Abstract

Title of dissertation: Runtime Enforcement of Memory Safety for the C Programming Language

Matthew Stephen Simpson, Doctor of Philosphy, 2011

Dissertation directed by: Professor Rajeev Barua

Memory access violations are a leading source of unreliability in C programs. Although the low-level features of the C programming language, like unchecked pointer arithmetic and explicit memory management, make it a desirable language for many programming tasks, their use often results in hard-to-detect memory errors. As evidence of this problem, a variety of methods exist for retrofitting C with software checks to detect memory errors at runtime. However, these techniques generally suffer from one or more practical drawbacks that have thus far limited their adoption. These weaknesses include the inability to detect all spatial and temporal violations, the use of incompatible metadata, the need for manual code modifications, and the tremendous runtime cost of providing complete safety.

This dissertation introduces MemSafe, a compiler analysis and transformation for ensuring the memory safety of C programs at runtime while avoiding the above drawbacks. MemSafe makes several novel contributions that improve upon previous work and lower the runtime cost of achieving memory safety. These include (1) a method for modeling temporal errors as spatial errors, (2) a hybrid metadata representation that combines the most salient features of both object- and pointer-based approaches,

and (3) a data-flow representation that simplifies optimizations for removing unneeded checks and unused metadata.

Experimental results indicate that MemSafe is capable of detecting memory safety violations in real-world programs with lower runtime overhead than previous methods. Results show that MemSafe detects all known memory errors in multiple versions of two large and widely-used open source applications as well as six programs from a benchmark suite specifically designed for the evaluation of error detection tools. MemSafe enforces complete safety with an average overhead of 88% on 30 widely-used performance evaluation benchmarks. In comparison with previous work, MemSafe's average runtime overhead for one common benchmark suite (29%) is a fraction of that associated with the previous technique (133%) that, until now, had the lowest overhead among all existing complete and automatic methods that are capable of detecting both spatial and temporal violations.

Runtime Enforcement of Memory Safety for the C Programming Language

by

Matthew Stephen Simpson

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

Advisory Committee:

Professor Rajeev Barua, Chair
Professor Shuvra Bhattacharyya
Professor Michael Hicks
Professor Chau-Wen Tseng
Professor Donald Yeung

Acknowledgements

There are several people I would like to thank who were instrumental in the completion of my doctoral dissertation and my graduate studies in general. Foremost among these are my parents, Steve and Judy Simpson. I thank them for their constant love, support, and patience while completing my graduate studies. They instilled in me, at an early age, the hard work and disciple required for pursing a doctoral degree, and my achievement is a reflection of their own success as parents. I will never be able to thank them enough for all that they have done for me.

I would like to thank my advisor, Dr. Rajeev Barua. Dr. Barua became my advisor in the fall semester of 2004, but we had previously worked together the year before when I participated in a summer-long research program for undergraduate students. My experience at the University of Maryland then was enormously influential in the overall trajectory of my academic endeavors, and I am grateful for having been given the opportunity to return as a graduate student. His advice throughout these years has made me a much better writer and researcher and also given me insight into the challenges and rewards associated with a career in academia. Many of the ideas presented in this dissertation are the direct result of our conversations and discussions, and it is fair to say that without his eager support and extensive knowledge of compilers and computer systems, this dissertation would not have been possible.

I would also like to thank Dr. Dina Demner-Fushman, Dr. Sameer Antani, and Dr. George Thoma—colleagues and mentors of mine at the U.S. National Library of Medicine—for providing a welcome distraction from the stress of my dissertation, and introducing me to new and exciting areas of research.

Finally, I would like to thank several other people whose contribution to this dissertation may have been small, but whose broader impact was large. I thank Allison Barnett for her unwaivering support and companionship and for encouraging me during the most stressful times of my doctoral studies. I thank David Sander for helping me to survive in the Electrical and Computer Engineering department, and I thank members of the Systems & Computer Architecture Lab, especially Bhuvan Middha and Nghi Nguyan, for discussions related to this work. Lastly, I would like to acknowledge the friends I have in the Washington, D.C. area and my friends and family elsewhere. Thank you all!

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 5
- 1.3 Organization of Dissertation . . . 9

2 C Language Compilation and Analysis **13**
- 2.1 The C Programming Language . . . 14
  - 2.1.1 History . . . 14
  - 2.1.2 Common use . . . 16
  - 2.1.3 Low-level features . . . 17
- 2.2 Compilation . . . 21
- 2.3 Analysis . . . 23
  - 2.3.1 Control-flow and call graph construction . . . 24
  - 2.3.2 Data-flow analysis . . . 25
  - 2.3.3 SSA form . . . 26
  - 2.3.4 Alias analysis . . . 27

3 Memory Safety Violations and Prior Enforcement Methods **29**
- 3.1 Memory Safety Violations . . . 29
  - 3.1.1 Bounds violations . . . 31
  - 3.1.2 Uninitialized pointer dereference . . . 32
  - 3.1.3 Null pointer dereference . . . 32
  - 3.1.4 Manufactured pointer dereference . . . 33
  - 3.1.5 Dereference of dangling stack pointers . . . 34
  - 3.1.6 Dereference of dangling heap pointers . . . 34
  - 3.1.7 Multiple deallocations . . . 35
- 3.2 Prior Enforcement Methods . . . 35
  - 3.2.1 Spatial safety . . . 36
  - 3.2.2 Temporal safety . . . 41

4 MemSafe **45**
- 4.1 Language Extensions and Assumptions . . . 46
  - 4.1.1 Memory deallocation . . . 48
  - 4.1.2 Pointer stores . . . 52
- 4.2 The Required Checks and Metadata . . . 56

4.2.1 Pointer metadata . . . 58
4.2.2 Pointer bounds check . . . 59
4.2.3 Object metadata . . . 61
4.2.4 Object bounds check . . . 62
4.3 Propagation of the Required Metadata . . . 64
4.3.1 Memory allocation . . . 64
4.3.2 Memory deallocation . . . 66
4.3.3 Address-of operator . . . 68
4.3.4 Pointer copies and arithmetic . . . 69
4.3.5 ϱ-functions . . . 69
4.3.6 NULL and manufactured pointers . . . 71
4.3.7 Function arguments and return values . . . 72
4.4 Memory Safety for Multithreaded Programs . . . 75
4.4.1 Declaration of the required locks . . . 77
4.4.2 Object bounds check . . . 78
4.4.3 Memory allocation . . . 79
4.4.4 Memory deallocation . . . 82
4.4.5 ϱ-functions . . . 84
4.4.6 Function calls . . . 86
4.5 Example Application . . . 88

5 Reducing the Runtime Cost of Enforcing Memory Safety 95
5.1 A Data-flow Graph for Pointers . . . 95
5.1.1 Construction . . . 97
5.1.2 Connectivity . . . 99
5.1.3 Properties . . . 100
5.1.4 Example application . . . 104
5.2 Optimizations of the Basic Approach . . . 107
5.2.1 Dominated dereferences optimization . . . 108
5.2.2 Temporally safe dereferences optimization . . . 111
5.2.3 Non-incremental dereferences optimization . . . 113
5.2.4 Monotonically addressed ranges optimization . . . 115
5.2.5 Partitioned metadata optimization . . . 119
5.2.6 Unused metadata optimization . . . 122

6 MemSafe Implementation 127
6.1 MemSafe's Analysis and Transformation . . . 127
6.2 Metadata Facilities . . . 130
6.2.1 Implementation alternatives . . . 130
6.3 Metadata Allocation . . . 132
6.4 Limitations . . . 135
6.4.1 Separate compilation . . . 135
6.4.2 NULL and manufactured pointers . . . 136

7 Results **137**
7.1 Effectiveness in Detecting Errors . . . 138
7.2 Runtime Performance . . . 140
7.2.1 Increase in runtime . . . 141
7.2.2 Increase in memory consumption . . . 146
7.2.3 Effectiveness of optimizations . . . 147
7.2.4 Additional cost of temporal safety . . . 150
7.3 Static analysis . . . 151

8 Related Work **157**
8.1 Spatial and Temporal Safety . . . 157
8.2 Spatial Safety . . . 158
8.3 Temporal Safety . . . 159
8.4 Software Debugging Tools . . . 159
8.5 Other Methods of Memory Protection . . . 160
8.6 SSA Extensions . . . 161

9 Future Work **163**
9.1 Performance Enhancements and Evaluation . . . 163
9.2 Specification and Verification . . . 165
9.3 Additional Uses . . . 166

10 Conclusion **169**

A Metadata Propagation for the C Standard Library **173**
A.1 Memory Copying Functions of `string.h` . . . 173
A.2 Variadic Function Macros of `stdarg.h` . . . 175

B Spatial Safety and Segmentation **179**
B.1 Spatial Safety . . . 179
B.1.1 The required checks and metadata . . . 180
B.1.2 Propagation of the required metadata . . . 181
B.2 Segmentation . . . 189
B.2.1 Propagation of the required metadata . . . 191

List of Tables

1.1 Related work 8

3.1 Memory safety violations 30

7.1 Violations detected in Bugbench 138
7.2 Violations detected in real-world applications 139
7.3 Dynamic results with whole-program analysis 142
7.4 Dynamic results with separate compilation 143
7.5 Static results with whole-program analysis 152
7.6 Static results with separate compilation 153

List of Figures

3.1 Prior enforcement methods 38

4.1 Language syntax for MemSafe presention 46
4.2 Prior enforcement methods 57
4.3 Merge sort algorithm 89
4.4 Merge sort in SSA form 91
4.5 Merge sort fragment with metadata and checks 93

5.1 DFPG construction 97
5.2 Merge sort DFPG 105

6.1 Header allocation 134

7.1 Runtime comparison with MSCC 144
7.2 Optimization effectiveness with whole-program analysis 148
7.3 Optimization effectiveness without whole-program analysis 149
7.4 Effect of aliasing 151
7.5 Compile-time slowdown 154

List of Runtime Checks

4.1 Pointer bounds check 59
4.2 Object bounds check 62
4.3 Object bounds check (thread safe) 79

5.1 Monotonically addressed range check 116
B.1 Pointer bounds check (spatial safety) 181

List of Metadata Rules

4.1 Automatic memory allocation 64
4.2 Dynamic memory allocation 65
4.3 Automatic memory deallocation 66
4.4 Dynamic memory deallocation 67
4.5 Address-of operator 68
4.6 Pointer copies and arithmetic 69
4.7 Pointer loads 70
4.8 Pointer stores 71
4.9 `NULL` and manufactured pointers 72
4.10 Function calls 73
4.11 Function declarations 74
4.12 Automatic memory allocation (thread safe) 80
4.13 Dynamic memory allocation (thread safe) 81
4.14 Automatic memory deallocation (thread safe) 82
4.15 Dynamic memory deallocation (thread safe) 83
4.16 Pointer loads (thread safe) 84
4.17 Pointer stores (thread safe) 85
4.18 Function calls (thread safe) 86
4.19 Function declarations (thread safe) 87

A.1 Memory copying functions 174
A.2 Variadic function arguments 176
A.3 Argument list bounds 177
A.4 Argument list pointer 178
B.1 Dynamic memory allocation (spatial safety) 182
B.2 Address-of operator (spatial safety) 183
B.3 Pointer copies and arithmetic (spatial safety) 184
B.4 Pointer loads (spatial safety) 185
B.5 Pointer stores (spatial safety) 185
B.6 `NULL` and manufactured pointers (spatial safety) 186
B.7 Function calls (spatial safety) 187
B.8 Function declarations (spatial safety) 188
B.9 Dynamic memory allocation (segmentation) 191
B.10 Address-of operator (segmentation) 192

Chapter 1

Introduction

This dissertation shows that an automatic compiler analysis and transformation technique is capable of ensuring the memory safety of C programs at runtime. A program is transformed such that it detects spatial and temporal memory errors before they occur, while remaining compatible with existing code and requiring lower runtime overhead than similar techniques. The motivation and contributions of this research are outlined below.

