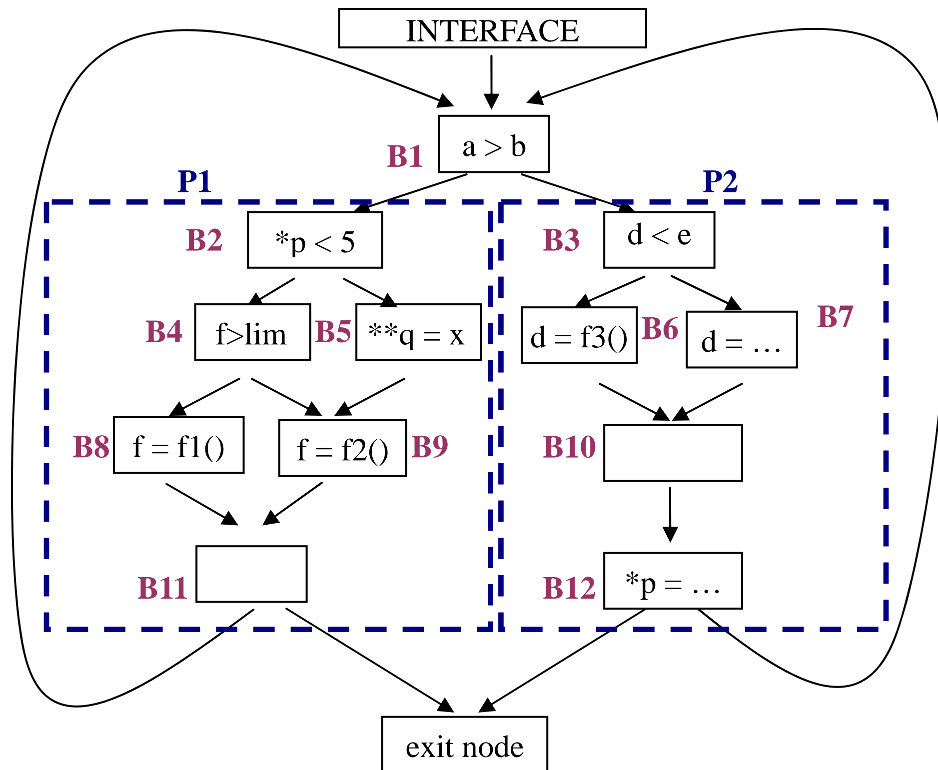


Figure 67. Example showing predicates inside a loop.

At the end of the static analysis, an Interface Function is constructed at the pre-header of the loop. This function performs the dynamic analysis using invariant and maybe invariant lists. The invariant predicate a > b is evaluated by the Interface Function based on the runtime value of a and b. If a > b evaluates to the path P1 as shown in Figure

68, then the path P2 becomes dead. The algorithm achieves this by first removing the edge <B1, B3>. Since the number of predecessors of B3 is zero, B3 can now be marked dead. Continuing with B3, the edges <B3, B6> and <B3, B7> are removed. Now Both B6 and B7 have zero predecessors, and hence they are marked dead. The algorithm proceeds by marking blocks B10 and B12 as dead except for the exit node as it still has a predecessor other than B12. After all of the invariant predicates are processed, the algorithm proceeds to handle the maybe invariant predicate *p < 5. Statically, the pointer q could chase p. However, assume in the current context, p is not chasing p, so the definition in B5 will not modify the value of *p, and the set Def(*p < 5) has only one definition block B12. B12 is dead, so it is removed from the Def(*p < 5) set leaving it empty. As a result, the predicate *p < 5 becomes invariant, and it is added to the invariant predicate list.