1.1 Motivation

Use of the C programing language remains common despite the well-known memory errors it allows. The features that make C a desirable language for many system-level programing tasks—namely its weak typing, low-level access to computer memory, and pointers—are the same features whose misuse cause the variety of difficult-to-detect memory access violations common among C programs. Although these violations often cause a program to crash immediately, their symptoms can frequently go undetected long after they occur, resulting in data corruption and incorrect results while making

software testing and debugging a particularly onerous task.

A commonly cited memory error is the buffer overflow, where data is stored to a memory location outside the bounds of the buffer allocated to hold it. Although buffer overflow errors have been understood as early as 1972 [5, pg. 61], they and other memory access violations still plague modern software and are a major source of recently reported security vulnerabilities. For example, according to the United States Computer Emergency Readiness Team (US-CERT), 67 (29%) of the 228 vulnerability notes released in 2008–2009 were due to buffer overflow errors alone [80].

Several safety methods [e.g. 9, 62, 68, 82] have characterized memory access violations as either *spatial* or *temporal* errors. A spatial error is a violation caused by dereferencing a pointer that refers to an address outside the bounds of its "referent." Examples include indexing beyond the bounds of an array; dereferencing pointers obtained from invalid pointer arithmetic; and dereferencing uninitialized, `NULL` or "manufactured" pointers.[1] A temporal error is a violation caused by using a pointer whose referent has been deallocated (e.g. by calling the `free` standard library function) and is no longer a valid memory object. The most well-known temporal violations include dereferencing "dangling" pointers to dynamically allocated memory and attempting to deallocate a pointer more than once. However, dereferencing pointers to automatically allocated memory (i.e., stack variables) is also a concern if the address of the referent "escapes" and is made available outside the function in which it was

[1] A *manufactured* pointer is a pointer created by means other than explicit memory allocation (e.g., by calling the `malloc` standard library function) or taking the address of a variable using the address-of operator (`&`). Type-casting an integral type to a pointer type is a common example. The various memory safety violations are discussed in detail in Chapter 3.

defined. A program is *memory safe* if it does not commit any spatial or temporal errors.

Safe languages, such as Java, ensure memory safety through a combination of syntax restrictions and runtime checks, and are widely-used when security is a major concern. Others, like Cyclone [48] and Deputy [21], preserve many of the low-level features of C, but require additional programmer annotations to assist in ensuring safety. Although the use of these languages may be ideal for safety-critical environments, the reality is that many of today's applications—including operating systems, web browsers, and database management systems—are still typically implemented in C or C++ because of its efficiency, predictability, and access to low-level features. This trend will likely continue into the future.

As an alternative to safe languages, sophisticated static analysis methods for C [e.g. 11, 13, 27, 32, 34] can be used alone, or in conjunction with other systems, to ensure the partial absence of spatial and temporal errors statically. While these techniques are invaluable for software verification and debugging, they can rarely prove the absence of all memory errors and often require a significant amount of verification time due to the precision of their analyses.

A growing number of methods rely primarily on inserted runtime checks to detect memory access violations dynamically. However, the methods capable of detecting both spatial and temporal memory safety violations [9, 24, 30, 33, 43, 49, 64, 66, 68, 74, 82, 83] generally suffer from one or more practical drawbacks that have thus far limited their widespread adoption. These drawbacks can be summarized by the following qualities.

- **Completeness.** Methods that associate *metadata* (the base and bound information required for runtime checks) with objects [e.g. 24, 30, 33, 43, 49, 74, 83]—rather than the pointers to these objects—generally do not detect two kinds of memory errors. First, because C supports the allocation of nested objects (e.g., an array of structures), spatial errors involving sub-object overflows are not detected since inner objects share metadata with the outer object. Second, if the system allocates an object to a previously deallocated location, temporal errors are not detected since dangling pointers to the deallocated object may still refer to a location within bounds of the newly allocated object.

- **Compatibility.** The use of alternate pointer representations, such as multi-word "fat-pointers" [e.g. 9, 64] to store metadata raises compatibility concerns. Inline metadata breaks many legacy programs—and requires implicit language restrictions for new ones—because it changes the memory layout of pointers. For example, since their data types are the same size, programmers often cast pointers to integers to compute certain addresses. However, since fat-pointers alter memory layout, this computation is no longer valid and can result in data corruption. Inline metadata also breaks the calling convention of external libraries whose parameters or return types involve pointers.

- **Code Modifications.** Some methods [e.g. 64] require non-trivial source code modifications to avoid the above compatibility issues or to prevent an explosion in runtime. A common example is for a programmer to write "wrapper functions" that remove inline metadata in order to interface with external libraries.

- **Cost.** Methods capable of detecting both spatial and temporal errors often suffer from high performance overheads [e.g. 9, 66, 68, 82]. This is commonly due to the cost of maintaining the metadata required for ensuring spatial safety and the use of conservative garbage collection for ensuring temporal safety. High runtime overhead can make a method prohibitively expensive for deployment and can slow the development process when it is used for testing, especially if a program is to be executed many times to increase coverage.

1.2 Contributions

This dissertation introduces MemSafe [77], a method for ensuring both the *spatial* and *temporal* memory safety of C programs at runtime. MemSafe is a whole-program compiler analysis and transformation that, like other runtime methods, utilizes a limited amount of static analysis to prove memory safety whenever possible, and then inserts checks to ensure the safety of the remaining memory accesses at runtime. MemSafe is *complete*, *compatible*, requires no *code modifications*, and generally has lower runtime *cost* than other complete and automatic methods achieving the same level of safety. MemSafe makes the following contributions for lowering the runtime cost of dynamically ensuring memory safety:

- MemSafe uniformly handles all memory violations by modeling temporal errors as spatial errors. Therefore, the use of separate mechanisms for detecting temporal errors (e.g. garbage collection or explicit checks for temporal safety [9, 68, 82, 83]) is no longer required.

- MemSafe captures the most salient features of object and pointer metadata in a hybrid spatial metadata representation. MemSafe's handling of pointer metadata is similar to that of SoftBound [62], a previous technique for detecting spatial errors, and ensures MemSafe's completeness and compatibility. However, MemSafe's additional use of object metadata creates a novel synergy with pointer metadata that allows the detection of temporal errors as well.

- MemSafe uniformly handles pointer data-flow in a representation that simplifies several performance-enhancing optimizations. Unlike previous methods that require checks for all dereferences and the expensive propagation of metadata at every pointer assignment [e.g. 9, 66, 68, 82], MemSafe eliminates redundant checks and the propagation of unused metadata. This capability is further enhanced with whole-program analysis.

In order to achieve the above, MemSafe exploits several key insights related to the flow of pointer values in a program. The following program behavior models form the foundation of MemSafe's approach.

1. *Memory deallocation can be modeled as an assignment.* For example, the statement `free(p)` can be represented by the statement `p = invalid`, where *invalid* is a special untyped pointer to a temporally "invalid" range of memory.

   This insight is useful because it enables spatial safety mechanisms to be reused to ensure temporal safety. In order to detect spatial safety violations, existing methods insert before pointer dereferences runtime checks that determine whether the pointers refer to a location within the base and bound addresses of

their referents. If a dereferenced pointer refers to a location outside the region of memory occupied by its referent, a spatial safety violation is signaled. By assigning pointers to deallocated memory to be equal to the *invalid* pointer, they inherit the base and bound addresses of the "invalid" region of memory. If the base and bound addresses of this region are defined such that they represent some impossible address range (e.g., a block with a negative size), any legal pointer must refer to a location outside this range. Thus, dereferences of dangling pointers and multiple deallocation attempts can then be detected with the inserted checks for spatial safety.

2. *Indirect pointer assignments can be modeled as explicit assignments.* Statements of the form `ptr1 = *p`, where both ptr_1 and p are pointers, make low-cost memory safety difficult to achieve since ptr_1*'s* set of potential referents is not known statically. Alias analysis can be used to narrow this set, and MemSafe makes the results of this analysis explicit in the program's Static Single Assignment (SSA) [26] form by using a new ϕ-like construct called the ϱ-function.

   For example, assume the statement `s0:*p = ptr0` is the only direct reaching definition of a pointer defined as `s1:ptr1 = *p`. The statement `s2:*q = ptr2` may *indirectly* redefine ptr_1 if p and q may alias and control-flow may reach statement s_1 from s_2. Therefore, MemSafe models the statement `ptr1 = *p` as `ptr1 = ϱ(ptr0, ptr2)`, meaning the value of ptr_1 may equal that of ptr_0 or ptr_2 but only these two values.

   This insight is useful because it enables MemSafe to construct a convenient

Approach	Complete	Compatible	No Code Modifications	Whole Program	Slowdown
Purify [43]	no	yes	yes	yes	148.44*
Patil, Fischer [68]	yes	yes	yes	no	6.38†
Safe C [9]	yes	no	yes	no	4.88†
Fail-Safe C [66]	yes	yes	yes	no	4.64†
MSCC [82]	yes	yes	yes	no	2.33
Yong, Horwitz [83]	no	yes	yes	no	1.37‡
CCured [64]	yes	no	no	yes	1.30
MemSafe	**yes**	**yes**	**yes**	**yes**	**1.29**

Table 1.1: Related work. A comparison of methods providing both spatial and temporal memory safety is given. Slowdown is computed as the ratio of the execution time of the instrumented program to that of the original program. Slowdown is reported for the Olden benchmarks [71] unless otherwise noted.

*Checks are only inserted for heap objects.
†Slowdown is the average of all results reported by the authors.
‡Checks are only inserted for store operations.

data-flow graph that codifies both direct and indirect pointer assignments—in addition to memory deallocation with the insight above (1)—as simple definition and use relationships. Thus, this representation greatly simplifies optimizations for reducing the cost of achieving memory safety.

A prototype implementation of MemSafe has been evaluated in terms of its completeness and runtime cost. MemSafe was able to successfully detect known memory violations in multiple versions of the Apache HTTP server [6] and the GNU Core Utilities [39] software package. Additionally, MemSafe detected all previously reported memory errors in six programs from the BugBench [60] benchmark suite. In terms of cost, MemSafe's average overhead was 88% on 30 large programs widely-used in evaluating error detection tools. Finally, as evidence of its compatibility, MemSafe compiled each of the above programs without requiring any code modifications or programmer intervention.

Table 1.1 summarizes previous software approaches for ensuring both spatial and temporal safety.[2] Each method is evaluated on its completeness, compatibility, lack of code modifications, use of whole-program analysis, and runtime cost. For consistency, slowdown is reported for the Olden benchmarks [71] where results are available. MemSafe compares favorably in each category and has the lowest overhead among all existing complete and automatic methods. This result is primarily due to MemSafe's novel contributions based on the above insights.

Since MemSafe's performance overheads cannot necessarily be considered "low," MemSafe is deployable in systems whose primary concern is memory safety. In practice, it has been observed that many runtime checks can be avoided with MemSafe's simple optimizations, and for safety-critical applications, MemSafe's moderate runtime overheads can be an acceptable trade-off compared to redesigning systems in a safe language. However, for performance-critical applications, MemSafe is primarily useful as a dynamic bug detection tool.

1.3 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents an overview of the C programming language and compilation process. It briefly reviews the history of the language, introduces the low-level features of C whose misuse can result in memory safety violations, and describes the basic compiler analysis techniques

[2]Other methods (e.g, CIT [1], DFI [17], WIT [2], SoftBound [62], SafeCode [31], "baggy" bounds checking [3], etc.) are excluded from Table 1.1 since they either are (1) not software-only mechanisms for detecting memory errors or (2) do not aim to ensure complete spatial and temporal safety. However, these methods are discussed in detail in Chapter 8.

that MemSafe requires for enforcing memory safety.

Chapter 3 describes the memory safety violations that are commonly found in C programs and reviews prior work related to the detection and prevention of memory errors. In reviewing previous work, this chapter primarily focusses on the way in which prior methods organize the base and bound information required by runtime safety checks, and it discusses the strengths and weakness of each approach.

Chapter 4 describes MemSafe's basic, unoptimized approach for ensuring the memory safety of C programs at runtime. It presents the challenges associated with the use of memory deallocation and indirect pointer assignments, and it describes the C language syntax extensions that MemSafe uses to reason about these programming idioms. This chapter then defines the runtime checks, metadata, and metadata propagation rules required for MemSafe to enforce spatial and temporal memory safety. The chapter concludes with additions to these basic rules that allow MemSafe to ensure the memory safety of multithreaded programs.

Chapter 5 describes how MemSafe is able to reduce the runtime overhead of achieving memory safety. It builds upon the language extensions introduced in Chapter 4 to construct a novel data-flow representation for pointers, and it then describes how this representation can be utilized for identifying and eliminating unneeded runtime checks and code for propagating unused metadata.

Chapter 6 describes the prototype implementation of MemSafe and some of its limitations. Then, Chapter 7 evaluates the implementation of MemSafe based on its ability to detect memory safety violations in real-world programs and the runtime overhead required for it to do so. This chapter demonstrates that MemSafe's key

contributions—namely, the modeling temporal errors as spatial errors, a hybrid metadata representation, and MemSafe's data-flow representation—are effective tools for reducing the cost of dynamically ensuring memory safety.

Finally, Chapter 8 describes additional related work by reviewing methods capable of detecting both spatial and temporal violations as well as techniques that can only detect one type of memory error. This chapter also presents a discussion of previous work related to MemSafe's data-flow analysis. Chapter 10 concludes this dissertation by summarizing important aspects of the above.

Chapter 2

C Language Compilation and Analysis

The C programming language is one of the most popular languages of all time, and a C compiler is available for almost all computer architectures. However, despite its ubiquity, the features that make C desirable for many system-level programing tasks—namely its weak typing, low-level access to computer memory, and pointers—are the same features whose misuse cause the variety of difficult-to-detect memory access violations that are common among C programs.

This chapter presents an overview of the C programming language and the C language compilation process. Specifically, Section 2.1 gives a brief history of C, discusses its use in common systems and applications, and introduces the low-level features whose misuse can result in memory safety violations. Section 2.2 describes the basic steps involved in the C compilation process, and Section 2.3 introduces some of the program analysis techniques used by most modern-day optimizing compilers. These techniques include control- and data-flow analyses, the Static Single Assignment (SSA) form [26], and alias analysis.

2.1 The C Programming Language

The C programming language [46] is a general-purpose programming language that was initially developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system [70]. Although C was originally intended to be used for implementing systems software, many of today's commonly used applications—including database management systems and web browsers—are implemented in C. As such, the C programming language is one of the most widely-used programming languages, and this trend is likely to continue into the future. This section reviews the development of the C, discusses its most common uses, and presents the low-level features of the language that are often responsible for contributing to the commonly occurring violations of memory safety.

2.1.1 History

The origin of the C programming language is closely tied to the development of the Unix operating system [70]. The Unix kernel (the central component of most computer operating systems) was originally developed in assembly language for the PDP-7 computer. The PDP-7 was an early computer developed by the Digital Equipment Corporation (DEC). Since assembly code is non-portable and specific to a particular computer architecture, changes in the computer hardware on which an assembly program is designed to run require developers to rewrite the program to match the physical features of the new architecture. The C programming language was developed—based on the specification of a previous language named "B" (from

which C derives its name)—to be a high-level version of assembly language with which the Unix kernel could be portably rewritten. Due to the early success of C, Unix became one of the first operating system kernels to be implemented in a language other than assembly.

The first effort at standardizing the C programming language came in 1978 with the publication of *The C Programming Language* by Kernighan and Ritchie [50]. This book served as the *de facto* specification of the language before C became standardized. Several new features were added to the language at this time, including a standard library for I/O operations and the `long int` and `unsigned int` data types. In the years following the publication of the book, several unofficial features were added to the language in addition to these, which were supported to varying degrees by the exiting C compilers. These extensions included `void` functions and the ability for a function to return `struct` and `union` types.

Given the large number of extensions and the increasing popularity of C, standardization became necessary. The C Programming language was standardized by the American National Standards Institute (ANSI) in 1989 and by the International Organization for Standards (ISO) in 1990. This standard is commonly referred to as ANSI C, C89, or C90. The standardization process resulted in the inclusion of additional features, such as function prototypes (a declaration specifying a function's name, arity, and argument and return types) and `void` pointers. A program conforming to the ANSI C standard that does not make any assumptions of the hardware on which it will run (e.g., byte endianness) will run correctly, within resource constraints, on any system that having an implementation of C (i.e., the standard libraries) that

also conform to the ANSI standard.

At the time of this writing, the most recent update to the ANSI/ISO C standard came in 1999. The corresponding specification is commonly referred to as C99, and it introduced several new features, including inline functions, additional data types, and support for variable-length arrays. C99 is backward compatible with ANSI C, but the reverse is not true. That is, a program conforming to the previous C standard conforms to the current C99 specification. MemSafe assumes that the source code to which it is applied conforms to the C99 specification, and all code transformations that MemSafe makes conform to this standard as well.

2.1.2 Common use

C is most often used for "systems programming." Systems programming is distinct from application programming in that the latter aims to produce software that provides a particular service to an end user (e.g, a word processor) whereas the former aims to produce software that provides services to a computer system. As such, systems programming requires greater knowledge of computer hardware. Examples of systems programming tasks include implementing operating systems and embedded systems applications. The C programming language is well-suited for these tasks because of its efficiency, predictability, weak typing, and low-level access to computer memory. Ironically, it is the misuse of these features that results in the memory safety violations MemSafe aims to detect.

Given its wide acceptance and efficiency, many commonly-used applications and

language implementations are also written in C. For example, the C language is used for implementing large database management systems and web browsers as well as the compilers, libraries, and interpreters of other languages. Python, Perl and PHP are examples of such languages. The efficiency of C code also makes it particularly well-suited for implementing computationally intensive software, such as applications for analyzing large amounts of scientific data.

2.1.3 Low-level features

As mentioned above, several of the features that make C a desirable programming language are the same features that are often responsible for commonly occurring memory access violations. Memory errors are made possible in C by the ability to acquire low-level access to computer memory and to manipulate the data that is stored in a particular region of memory. The discussion below describes the C language features that can lead to violations of memory safety—namely the unrestricted use of pointers and manual memory management.

2.1.3.1 Pointers

A pointer is a value that enables a program to indirectly access data that is stored in a computer's main memory or that is located in some peripheral device (e.g., through memory-mapped I/O). Pointers record the memory address of an object or function and are said to "point" or "refer" to the data located at that address. Pointers are *dereferenced* in order to access the data to which they point, or as is the case with pointers to a function, they may be dereferenced to invoke a procedure.

Pointers are useful for a variety purposes in C including manual memory management (discussed below) and the implementation of common data structures, such as trees and lists. Additionally, function pointers are frequently used to implement the callback mechanism required by event handlers. However, because a pointer variable can be made to refer to an arbitrary location, and because pointer operations are typically unchecked in C, the misuse of pointers are responsible for a variety of memory access violations, particularly those related to spatial safety.

Pointers in C are created using the address-of operator (`&`) or by calling the `malloc` standard library function. However, the value of one pointer may be assigned to another pointer, and a pointer may be assigned an arbitrary value through the use of type-casting. Pointers are manipulated through simple assignments and arithmetic. A pointer that is assigned the value of `NULL` refers to no object, and in most systems, a dereference of the `NULL` pointer results in a runtime error. `NULL` pointers are useful in C programming for indicating special cases. For example, a `NULL` pointer can be used to indicate that there are no items beyond the last element of a linked list.

A pointer's type indicates the type of data stored at the location to which it refers. However, a `void` pointer points to an object of unknown or unspecified type, and can therefore be used as a generic data pointer or to implement type polymorphism. Pointers that are `void` cannot be dereferenced, and pointer arithmetic on them is not allowed. However, a pointer of one type can be freely converted into a pointer of another type through type-casting.

A unique feature of the C programming language is the duality that exists between pointers and arrays. Essentially, a variable declared as an array of a particular type

also acts as a pointer to that type, and when the variable is used by itself (i.e., the array is not indexed), it is a pointer to the first element of the array. Formally, the array sub-script notation `a[i]` is equivalent to `*(a + i)`, where pointer arithmetic is performed on a pointer to the first element of the array to compute the address of the i^{th} element. The dereference operation accesses the data stored at the resulting address. Thus, pointer arithmetic and array indexing are identical operations. For simplicity, MemSafe represents all array indexing operations as functionally equivalent pointer arithmetic.

2.1.3.2 Manual memory management

The C programming language provides three mechanisms by which a programmer can manually manage objects stored in memory. These include static memory allocation for managing global objects, automatic memory allocation for managing stack objects, and dynamic memory allocation for managing heap objects. The misuse of memory allocation and deallocation mechanisms, especially those for managing heap-allocated objects, are responsible for a variety of memory access violations, particularly those related to temporal safety.