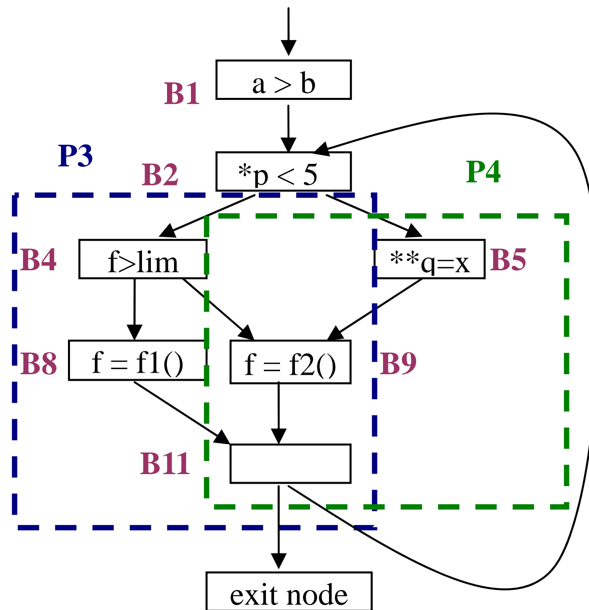


Figure 68. Control flow graph when the path P1 is taken.

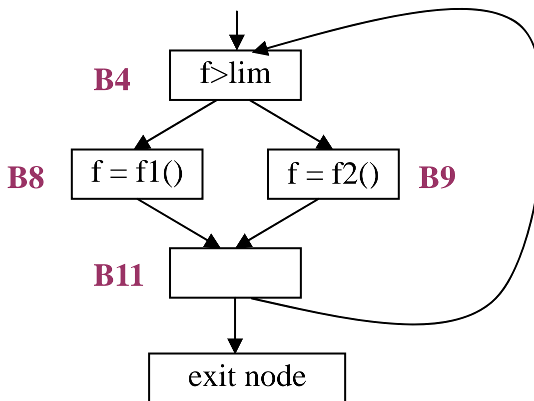


Figure 69. Control flow graph when the paths P1 and P3 are taken.

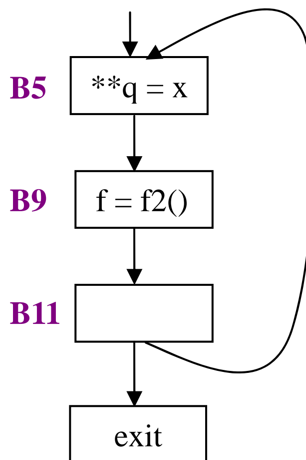


Figure 70. Control flow graph when the paths P1 and P4 are taken.

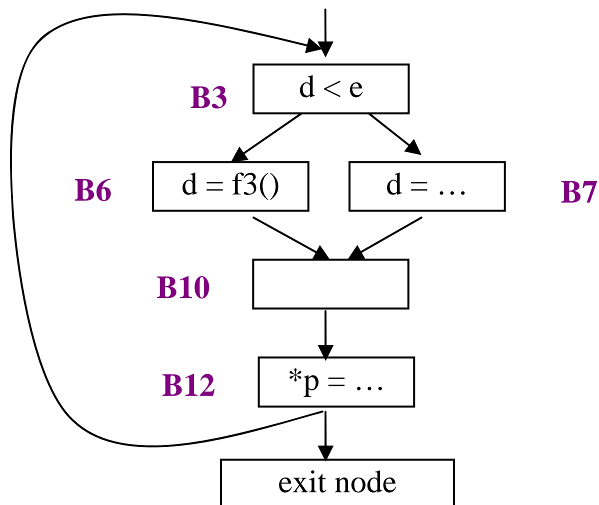


Figure 71. Control flow graph when the path P2 is taken.

It proceeds similarly for *p < 5 by first evaluating it. If the evaluation leads to the path P3, then the block B5 will be marked dead. The resulting control flow graph in this case is shown in Figure 69. Alternately, if *p < 5 evaluates to the path P4, then the blocks B4 and B8 would be marked dead. The final control flow graph would appear as shown in Figure 70. If the predicate a > b gets evaluated to the path P2, then the blocks under the path P1 become dead applying the above algorithm. Hence, the final resulting control flow graph is given by Figure 71.