Static memory allocation refers to the process by which global variables are allocated. Since the number and size of global variables is known statically, storage for global variables is provided in a program binary by the compiler during compilation. Static variables have a global scope, a lifespan equal to that of the program, and are not deallocated until the program terminates and the binary in which they are contained is removed from memory. Static allocation incurs no runtime overhead

since the storage space is managed by the compiler. Although this lack of overhead is desirable, static allocation is not suitable for many programming tasks, such as the implementation of data structures that can potentially grow in size at runtime.

Automatic memory allocation refers to the process by which variables local to a procedure are allocated on the stack. When a procedure is executed, the required storage space for its local variables, having been determined by the compiler, is automatically reserved on the procedure stack. Similarly, when a function exists, its local variables are automatically deallocated. Unlike global variables, the deallocation of local variables introduces the possibility of memory safety violations. If the address of an automatically allocated objects "escapes" and is made available outside the procedure in which the object is allocated, all references to the object's location become invalid when the procedure exists and the object is deallocated. Pointers to a deallocated object, whether the object was originally allocated on the stack or the heap, are collectively referred to as *dangling* pointers, and their dereference is a violation of temporal safety.

Dynamic memory allocation refers to the process by which blocks of memory of arbitrary size can be allocated dynamically at runtime on the system heap. This is accomplished with the `malloc` standard library function, which allocates a region of memory and returns either a pointer to its base address or the `NULL` pointer, indicating that a block of the specified size could not be allocated. Dynamically allocated memory is deallocated by passing, to the `free` function, a pointer that refers to the base address of the allocated block. In giving the programmer complete control over memory allocation and deallocation, memory safety violations related to

dynamically allocated memory are commonplace. Dereferences of dangling pointers to heap-allocated objects occur if a pointer to an object is dereferenced after the object is deallocated. Similarly, a violation occurs if an attempt is made to deallocate an object more than once with the `free` function. However, an example of a related programming error that does not result in a memory safety violation is the "memory leak." A leak occurs when a pointer to dynamically allocated memory is lost, and the program is never able to reclaim the allocated space. Since this is not a violation of spatial or temporal safety, MemSafe does not aim to detect such an error.

2.2 Compilation

A C language compiler [7] is a computer program that transforms a program written in C into a machine executable form. A compiler is responsible for performing a variety of tasks, including lexical and semantic analysis of the source code and machine code generation and optimization. Typically, tasks are organized into three groups based on the order in which they occur: the compiler frontend, middle-end and backend.

The main task of a compiler frontend is to analyze the source code in order to build an internal target-independent representation of the program, called the intermediate representation (IR), for use by the middle-end. This process includes the following steps: (1) *Lexical analysis* involves the tokenization of the source code in order to recognize keywords, identifiers, and symbols. (2) *Preprocessing* involves performing macro substitutions and the processing of inclusion, conditional compilation, and other preprocessor directives. (3) *Syntax analysis* involves parsing the resulting sequence

of tokens to identify the syntactic structure of the program. (4) *Semantic analysis*, the final step, involves performing various semantic checks (e.g. type checking) of the program's structure and is responsible for rejecting incorrect programs and issuing compiler warnings.

The primary task of the compiler backend is to transform the IR into a machine executable representation. This process involves the following steps: (1) Target-dependant *analysis* and *optimization* involves transforming a program into a functionally equivalent, but optimized, program based on features of the target machine (e.g., a machine's memory hierarchy). (2) *Instruction selection* involves selecting the appropriate machine instructions to implement a given operation or operations present in the IR. (3) *Instruction scheduling* involves determining the order in which the selected instructions are placed based on their latency on the target machine. (4) *Register allocation*, typically one of the final steps, involves assigning the large number of program variables to the much smaller number of machine registers.

Finally, the compiler middle-end is responsible for performing target-independent analyses and optimizations. These processes are performed after the source code is converted into the intermediate representation by the frontend and before the IR is converted into machine code by the backend. The analyses and optimizations performed by the middle-end (some of which are the focus of Section 2.3) are typically intended for gathering machine-independent information about the structure of a program and using this information to transform the program such that it becomes faster or smaller than the original version. Example analyses include call graph construction, control- and data-flow analysis, and alias analysis. Common optimizations include constant

propagation and dead code elimination. MemSafe's analysis and transformation is target-independent and operates in the middle-end of a C compiler.

2.3 Analysis

Compiler analysis is the process of gathering information about a source program's structure in order to transform, or in some way optimize, it such that an attribute of the executable program is maximized or minimized. Frequently, this involves the optimization of programs to minimize execution time, memory consumption, or power consumption. Given that some of the most basic compiler analyses have been shown to be undecidable [54], the goal of most analyses and optimizations is not to produce a program that is necessarily "optimal" in any way. Rather the goal is to apply heuristics that improve desirable characteristics of a "typical" program.

Compiler analyses can be grouped and categorized in several ways based on their scope and precision. The scope of an analysis can be (1) *intraprocedural*, meaning that each procedure is considered individually, (2) *interprocedural*, meaning that multiple procedures are considered at the same time, and (3) *whole-program*, meaning that the analysis considers the entire program at once. Interprocedural and whole-program analyses can acquire a greater amount of information about a program's behavior and lead to more effective optimizations. However, because they must reason over more of the program at once, these analyses can be complex and require a compilation time that can often be impractical.

Additionally, analyses can be grouped by the precision of their results. An analysis

is (1) *flow-sensitive* if it takes instruction ordering into consideration, and (2) *context-sensitive* if it takes calling context into consideration when analyzing the target procedure of function call. Flow- and context-insensitive analyses are more efficient to perform than the above, but produce results that are not as precise.

Many compiler analyses exist, and they are useful for performing a variety of program optimizations. However, only those required by MemSafe's transformation for ensuring memory safety are presented in this section.

2.3.1 Control-flow and call graph construction

A program's potential execution paths can be effectively represented using control-flow and call graphs. The construction of a control-flow graph (CFG) is an itraprocedural analysis that produces a graph $G = (V, E)$ where V is a set of vertices representing each basic block in a function, and E is a set of edges such that if control-flow can transfer from a block b_1 to another block b_2, there is an edge (b_1, b_2) in E. A sequence of instructions forms a *basic block* [7], if the sequence has only one entry point, meaning that only the first instruction is the target of a branch instruction, and has only one exit point, meaning that only the last instruction can cause the program to begin executing in a different basic block. The CFG is essential for many compiler analyses and optimizations. For example, backedges in the CFG are useful in recognizing loops, and disconnected subgraphs of the CFG are useful in identifying unreachable code.

The construction of a call graph is an interprocedural analysis that produces a graph $G = (V, E)$ where V is a set of vertices representing each function in a program,

and E is a set of edges such that if a function f calls another function g, there is an edge (f, g) in E. In the presence of function pointers, determining the exact call graph of a program is undecidable, so graph construction algorithms must produce over-approximations. That is, a program's call graph contains every call relationship that can potentially be realized at runtime in addition to other spurious relationships that can never occur. Frequently, alias analysis (discussed below) is used to determine a set of the potential targets of a function pointer. Call graphs are essential for most interprocedural analyses and optimizations.

2.3.2 Data-flow analysis

Data-flow analysis is a technique for reasoning about the possible set of values computed at various points in a computer program. Intuitively, a program's CFG is used to determine the locations in a program where particular values might propagate during the program's execution. Data-flow analysis is used by a compiler to perform a variety of optimizations including constant folding, dead code elimination and common-subexpression elimination [7].

Data-flow analysis is performed by iteratively solving a set of equations for each basic block of the CFG until the system of equations stabilizes and reaches a fixpoint [51]. Information is gathered at the boundaries among basic blocks instead of individual instructions since, once the information is gathered for each block, it is trivial to compute the required information for each point within a block. A data-flow analysis can be *forward* or *backward*, depending on the type of analysis being performed. In a

forward analysis, the exit state of a basic block is a function of its entry state, and the entry state of a block is a function of the exit states of the block's predecessors. Thus, for each basic block b in a program, a forward data-flow analysis can be characterized by the following equations:

$$in_b = join_{p \in pred_b}(out_p)$$
$$out_b = trans_b(in_b)$$

where $trans_b$, called the *transfer function* of block b, produces the exit state of b for a given input sate. The $join$ operation (usually set intersection or union) combines the exit states of the predecessors of b to form the entry state of b. These equations are applied iteratively to each basic block in the CFG until the entry and exit states of each no longer change. A backward data-flow analysis operates analogously but with the direction of the transfer function reversed.

2.3.3 SSA form

The Static Single Assignment (SSA) [26] form is a compiler intermediate representation in which every variable is assigned exactly once. SSA ensures that a use of a variable is dependent on exactly one definition, which greatly simplifies the construction of use-def and def-use chains [7]. Furthermore, all evaluations of variables having the same name are required to produce the same value. The primary reason for converting a program into SSA form is that, because of the above properties, many compiler analysis and optimization algorithms are enabled or significantly enhanced.

Within a single basic block, it is trivial to assign every definition a unique name.

For example, consider the statement `x = 1` and the following statement `x = 2`. In SSA form, these two assignments are represent as `x1 = 1` and `x2 = 2`. All remaining uses of the original variable x would now be uses of variable x_2. However, at the point where control-flow paths merge, it is not obvious how each variable can have exactly one definition. Assume a basic block b has exactly two predecessor blocks and that variable x is assigned in each one, becoming definitions of x_1 and x_2 in SSA form. The value of x that is used in b is dependent on the control-flow of the program. Therefore, this uncertainty in control-flow is resolved using a special statement `x3 =` ϕ`(x1, x2)`, which is inserted at the beginning of b, meaning that the value of x_3 is either equal to the value of x_1 or x_2. The ϕ-function produces the value of x_1 if the predecessor of b contains the definition of x_1 and produces the value of x_2 if the predecessor contains the definition of x_2.

The algorithm for converting a program into SSA form is straightforward. It first inserts the required ϕ-functions to resolve the uncertainty in control-flow, and it then renames all definitions and uses of variables such that each assignment is given a unique name. The algorithm relies on a data-flow analysis that computes the *dominance frontier* [7] of each basic block in order to efficiently determine the locations of the required ϕ-functions.

2.3.4 Alias analysis

Alias analysis is a type of data-flow analysis that is used to determine whether the data stored at a particular location in memory may be accessed in more than one

way. Two pointers are said to *alias* if they refer to the same location and, therefore, could both be used to access the same data. Alias analysis is responsible for deciding, for any two pointers, whether they *must*, *must not*, or *may* alias. The analysis is commonly used for determining, for example, whether a value stored to memory (e.g., `*q = x`) can affect a value loaded from memory (e.g., `x = *p`). Such would be the case if the two pointers (p and q) must alias. Compilers perform alias analysis because it can significantly improve the performance of other analyses and transformations.

However, the usefulness of an alias analysis depends on the precision of the analysis. For example, an analysis the produces a *may-alias* response for every alias query would not improve the effectiveness of other analyses and optimizations. Flow- and context-sensitive algorithms produce the most precise results yet typically require far too much computation time and memory to be very useful in practice.

Andersen's analysis [4] is widely-regarded to be one of the most precise inter-procedural flow- and context-insensitive alias analyses. Even so, it requires $O(n^3)$ computation time. In practice, techniques such as online cycle elimination [35] and offline variable substitution [72] are required to make the analysis more efficient and scalable to large programs. Because of its balance between precision and efficiency, MemSafe uses Andersen's analysis as its default alias analysis.

Chapter 3

Memory Safety Violations and Prior Enforcement Methods

Having introduced the C programming language, the processes by which it is compiled and optimized, and the low-level features of C that can enable or contribute to the existence of hard-to-detect memory errors, this chapter examines the resulting memory safety violations in more detail. Section 3.1 presents the spatial and temporal errors that can result from the misuse of the low-level C language features, and Section 3.2 gives and overview of previous methods for detecting some or all memory access violations in C programs. The overall strengths and weaknesses of these strategies are compared with that of MemSafe.

3.1 Memory Safety Violations

Memory safety violations can be divided into two categories: violations of *spatial safety* and violations of *temporal safety*. A spatial safety violation is an error in which a pointer is used to access the data at a location in memory that is outside the bounds of an allocated object. The error is "spatial" in the sense that the dereferenced pointer

Memory Safety	Violation	Example

Spatial Safety	Bounds violation	`struct { ... int array[100]; ... } s;`
		`int *p;`
		`...`
		`p = &(s.array[101]);`
		`... *p ...` ▷ *bounds violation*

	Uninitialized pointer	`int *p;`
		`...`
		`... *p ...` ▷ *uninitialized pointer dereference*

	NULL pointer	`int *p;`
		`...`
		`p = NULL;`
		`... *p ...` ▷ *null pointer dereference*

	Manufactured pointer	`int *p;`
		`...`
		`p = (int*) 42;`
		`... *p ...` ▷ *manufactured pointer dereference*

Temporal Safety	Dangling stack pointer	`int *p;`
		`...`
		`void f() {`
		`  int x;`
		`  ...`
		`  p = &x;`
		`}`
		`...`
		`void g() {`
		`  f();`
		`  ... *p ...` ▷ *dangling stack pointer dereference*
		`}`

	Dangling heap pointer	`int *p, *q;`
		`...`
		`p = (int*) malloc(10*sizeof(int));`
		`q = p;`
		`...`
		`free(p);`
		`... *q ...` ▷ *dangling heap pointer dereference*

	Multiple deallocations	`int *p, *q;`
		`...`
		`p = (int*) malloc(10*sizeof(int));`
		`q = p;`
		`...`
		`free(p);`
		`free(q);` ▷ *multiple deallocations*

Table 3.1: Memory safety violations. Example code fragments demonstrating memory safety violations are presented grouped by whether they affect aspects of spatial or temporal safety.

refers to an incorrect location in memory. A temporal safety violation is an error in which a pointer is used in an attempt to access or deallocate an object that has already been deallocated. The violation is "temporal" in the sense that the pointer use occurs at an invalid instance during the execution of the program (i.e., after the object to which it refers has been deallocated). Table 3.1 lists spatial and temporal memory safety violations that MemSafe detects and gives examples of each. These errors are discussed in detail below.

3.1.1 Bounds violations

A spatial violation occurs when a pointer is used to access a location outsize the bounds of an allocated object. Common examples include accessing elements beyond the end of an array and dereferencing a pointer derived from invalid pointer arithmetic. Note that a pointer must be dereferenced for a violation to occur; it is not sufficient for a pointer to simply refer to a location outside the bounds of an object to cause a spatial safety violation. In many C programs, it is common for a pointer to be out-of-bounds and later refer to an in-bounds location.

The first category of Table 3.1 ("Bounds violation") shows the dereference of a pointer that is out-of-bounds. Here, a pointer p is created that refers to a location beyond the last element of the *array* field of structure s. The dereference of p is an example of a "sub-object" bounds violation. A sub-object is an object that is allocated at part of a larger, nested structure. Examples include arrays of structures and structures containing array fields. Sub-object bounds violations frequently go

undetected by error detection mechanisms since an out-of-bounds pointer to a sub-object can remain *within* the bounds of the outer object.

3.1.2 Uninitialized pointer dereference

The dereference of an uninitialized pointer also results in a spatial violation. Statically allocated pointer values that are not given an initial value are typically initialized by the compiler to be equal to `NULL`. Therefore, a dereference of such a pointer, while still resulting in a spatial violation, commonly results in a runtime error on many systems. However, automatically allocated pointer values that are not given an initial value refer to whatever address is specified in the location on the stack in which they are stored. Thus, since it is not known statically what this value might be, the dereference of an uninitialized pointer that is allocated on the stack might result in data corruption or cause the program to eventually crash in a way that obfuscates the root problem. The second category of Table 3.1 shows an example of an uninitialized pointer dereference.

3.1.3 Null pointer dereference

The `NULL` pointer, often—but not necessarily—equal the value zero, is used to indicate that a pointer refers to no object. Thus, the dereference of a `NULL` pointer results in a spatial violation. Commonly, the dereference of a `NULL` pointer causes a runtime error and immediately halts a program with a segmentation fault, since an operating system never allocates a running program the address of `NULL`.

However, this is not true for all systems. The XScale [45] and other ARM

microprocessors [8] that lack virtual memory reserve address zero for their interrupt vector table. An interrupt vector table contains the memory addresses of interrupt handlers and is used by a processor to determine the correct response to hardware interrupts and exceptions. A program running on these microprocessors is able to modify the contents of the interrupt vector table by dereferencing the `NULL` pointer, which results in a security exploit known as the vector rewrite attack [47]. The third category of table 3.1 shows an example of a `NULL` pointer dereference.

3.1.4 Manufactured pointer dereference

A "manufactured" pointer is a pointer created by a means other than explicit memory allocation (i.e., with the `malloc` standard library function) or by using the address-of operator (`&`). Type-casting an integral type to a pointer type is a common method for creating such a pointer, and in doing so, results in a pointer that refers to an address equal to the value of the integer. The dereference of a manufactured pointer results in a spatial safety violation since the layout of objects in memory is not specified by the C language, and therefore, object addresses are unknown to a programmer at compile-time.[1] The fourth category of Table 3.1 shows the dereference of a pointer that is type-cast from integer 42.

[1]Memory-mapped I/O locations (discussed in Section 4.3.6) are an exception to this rule since the addressable memory map is made known to a programmer.

3.1.5 Dereference of dangling stack pointers

As discussed previously, if the address of an automatic variable is made available outside the function in which it is defined (e.g., by assigning it to a global pointer or storing it in a heap object), all pointers to the local variable become dangling when the function exists and its local storage is deallocated. Since a function's local storage may be reallocated for the execution of another function, reading or writing the address of a previously allocated variable results in a of temporal violation.

The fifth row of table 3.1 shows two functions f and g, and function g is shown calling f. However, in function f, the address of local variable x is assigned to the global pointer p. The dereference of p in function g after the call to f results in a temporal safety violation.

3.1.6 Dereference of dangling heap pointers

Like pointers to local variables, pointers to dynamically allocated memory become dangling when the objects to which they refer are deallocated. In this case, memory is deallocated dynamically by `free` instead of automatically when a function exits. Since some or all of the storage occupied by a deallocated object may be subsequently reallocated with `malloc` for another object of an incompatible type, reading or writing the deallocated location is undefined, and results in a violation of temporal safety.

The sixth row of Table 3.1 shows two pointers p and q that must alias. A dynamic array of integers is allocated with `malloc`, and the base address of the allocated region (assuming the allocation was successful) is assigned to both p and q. Since p is used

to deallocate the array with a call to `free`, the subsequent dereference of pointer q is a dangling pointer dereference, and results in a temporal safety violation.

3.1.7 Multiple deallocations

A program that attempts to deallocate the same object more than once, or attempts to deallocate a location that was not originally allocated by `malloc`, commits a temporal safety violation. Calling `free` twice with the same value or with a value not returned by `malloc` typically corrupts the internal data structures of a system's memory allocator. This can result in application data corruption, in `malloc` returning the same value for subsequent allocations, and in the eventual crash of a program.

The last row of Table 3.1 shows two pointers p and q that must alias. A dynamic array of integers is again allocated with `malloc`, and the base address of the allocated region (assuming the allocation was successful) is assigned to both p and q. Since p is used to deallocate the array with a call to `free`, the subsequent call to `free` using pointer q results in multiple deallocation attempts of the same objects, and causes a violation of temporal safety.

3.2 Prior Enforcement Methods

As evidence of the significance of the above memory safety violations, the instrumentation of C programs to ensure memory safety remains an actively researched topic. This section reviews previous approaches for detecting some or all spatial and temporal safety violations, primarily focusing on the prior works' use of metadata. In

the context of enforcing memory safety, *metadata* refers to the creation of additional data for describing the spatial or temporal properties of an object or pointer. For example, metadata often consists of the base and bound addresses that indicate the valid address range to which a pointer may refer.

3.2.1 Spatial safety

The goal of spatial safety is to ensure that every memory access occurs within the bounds of a known object. Spatial safety is typically enforced by inserting runtime checks before pointer dereferences. Alternatively, checking for bounds violations after pointer arithmetic is also possible [e.g. 3, 30, 49, 74], but requires care since pointers in C are allowed to be out-of-bounds so long as they are not dereferenced. The metadata required for spatial safety checks can be associated either with objects or pointers, and there are strengths and weaknesses of each approach.

3.2.1.1 Object metadata

Methods that utilize object metadata usually record the base and bound addresses of objects, as they are allocated, in a global database that relates every address in an allocated region to the metadata of its corresponding object. Advantages of this approach include efficiency, since it avoids the propagation of metadata at every pointer assignment (see the discussion of pointer metadata below), and compatibility, since it does not change the layout of objects in memory or prohibit the use of pre-compiled libraries. Prominent methods employing this strategy include the work by Jones and Kelly [49], Ruwase and Lam [74], Dhurjati and Adve [30], Akritidis et al. [3], SafeCode

[31] and SVA [24].

However, the use of object metadata as means of enforcing spatial safety results in several drawbacks. First, this approach prevents complete spatial safety. Since nested objects (e.g., an array of structures) are assigned a base and bound address that spans the entire allocated region, it is impossible to detect sub-object overflows if an out-of-bounds pointer to an inner object remains *within* bounds of the outer object. Second, this approach requires a runtime lookup operation to be performed in order to retrieve metadata from the object database. Dhurjati and Adve [30] improve the runtime cost associated with this lookup operation by partitioning the object database using Automatic Pool Allocation [55, ch. 5], and Akritidis et al. [3] improve runtime by constraining the size and alignment of allocated objects. However, these methods do not detect sub-object overflows or temporal errors.