6.4 Summary

In this chapter, we propose a dynamic invariant predicate identification method. Dynamic invariant predicate identification could detect more invariants than the static analysis. It is useful in program understanding and other applications. For example, it could help eliminate more dead code during program migration in mobile computing. It could assist runtime program properties verification, which often relies on an invariant assert statement in formal specifications. In this work, we combine the static analysis and the runtime context to determine the dead branches and basic blocks. The algorithm starts with the invariant predictions identified statically, and then proceeds to propagate the deadness information to other basic blocks until it gets a transitive closure of dead blocks. The maybe invariant predicate becomes invariant if all of the predicate definitions are in the dead basic blocks. With dynamic pointer tracking, the memory ambiguity is clarified, and thus explores more opportunities for dynamic invariant predicate identification.

CHAPTER 7

CONCLUSION

In this thesis, we propose and evaluate a dynamic pointer tracking framework to assist and enable program analysis, understanding, and optimization. This thesis represents two example applications as the dynamic pointer tracking consumers with different design and implementation goals:

Memory protection: a security-oriented approach that prevents erroneous memory stores from corrupting the sensitive program data. This approach can be integrated with enhanced secure architecture design for lowering the overheads.

Garbage collection: an approach that targets weakly typed programming languages such as C. The innovation of this approach is that our work enables a hybrid garbage collector. This garbage collector has both of the advantages of the reference counting style and mark sweep style garbage collectors. The dynamic unreachable code detection enables our garbage collector with more useful information, and thus improves the collection strength.

Dynamic invariant predicate identification: a method that dynamically identifies invariant predicate is proposed in this thesis. The static invariant identification is not good enough to catch all invariants at runtime. A framework based on the current context is necessary to aggressively find all possible invariant predicate. The branch conditions are evaluated, and the unreachable branches and basic blocks are marked as dead in the control flow graph. The new dead basic blocks could remove some definitions of the variables that are used by other branch condition. More variables become invariant, and more branch conditions could be predicted. The contribution of the proposed algorithm could detect both of the runtime dead basic blocks and functions.

Our work is still preliminary. The dynamic pointer tracking is practical to be customized and further optimized for other research topic. The effectiveness our approach at monitoring the program pointers behavior suggests a direction for future research work on understanding other program properties with dynamic tracking.

REFERENCES

[1] Zhang, K., Zhang T., and Pande, S. "Memory protection through dynamic access control", in the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Dec 2006.

[2] Aho, A.V., Sethi, R., and Ullman, J.D. "Compilers, Principles, Techniques, and Tools". Addison Wesley. 1986.

[3] Burger, D. and Austin, T. M. "The SimpleScalar Tool Set Version 2.0".

[4] Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C., and Mergen, M. "Implementing Jalapeno in Java", in OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications. Volume 34(10) of ACM SIGPLAN Notices., Denver, CO, ACM Press, 1999.

[5] Annavaram, M., Patel, J., and Davidson, E. "Data prefetching by dependence graph precomputation", in 28th Annual International Symposium on Computer Architecture, pages 52--61, July 2001.

[6] Austin, T. M., Breach, S. E., and Sohi, G. S. "Efficient Detection of All Pointer and Array Access Errors", in Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, 1994.

[7] Azatchi, H., Levanoni, Y., Paz, H., Petrank, E. "An on-the-fly mark and sweep garbage collector based on sliding view", in OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'03), ACM SIGPLAN Notices, Anaheim, CA, ACM Press (2003).

[8] Baker, H. G.. "Minimizing reference count updating with deferred and anchored pointers for functional data structures", ACM SIGPLAN Not. 29, 9.

[9] Beyer, D., Henzinger T. A., Jhala, R., and Majumdar, R. “Checking memory safety with Blast”, in Proceedings, 8th International Conference on Fundamental Approaches to Software Engineering (FASE), 2005.

[10] Blackburn, S.M., McKinley, K.S. “Ulterior reference counting: Fast garbage collection without a long wait”, in OOPSLA’03 ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA’03), ACM SIGPLAN Notices, Anaheim, CA, ACM Press (2003).

[11] Burke, M., Carini, P., Choi, J., and Hind, M. “Flow-insensitive interprocedural alias analysis in the presence of pointers”, in Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, 1994.