Figure 3.1 depicts the utility of object metadata (shown in red) in enforcing memory safety. In order to ensure memory safety, complete spatial and temporal safety must be enforced. Since all pointer dereferences are either object-level references or sub-object references, it follows that all object and sub-object references must be both spatially and temporally safe for a program to be memory safe. However, object-level base and bound information is only useful in enforcing object-level spatial safety, since sub-objects must share metadata with their corresponding outer objects. [2] Figure 3.1 will be referenced again when describing the remaining prior enforcement strategies.

[2]Some methods [e.g., 3, 24, 30, 31, 49, 74] are capable of using object metadata to detect some, but not all, temporal safety violations. However, if an object is deallocated and its space is reallocated for use by another object, dangling pointer dereferences to the original object will not be detected because they are within bounds of the new object. Thus, these methods are incapable of enforcing either complete object-level or sub-object temporal safety.

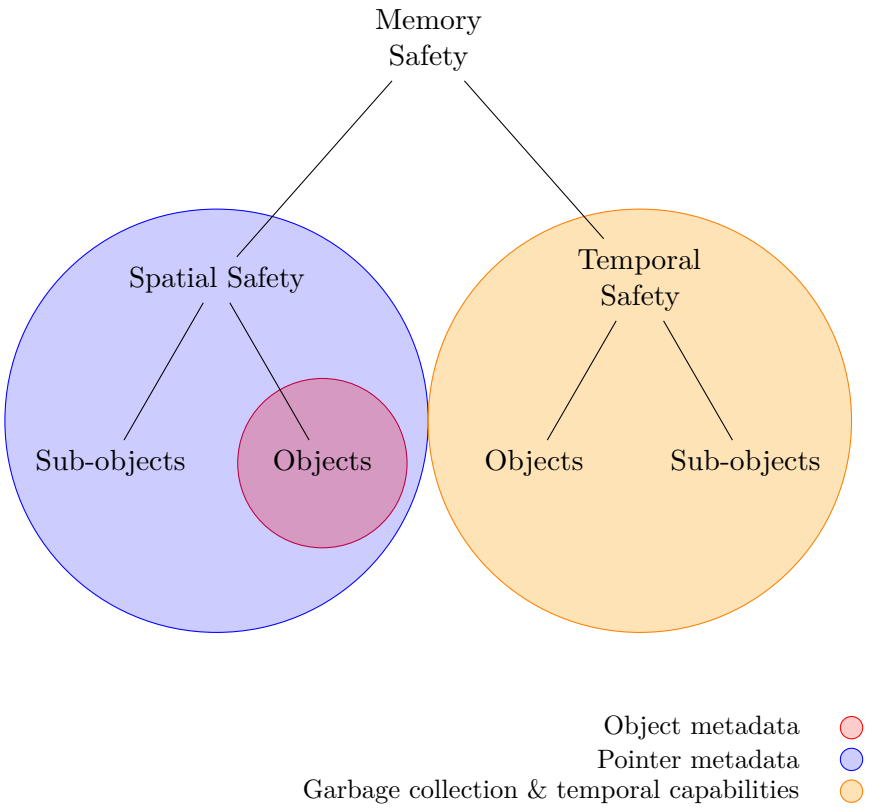


Figure 3.1: Prior enforcement methods. The use of spatial metadata, garbage collection, and temporal capabilities is shown for previous methods of enforcing memory safety.

3.2.1.2 Pointer metadata

An alternative to using object metadata for enforcing spatial safety is to associate metadata with individual pointers. When a new pointer is created (i.e., with `malloc` or the address-of operator), its metadata is initialized to be the base and bound address of its referent, and when a pointer definition uses the value of another pointer (e.g., pointer arithmetic), its metadata is inherited from the original pointer. Advantages of this approach include avoiding costly database lookups and the ability to ensure complete safety, since sub-object overflows can be detected by assigning each pointer a unique base and bound address.

Pointer metadata is commonly implemented using multi-word blocks of memory, called "fat-pointers," that record the required base and bound information inline with pointers. Each pointer in a program essentially becomes a `struct` containing three fields: the original pointer value and the base and bound address of its referent. Prominent methods employing this strategy include Safe C [9], Fail-Safe C [66] and CCured [64]. However, the use of inline metadata is not always compatible and breaks many programs. Since a pointer's size is no longer equal to the word size of the target architecture, many programming idioms no longer work as expected. Additionally, interfacing with external libraries becomes difficult and requires wrapper functions to pack and unpack fat-pointers at boundaries with uninstrumented code.

Several pointer-based methods have developed approaches that avoid some of the compatibility issues of fat-pointers. CCured [64], MSCC [82] and Patil and Fischer [68] record metadata in disjoint structures that mirror the shape of the underlying

data, but maintaining this representation increases runtime. Fail-Safe C [66] combines fat-pointers with fat-integers and virtual structure offsets, but this too increases cost. Finally, Softbound [62] maintains metadata for in-memory pointers in an efficient global lookup table, but this method only detects spatial violations.

Another disadvantage of the use of pointer metadata as a means of enforcing spatial safety is its runtime cost. While it avoids the need for expensive database lookups operations, metadata must instead be propagated at every pointer assignment. CCured [64] reduces metadata propagation by using a type system to infer pointer usage. CCured classifies pointers as `SAFE`, `SEQ` and `WILD` and optimizes the inserted checks and code for propagating metadata for each pointer kind. However, CCured requires manual code modifications to avoid the expensive bookkeeping of `WILD` pointers and to correct the compatibility issues of fat-pointers.

Figure 3.1 depicts the utility of pointer metadata (shown in blue) in enforcing memory safety. Since individual pointers can be associated with a unique base and bound address, pointer metadata can be used to enforce complete object-level and sub-object spatial safety. However, prior methods are not capable of utilizing pointer metadata in enforcing temporal safety.

3.2.1.3 MemSafe's approach for ensuring spatial safety

MemSafe's use of metadata as a means for ensuring spatial safety avoids the drawbacks of the above approaches. MemSafe captures the most salient features of object and pointer metadata in a hybrid representation. To ensure complete and compatible spatial safety, MemSafe maintains disjoint pointer-based metadata in an approach

similar to that of SoftBound. However, to lower runtime cost, MemSafe models temporal errors as spatial errors and propagates pointer-based metadata only when it is needed for performing runtime checks. Additionally, MemSafe maintains some object-based metadata in a global database but performs lookup operations only when MemSafe's pointer-based metadata is insufficient for ensuring temporal safety.

3.2.2 Temporal safety

The goal of enforcing temporal safety is to ensure that every memory accesses refers to an object that has not been deallocated. As described in Section 3.1, temporal safety violations occur when dereferencing pointers to stack objects, if the function in which they were defined has exited, and when dereferencing pointers to heap objects, if the object to which they refer has been deallocated with `free`. Temporal safety is typically enforced with garbage collection or by software checks. Like the methods for ensuring spatial safety, there are strengths and weaknesses of each approach.

3.2.2.1 Garbage collection

Methods using garbage collection to prevent dangling pointers to heap objects commonly ignore calls to the `free` function and replace calls to `malloc` with the Boehm-Demers-Weiser conservative garbage collector [16]. To prevent dangling pointers to stack objects, local variables can be "heapified" and replaced with dynamically allocated objects that are managed by the garbage collector. This is the approach taken by CCured [64] and Fail-Safe C [66].

However, garbage collection negates several of C's primary benefits, including its

predictability and low-level access to memory. Garbage collection voids real-time guarantees [10], increases address space requirements, reduces reference locality, and increases page fault and cache miss rates [84]. Moreover, since the collector must be conservative, some memory may never be reclaimed by the system, resulting in memory leaks. Finally, heapifying stack objects increases the runtime overhead of enforcing temporal safety since dynamic allocation is slower than automatic allocation.

Despite these drawbacks, conservative garbage collection is capable of enforcing complete temporal safety. This capability is depicted in Figure 3.1, where the use of garbage collection is shown in orange.

3.2.2.2 Temporal checks

An alternative to using garbage collection for enforcing temporal safety is to insert explicit software checks that test the temporal validity of referenced objects. To achieve this, a "capability store" is commonly used to record the temporal capability of objects as they are created and destroyed. Additional temporal metadata that is created and propagated with spatial metadata links a pointer to the temporal capability of its referent. Methods employing this strategy include Safe C [9], MSCC [82], and the work by Patil and Fischer [68] and Yong and Horwitz [83].

There are advantages and disadvantages of using explicit temporal checks for enforcing temporal safety. The primary strength of this approach is that it retains C's memory allocation model and avoids the drawbacks associated with garbage collection. However, the inclusion of additional runtime checks and metadata significantly increases the runtime overhead beyond that of enforcing spatial safety alone. Figure 3.1 indicates

that temporal capabilities (shown in orange), like garbage collection, can be used to enforce complete temporal safety.

3.2.2.3 MemSafe's approach for ensuring temporal safety

One of MemSafe's main contributions is the modeling of temporal errors as spatial errors. Therefore, MemSafe does not require conservative garbage collection or explicit temporal checks, and it avoids the drawbacks of both approaches. Instead, MemSafe relies on spatial safety checks and the hybrid metadata representation mentioned above for ensuring temporal safety.

Chapter 4

MemSafe

MemSafe [77] is a compiler analysis and transformation for ensuring the spatial and temporal memory safety of C programs at runtime. MemSafe's source code transformation ensures complete memory safety, produces transformed code that is compatible with legacy software, and is entirely automatic. In order to reduce the runtime cost of enforcing memory safety, MemSafe requires a limited amount of static analysis—an alias analysis used for disambiguating memory operations—to avoid inserting unnecessary checks and the propagation of unneeded metadata. MemSafe inserts runtime checks and propagates the required metadata for the remaining memory accesses that cannot be statically verified to be safe.

This chapter describes MemSafe's basic, unoptimized approach for ensuring the memory safety of C programs. (Chapter 5 will describe optimization techniques for lowering the runtime cost associated with enforcing memory safety.) Section 4.1 presents a small C-like language and introduces syntax extensions of this language that MemSafe uses to model the challenging aspects of enforcing memory safety—namely, memory deallocation and pointer aliasing. Sections 4.2–4.3 present the runtime checks and metadata propagation rules required to achieve safety. Section 4.4 discusses

$$
\begin{array}{rl}
\text{Atomic Types } \alpha & := \texttt{int} \mid \tau\texttt{*} \\
\text{Types } \tau & := \alpha \mid \texttt{struct}\{d^+\} \mid \tau\texttt{[}n\texttt{]} \\
\text{Declarations } d & := \tau\ x\texttt{;} \\
\text{Functions } f & := \texttt{func(}x^*\texttt{) \{}b^*\texttt{\}} \\
\text{Blocks } b & := p^*\ d^*\ s^+ \\
\phi\text{-Functions } p & := x\ \texttt{=}\ \phi\texttt{(}x^+\texttt{);} \\
\text{LHS Expressions } l & := x \mid \texttt{*}l \mid l\texttt{.}y \\
\text{RHS Expressions } r & := r\texttt{+}r \mid l \mid \texttt{\&}l \mid \texttt{(}\alpha\texttt{)}r \mid \texttt{malloc(}r\texttt{)} \mid n \\
\text{Statements } s & := l\texttt{=}r\texttt{;} \mid l\texttt{(}r\texttt{);} \mid \texttt{for(}l\texttt{=}r\texttt{;}\ r\texttt{<}r\texttt{;}\ l\texttt{=}r\texttt{) \{}b\texttt{\}} \\
& \quad \mid \texttt{if(}r\texttt{) \{}b\texttt{\} else \{}b\texttt{\}} \mid \texttt{return}\ r\texttt{;} \mid \texttt{free(}r\texttt{);} \\
\text{where } & x \in \text{variables} \\
& y \in \text{structure field identifiers} \\
& n \in \mathbb{N}
\end{array}
$$

Figure 4.1: Language Syntax for MemSafe presentation. Syntax is given for a simple SSA [26] language with procedures, pointers, control flow, and manual memory management.

the use of MemSafe for ensuring the memory safety of multithreaded programs and describes the changes to MemSafe's basic approach that are required in order to achieve thread safety. Finally, Section 4.5 shows an implementation of MemSafe's unoptimized checks and metadata for an example real-world application.

4.1 Language Extensions and Assumptions

This section describes the main components of MemSafe's program analysis. Because the C programming language, in its entirety, is both large and complex, the language defined in Figure 4.1 will be used for describing MemSafe's source code translations and the rules for runtime check insertion and metadata propagation. Figure 4.1 defines a small SSA [26] intermediate language that captures all the relevant pointer-related portions of C. Features of the language include, among others, syntax for pointer types, manual memory management, type-casting of pointer values, pointer arithmetic,

and complex control-flow.

Without loss of generality, the following assumptions are made of the language presented in Figure 4.1. First, it is assumed that memory is only accessed with explicit load (e.g., `x = *ptr`) and store (e.g., `*ptr = x`) operations involving pointers. Second, it is assumed that pointer values are only created with the address-of operator (`&`) or by calling the `malloc` function. Recall that in C, a variable declared as an array of some particular type can act as a pointer to that type, and when used by itself, is a pointer that points to the first element of the array. To enforce the notion that pointers are only created through the two mechanisms above, all array accesses are represented as an indexing operation applied to the address of the first element of the array. For example, for the allocation of an array a of ten elements, an access of the fifth element `a[4]` is represented as `(&a[0])[4]`. That is, a pointer is created to the first element of the array, and then this pointer is used to compute the address of the fifth element. In this way, all new pointer values may only be created with the address-of operator and by calling the `malloc` system function.

Furthermore, MemSafe assumes all global variable definitions define a symbol that provides the address of an object instead of the actual object "contents." Since assignments of global variables must be conservatively accounted for in SSA form [26], compiler intermediate representations (e.g., LLVM [56]) often represent global variables as pointers to statically allocated regions of memory. The advantage of this approach is that within a procedure, a global variable can be loaded from memory, renamed according to the SSA conversion algorithm, and then stored back to memory before control-flow reaches another procedure. Therefore, MemSafe identifies statically

allocated objects by their location in memory. Note that MemSafe's representation of global variables is analogous to the discussion of arrays above in that the declaration of an object implicitly creates a pointer to that object.

MemSafe models both memory deallocation and pointer store operations as explicit assignments using syntax extensions to this C-like SSA language. The advantage of this approach is that it enables MemSafe to ensure complete memory safety by reasoning solely about pointer definitions, which eliminates the need for separate mechanisms for detecting spatial and temporal errors and reveals optimization opportunities. The remainder of this section describes these abstractions.

4.1.1 Memory deallocation

Memory deallocation can implicitly change the object to which a pointer refers. If the region of memory that was occupied by a deallocated object is ever reallocated, the contents of the region may change, and any remaining pointers to the original object implicitly become invalid. This implicit redefinition of pointers can be made apparent by modeling both automatic and dynamic memory deallocation as an explicit pointer assignment. For example, MemSafe models the statement `free(p)` as `p = invalid`, where *invalid* is a special untyped pointer constant that points to an "invalid" region of memory. The base and bound address associated with this abstract memory region are defined by the impossible address range $[1, 0]$. Thus, if the spatial metadata of p is located at address $addr_p$ in memory, then p could be associated with the base and bound of the *invalid* pointer by the statements `addr_p->base = 1` and

`addr_p->bound = 0`. Since the size of this block is -1, spatial safety checks involving the base and bound of the *invalid* pointer are guaranteed to always report a memory safety violation. Therefore, temporal safety violations can be detected with runtime checks inserted for enforcing spatial safety. The rules for inserting assignments of the *invalid* pointer are given below.

4.1.1.1 Automatic memory deallocation

If the address of a stack-allocated object is taken with the address-of operator (`&`), the pointer to this object may "escape" and be made available outside the function in which the object is allocated. Such an occurrence is possible, for example, if a local variable's address is stored in a global or heap variable. While this is a legal operation in the C programming language, a common consequence of escaping pointers is the program committing a temporal safety violation. When a function exits, its local variables are automatically deallocated, and any escaping pointers to these deallocated objects become dangling. To make the implicit redefinition of these pointers explicit, MemSafe inserts assignments of the *invalid* pointer at the end of a procedure for each of its local variables whose address is taken. MemSafe assumes the address of a local variable escapes if it is ever stored in another variable. Assignments of the *invalid* pointer are inserted according to the following rule for automatic memory deallocation. Numbered lines represent original code.

Syntax Extension 4.1—Automatic memory deallocation:

```
1:  void f() {
2:    struct { ...  int array[100]; ...  } s, *p;
3:    ...
```

```
4:    p = &s;                 ▷ address of s is taken and may escape
5:    ...
      p = invalid;            ▷ MemSafe models deallocation as an explicit
6:  }                           pointer assignment of 'invalid'
```

In this example, the nested structure s is allocated automatically on the stack as a local variable of function f. In line 4, the address of s is taken and stored in pointer p. It is assumed that p may escape to another procedure and result in a dangling pointer when function f exits. Therefore, MemSafe assigns p the value of the *invalid* pointer before the function exits, indicating that the pointer now refers to a temporally "invalid" region of memory. After this assignment, the base and bound of p would be updated to be equal to that of the *invalid* pointer, and any pointer derived from p would inherit this metadata as well (see Section 4.3).

4.1.1.2 Dynamic memory deallocation

If a pointer's referent is deallocated dynamically by a program calling the `free` function, all pointers that refer to this object become dangling pointers. The subsequent dereference of a dangling pointer results in a temporal safety violation. To make the redefinition of these pointers explicit, MemSafe inserts assignments of the *invalid* pointer after calls to `free` for the pointer used in deallocating the object. Assignments of the *invalid* pointer are inserted according to the following rule for dynamic memory deallocation. Numbered lines represent original code.

Syntax Extension 4.2—Dynamic memory deallocation:

```
1:  int *p;
2:  ...
3:  p = malloc(size);
```

```
4:  ...
5:  free(p);                  ▷ MemSafe models deallocation as an explicit
    p = invalid;                pointer assignment of 'invalid'
```

In this example, an object of *size* bytes is dynamically allocated by a program with the `malloc` function, and the base address of object is assigned to pointer p. In line 5, the object to which p refers is deallocated by the program with the `free` function. Therefore, MemSafe assigns p the value of the *invalid* pointer to indicate that the pointer now refers to a temporally "invalid" region of memory.

4.1.1.3 Inserting assignments of the *invalid* pointer

MemSafe inserts assignments of the *invalid* pointer according to the above rules for the deallocation of stack- and heap-allocated objects. Since global variables have a lifetime equal to that of the program, they are not deallocated until the program terminates and, therefore, do not require assignments of the *invalid* pointer. MemSafe removes all inserted assignments of *invalid* after instrumenting the program with the required safety checks and code for propagating metadata. The pseudocode of the algorithm for inserting assignments of the *invalid* pointer is given below. Pseudocode conventions follow those of Cormen et al. [23].

Algorithm 4.1—Pseudocode for inserting assignments of *invalid*:

```
1: for each function f in the program
2:     do for each instruction i in function f
3:         do if i defines a pointer p
4:             then if RHS(i) computes the address of a local variable
5:                     then INVALIDATE-AT-END(p, f)
6:           else if i is a call to function free
7:                     then p ← pointer argument of call
8:                          INVALIDATE-AFTER(p, i)
```

The algorithm for inserting assignments of the *invalid* pointer is straightforward and operates as follows. For each instruction in the program, if the instruction defines a pointer to be equal to the address of a local variable, the pointer is assigned the value of the *invalid* pointer at the end of the function containing the instruction, according to the rule for automatic memory deallocation. The procedure INVALIDATE-AT-END inserts the assignment into the program at the end of the specified function. Otherwise, if the instruction is a call to `free`, the pointer argument of `free` is assigned the value of the *invalid* pointer after the call, according to the rule for dynamic memory deallocation. The procedure INSERT-AFTER inserts the assignment into the program after the specified instruction.

4.1.2 Pointer stores

Having inserted assignments of the *invalid* pointer to make the redefinition of pointers to deallocated memory explicit, MemSafe then transforms the program to make indirect pointer store operations explicit assignments as well.

Indirect assignments are problematic in SSA form and make the representation of pointer stores nonintuitive. A key property of SSA is that each assignment is given a unique name (hence, the "single assignment" condition of Static Single Assignment). However, this property does not hold for in-memory assignments of the form `*p = x`. In this case, `*p` is not given a unique name when it is assigned the value of x, and it is unclear whether other values loaded from memory can be equal to x or not.

To address this problem for the indirect assignment of pointer values, MemSafe

models in-memory pointer assignments (including those induced for the *invalid* pointer) as explicit assignments using alias analysis and a ϕ-like SSA extension called the ϱ-function. In the same way that the ϕ-function of SSA is used to resolve control-flow uncertainty, thereby giving a unique name to conditional assignments at the point where control-flow paths merge, MemSafe uses the ϱ-function to resolve the data-flow uncertainty of pointer values, thereby giving a unique name to indirect pointer assignments at the point where pointers are loaded from memory.

For example, assume the statement `s0:*p = ptr0` is the only direct reaching definition of a pointer defined as `s1:ptr1 = *p`. The statement `s2:*q = ptr2` may *indirectly* redefine ptr_1 if p and q may alias and control-flow may reach statement s_1 from s_2. Therefore, MemSafe models `ptr1 = *p` as `ptr1 = ϱ(ptr0, ptr2)`, meaning the value of ptr_1 may equal that of ptr_0 or ptr_2 but only these two values. In this way, all indirect pointer assignments and object deallocations are represented as direct assignments of the pointers that are potentially modified.

The following code fragment provides a more concrete example of the ϱ-function and demonstrates how this syntax extension creates a synergy with MemSafe's representation of memory deallocation that results in the propagation of the data-flow associated with the *invalid* pointer. Numbered lines represent original code.