[12] Callahan, D. and Kennedy, K. “Analysis of Interprocedural Side Effects in a Parallel Programming Environment”. Journal of Parallel and Distributed Computing, 5:517–550, 1988.

[13] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R., K. "Non-Control-Data Attacks Are Realistic Threats", in Proc. USENIX Security Symposium, 2005.

[14] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. “Non-control-data attacks via pointer taintedness detection”, in IEEE International Conference on Dependable Systems and Networks (DSN), 2005.

[15] Castro, M., Costa, M., and Harris, T. “Securing Software by Enforcing Data-flow Integrity”, 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006.

[16] Cher, C.Y., Hosking, A.L., Vijaykumar, T. “Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign”, in Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA (2004), 199–210.

[17] Choi, J., Burke, M., and Carini, P. "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects". In 20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1993.

[18] Collins, G. E. "A method for overlapping and erasure of lists.", Commun. ACM 3, 12 (Dec.), 655–657, 1960.

[19] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. "Vigilante: End-to-End Containment of Internet Worms", SOSP'05, Brighton, UK, October 2005.

[20] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities", in Proc. of the 12th USENIX Security Symp., pages 91--104, 2003.

[21] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". 7th USENIX Security Conf., pages 63-78.

[22] Crandall, J. and Chong, F. "Minos: Control data attack prevention orthogonal to memory model", in MICRO-37, . 2004.

[23] Creusillet, B. and Irigoin, F. "Exact vs. Approximate Array Region Analyses". In Lecture Notes in Computer Science. Springer Verlag, NY, August 1996.

[24] Das, M. "Unification-Based Pointer Analysis with Directional Assignments", in Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, pages 35-46, June 2000.

[25] Deutsch, L.P., Bobrow, D.G.. "A efficient incremental automatic garbage collector. Communications", Communications of the ACM(9) (1976) 522–526.

[26] Devietti, J., Blundell, C., Martin M. M. K., and Zdancewic, S. "HardBound: Architectural Support for Spatial Safety of the C Programming Language", in

Architectural Support for Programming Languages and Operating Systems (ASPLOS'08), 2008.

[27] Dhurjati, D. and Adve, V. "Backwards-Compatible Array Bounds Checking for C with Very Low Overhead", in proceeding of the 28th International Conference on Software Engineering, 2006.

[28] Drinic M. and Kirovski, D. "A Hardware-Software Platform for Intrusion Prevention", in proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.

[29] Emami, M., Ghiya, R., and Hendren, L. J. "Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers", in SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 242-256, June 1994.

[30] Evans, D. and Larochelle, D. "Improving Security Using Extensible Lightweight Static Analysis", in IEEE Software, February 2002.

[31] Fahndrich, M., Rehof, J., and Das, M. "Scalable context-sensitive flow analysis using instantiation constraints", In Proc.PLDI, 2000.

[32] Feng, H. H., Kolesnikov O. M., Fogla P., Lee, W., Gong, W. "Anomaly Detection Using Call Stack Information", in Proceedings of IEEE Symposium on Security and Privacy, 2003.

[33] Forrest, S., Hofmeyr, S., A., Somayaji, A., and Longstaff, T., A. "A Sense of Self for Unix Processes", In Proceedings of the 1996 IEEE Symposium on Security and Privacy, 1996.

[34] Gao, D., Reiter, M. K.,and Song, D. "Gray-Box Extraction of Execution Graphs for Anomaly Detection", the 11th ACM CCS conf., pages 318-329, October 2004.

[35] Gao, D., Reiter, M. K., and Song, D. "On Gray-Box Program Tracking for Anomaly Detection", 13th USENIX Security Symposium, 2004.

[36] Hastings, R. and Joyce, B. “Purify: Fast Detection of Memory Leaks and Access Errors”, in proceedings of the Winter Usenix Conference, 1992.

[37] Hind, M., Burke, M., Carini, P., and Choi, J. “Interprocedural pointer alias analysis”, ACM Transactions on Programming Languages and System, 1999.

[38] Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. “Cyclone: A Safe Dialect of C”, USENIX Annual Technical Conference, 2002.

[39] Jones, R.E. “Garbage Collection: Algorithms for Automatic Dynamic Memory Management”, Wiley, Chichester with a chapter on Distributed Garbage Collection by R. Lins, 1996.

[40] Jones, W. M. R., and Kelly, H. J. P. “Backwards-Compatible Bounds Checking for Arrays and Pointers in C Program”, Automated and Algorithmic Debugging, 1997.

[41] Kirovski, D., Drinic, M., and Potkonjak, M. “Enabling Trusted Software Integrity”, in Proceedings of ASPLOS, San Jose, CA, 2002, pp. 108-120.

[42] Kiriansky, V., Bruening, D., and Amarasinghe, S. “Secure Execution Via Program Shepherding”, 11th USENIX Security Symposium, 2002.

[43] Kosoresow, A., Hofmeyr, S. “Intrusion Detection via System Call Traces”, IEEE Software, 1997.

[44] Krügel, C., Mutz, D., Valeur, F., and Vigna, G.. “On the Detection of Anomalous System Call Arguments”, in Proceedings of ESORICS 2003, 2003.

[45] Larochelle, D. and Evans, D. “Statically Detecting Likely Buffer Overflow Vulnerabilities”, in 2001 USENIX Security Symposium, 2001.

[46] Levanoni, Y., Petrank, E. “An on-the-fly reference counting garbage collector for Java”, in OOPSLA’01 ACM Conference on Object-Oriented Systems, Languages and Applications, Volume 36(10) of ACM SIGPLAN Notices., Tampa, FL, ACM Press, 2001.

[47] Michael, C., Ghosh, A. “Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report”, RAID, 2000.

[48] Mock, M., Das, M., Chambers, C., and Eggers, S. J. "Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization", in ACM SIGPLAN – SIGSOFT workshop on program analysis for software tools and engineering, 2001.

[49] Mock, M., Atkinson, D. C., Chambers, C., and Eggers, S. J. "Program Slicing with Dynamic Points-to Sets", in IEEE Transactions on Software Engineering, 2005.

[50] Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C", in Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, 2009.

[51] Necula, G., McPeak, S., and Weimer, W. "CCured: Type-safe retrofitting of legacy code", ACM Symposium on Principles of Programming Languages (POPL), 2002.

[52] Newsome, J. and Song, D. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software", in Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05), 2005.

[53] Paek, Y., Hoeflinger, J., and Padua, D. "Efficient and Precise Array Access Analysis", ACM Transactions on Programming Languages and Systems (TOPLAS). Vol. 24, Issue 1, pp. 65-109, January 2002.

[54] Paz, H. and Petrank, E. "Using Prefetching to Improve Reference-Counting Garbage Collectors", in Proceedings of the 16th International Conference on Compiler Construction (CC'07), March, 2007.

[55] Paz, H., Petrank, E., Bacon, D.F., Rajan, V., Kolodner, E.K. "An efficient on-the-fly cycle collection", in Proceedings of the 14th International Conference on Compiler Construction, Edinburgh, Springer-Verlag (2005).

[56] Paz, H., Petrank, E., Blackburn, S.M. "Age-oriented garbage collection", in Proceedings of the 14th International Conference on Compiler Construction, Edinburgh, Springer-Verlag (2005).

[57] Pearce, D. J., Kelly, P. H. J., and Hankin, C. "Efficient Field-sensitive pointer analysis for C", in Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2004.

[58] Pugh, W. "A Practical Algorithm for Exact Array Dependence Analysis", Communications of the ACM, 35(8), August 1992.

[59] Rabbah, R. M., Sandanagobalane, H., Ekpanyapong, M., and Wong, W. "Compiler Orchestrated Prefetching via Speculation and Predication", in Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04). Pages 189 – 198, 2004.

[60] Rugina, R., and Rinard M. "Pointer analysis for multithreaded programs", in Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, pp. 77-90, 1999.

[61] Ruwase, O. and Lam, M. S. "A Practical Dynamic Buffer Overflow Detector", in Proceedings of the 11th Annual Network and Distributed System Security Symposium, pages 159-169, February 2004

[62] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P., "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors", in Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.

[63] Shankar, U., Talwar, K., Foster, J., and Wagner, D. "Detecting Format String Vulnerabilities with Type Qualifiers", 10th USENIX Security Symposium, 2001.

[64] Shapiro, M., and Horwitz, S. "Fast and Accurate Flow-Insensitive Points-To Analysis", in Conference Record of POPL '97: Symposium on Principles of Programming Languages, January, 1997.

[65] Shi, W., Fryman, J. B., Gu G., Lee, S., Zhang, Y., Yang, J. "InfoShield: A Security Architecture for Protecting Information Usage in Memory", in 12th IEEE International Symposium on High Performance Computer Architecture, 2006.

[66] Smith, N. P. "Stack smashing vulnerabilities", in the UNIX operating system. Technical report, Computer Science Department, Southern Connecticut State University, 1997.

[67] Suh, G., Lee, J., and Devadas, S. "Security Program Execution via Dynamic Information Flow Tracking", 11th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2004.

[68] Steensgaard, B. "Points-to Analysis in Almost Linear Time", in 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1996.

[69] Triolet, R., Feautrier, P., and Irigoin, F. "Direct Parallelism of Call Statements", ACM SIGPLAN Symposium on Compiler Construction, pages 176–185, 1986.

[70] Tuck, N., Calder, B., and Varghese, G.. "Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow", in proceedings of the 37th International Symposium on Microarchitecture, December 2004.

[71] Venkataramani, G., Roemer, B., Solihin, and Y., Milos, P. "Memtracker: Efficient and programmable support for memory access monitoring and debugging", in High Performance Computer Architecture, 2007. HPCA 2007.

[72] Wagner, D., and Dean, D. "Intrusion Detection via Static Analysis", in Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.

[73] Wei J., Payne, B. D., Giffin, J., and Pu, C. "Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense", Computer Security Applications Conference, 2008.

[74] Whaley, J. and Lam, M. “Cloning-based context-sensitive pointer alias analyses using binary decision diagrams”, in Programming Language Design and Implementation, PLDI’04. 2004.

[75] Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D. “Dynamic storage allocation: A survey and critical review”. In Baker, H., ed.: Proceedings of International Workshop on Memory Management. Volume 986 of Lecture Notes in Computer Science., Kinross, Scotland, Springer-Verlag , 1995.

[76] Witchel, E., Cates, J., and Asanovic, K. “Mondrian Memory Protection”, Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X) , San Jose, CA, October 2002.

[77] Zhang, T., Zhuang X., Pande, S. and Lee, W. "Anomalous Path Detection with Hardware Support", International Conference on Compilers, Architectures and Synthesis for Embedded Processors (CASES), San Francisco, CA, Sep. 2005.

[78] Zhang, X., Gupta, R., and Zhang, Y. “Precise dynamic slicing algorithms”, ICSE '03: Proceedings of the 25th International Conference on Software Engineering. 2003.

[79] Zhou, P., Liu, W., Long, F., Lu, S., Qin, F., Zhou, Y., Midkiff, S., and Torrellas, J. “AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants”, in Proceedings of 37th Annual IEEE/ACM International Symposium on Micro-architecture (Micro'04), December, 2004.

[80] CERT Coordination Center. www.cert.org.

[81] Mach-Suif Backend Compiler, The Machine-Suif 2.1 compiler documentation set. Harvard University, Sep. 200. http://ececs.harvard.edu/hube/research/machsuif.html.

[82] “MIT Kerberos 5 Multiple Double-Free Vulnerabilities”. http://www.securityfocus.com/bid/11078

[83] “Multiple Vendor C Library realpath() Off-By-One Buffer Overflow Vulnerability”. http://www.securityfocus.com/bid/8315/discuss.

[84] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

[85] "An Artificial Garbage Collection Benchmark". http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html

[86] www.trimaran.org/benchmarks.shtml

[87] http://www.cs.umd.edu/~jfoster/cqual/

[88] http://www.cs.princeton.edu/~mcc/olden.html