Example 4.1—ϱ-function insertion :

```
1:  int  *a,  *b,  *c;
2:  int **p, **q;           ▷ assume p and q may alias
    ...
3:  *q = a;
    ...
4:  if (condition) {
5:    *p = b;
```

```
6:  } else {
7:    free(*p);
      *p = invalid;          ▷ MemSafe models deallocation as an explicit
8:  }                          pointer assignment of 'invalid'
    ...
9:  c0 = *q
    c1 = ϱ(a, b, invalid);   ▷ MemSafe models in-memory data-flow with the
                               ϱ-function.  All subsequent uses of c0 are
                               replaced with uses of c1.
```

In this example, all pointer values exist in memory, and pointer assignments are made possible by pointer store and load operations. After the call to the `free` function in line 7, an in-memory assignment of the *invalid* pointer is inserted to indicate that the referent of `*p` has been deallocated. Since p and q are assumed to potentially alias, this store operation and the ones in lines 3 and 5 may define the pointer loaded and assigned to the variable c_0 in line 9. Therefore, MemSafe resolves this uncertainty in pointer data-flow and gives a unique name to these assignments by inserting the ϱ-function after line 9 and assigning it to the variable c_1. Pointers a, b and *invalid* are added to the ϱ-function assigned to pointer c_1, meaning c_1 may be equal to any of these three values, but only these values. All subsequent uses of pointer c_0 are replaced with uses of c_1.

By default, MemSafe utilizes flow- and context-insensitive pointer alias information to determine the arguments of ϱ-functions. However, MemSafe is capable of using more precise alias analyses, and in general, their uses result in ϱ-functions with smaller arity. In the case of the former, MemSafe performs a simple reachability analysis to improve the results of alias analysis. For example, consider the pointer store operation `*ptr1 = p0` and the pointer load operation `p1 = *ptr2`. If alias analysis indicates

that the pointers ptr_1 and ptr_2 may alias, p_0 would be added to the ϱ-function inserted for p_1. However, if there is no control-flow path from the store operation to the load operation, this is unnecessary since there is no program execution in which the stored value can modify the loaded value. MemSafe does not include stored pointers in the ϱ-functions inserted for loaded pointers if the store cannot reach the load. Note that MemSafe's reachability analysis does not result in a flow-sensitive alias analysis.

4.1.2.1 Inserting ϱ-functions

MemSafe inserts ϱ-functions after all pointer loads, and like the inserted assignments of the *invalid* pointer, they are removed after MemSafe instruments the program with the required runtime checks and code for propagating metadata. Crucially, MemSafe does not insert ϱ-functions for loads of non-pointer values since they are not required for MemSafe's analysis and could potentially lead to a large increase in code size. The pseudocode of the ϱ-function insertion algorithm is given below. Pseudocode conventions follow those of Cormen et al. [23].

Algorithm 4.2—Pseudocode for inserting ϱ-functions:

```
1:  S ← set of all pointer store instructions
2:  L ← set of all pointer load instructions
3:  for each load instruction l ∈ L
4:      do Args ← {}
5:         for each store instruction s ∈ S
6:             do if statement l is reachable from s
7:                    then m1 ← stored location of s
8:                         m2 ← loaded location of l
9:                         if m1 and m2 may alias
10:                           then q ← stored value of s
11:                                Args ← Args ∪ {q}
12:        p_old ← defined value of l
13:        p_new ← INSERT-RHO(Args, p_old)
```

14: **for** each use of p_{old}
15: **do** replace p_{old} with p_{new}

Intuitively, the ϱ-function insertion algorithm operates as follows. First, the algorithm identifies all pointer load and store operations. Then, for each pointer load operation it identified, the algorithm determines a set of stored pointer values such that (1) there is a control-flow path from each corresponding store operation to the load operation and (2) the memory locations stored by each store operation may alias the memory location loaded by the load operation. The procedure INSERT-RHO creates and inserts into the program a new ϱ-function at the location following the load operation, which indicates that the loaded value may be equal to the computed set of stored pointer values. Finally, each use of the original loaded value is replaced by a use of the value defined by the newly created ϱ-function.

4.2 The Required Checks and Metadata

After inserting code for modeling memory deallocation and pointer stores as explicit assignments, MemSafe then inserts the runtime checks and metadata necessary for enforcing memory safety. This section describes the pointer- and object-based checks and metadata that MemSafe requires.

Figure 4.2 depicts MemSafe's unique combination of object- and pointer-based metadata. In contrast to the enforcement methods shown in Figure 3.1, MemSafe utilizes this hybrid metadata representation for ensuring both the spatial and temporal memory safety of C programs.

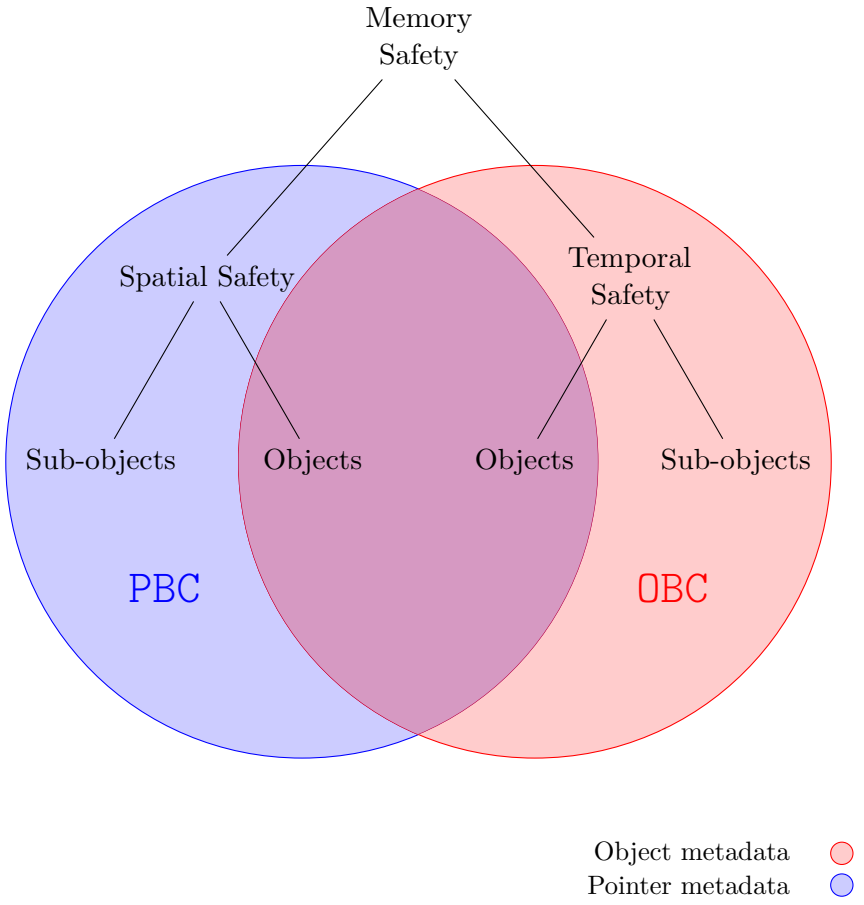


Figure 4.2: Hybrid metadata representation. The use of object and pointer spatial metadata is shown for MemSafe's method of enforcing memory safety.

The differences between Figures 3.1 and 4.2 can be explained as follows. First, because MemSafe models temporal errors as spatial errors, MemSafe avoids the drawbacks associated with the use of conservative garbage collection and the use of additional checks and metadata for enforcing temporal safety (shown in orange in Figure 3.1). Second, since MemSafe's hybrid metadata representation captures the most salient features of object and pointer metadata, MemSafe avoids the drawbacks associated with the use of each in enforcing spatial safety, and gains the ability to reuse this metadata for enforcing temporal safety as well. As shown in Figure 4.2, MemSafe utilizes pointer-based metadata for enforcing complete spatial and partial temporal safety, and MemSafe utilizes object-based metadata for enforcing complete temporal and partial spatial safety. `PBC` and `OBC` are MemSafe's runtime checks that utilize this metadata. These checks, in addition to MemSafe's object- and pointer-based metadata, are discussed below.

4.2.1 Pointer metadata

For the definition of a new pointer p (i.e., a pointer created with `malloc` or the address-of operator), MemSafe creates pointer metadata in the form of a 3-tuple $\langle base, bound, id \rangle_p$ of intermediate values. Together, $base_p$ and $bound_p$ indicate the range $[base, bound)$ of memory p is permitted to access. id_p is a unique key that is assigned to p*'s* referent object, and it is used to associate p with the metadata of its referent (discussed in Section 4.2.3). MemSafe maintains pointer metadata in memory and allocates at runtime an address $addr_p$ from a set of unused addresses A for storing

$\langle base, bound, id \rangle_p$. These values are stored to memory with an explicit dereference operation, represented by $M[addr_p] \leftarrow \langle base, bound, id \rangle_p$, where $M[addr_p]$ holds the value at address $addr_p$ in memory.

In addition to the pointer metadata described above, MemSafe also creates a tuple $\langle addr, id \rangle_p$ of intermediate values. These values are created for the definition of each pointer p in a program (i.e., not just those pointers created with `malloc` or the address-of operator), and are statically named such that there is a known compile-time association with p. Unlike the metadata described above, no dereference is required at runtime for retrieving these values. As previously described, $addr_p$ is the location in memory containing the base and bound addresses that indicate the range of memory p is permitted to access. Finally, in order to allow the reuse of location $addr_p$ (discussed later), a copy of the id associated with p*'s* referent is also maintained with this statically associated tuple.

4.2.2 Pointer bounds check

MemSafe utilizes pointer metadata for performing a Pointer Bounds Check (PBC). MemSafe inserts a `PBC` before each pointer dereference that cannot be verified to be safe statically (see Chapter 5 for optimizations that reason about dereferences that must be safe). `PBC` is the forcibly inlined procedure defined by:

Runtime Check 4.1—Pointer bounds check:

```
1: inline void PBC(ptr, size, addr, id) {
2:   ⟨base, bound, id⟩_ptr ← M[addr]
3:   if ((id != id_ptr) || (ptr < base_ptr) || (ptr + size > bound_ptr)) {
4:     signal_safety_violation();
```

```
5:     }
6:  }
```

In the above runtime check, *ptr*, $base_{ptr}$, and $bound_{ptr}$ are all pointers to the type `unsigned char` and *size* is the size in bytes (as indicated by the `sizeof` operator) of *ptr's* referent.[1] For example, MemSafe utilizes the pointer metadata of a pointer *ptr* to ensure the safety of its dereference at runtime:

Example 4.2—PBC insertion:

```
    PBC(ptr, sizeof(*ptr), addr_ptr, id_ptr);
1:  ...  *ptr ...
2:  ▷ some load or store operation involving ptr
```

In this example, MemSafe will abort the program and report a violation of memory safety (by calling `signal_safety_violation`) if the dereference `*ptr` will access a location outside the range specified by $[base_{ptr}, bound_{ptr})$. Because the `PBC` only utilizes pointer metadata, no costly database lookup is required to retrieve $base_{ptr}$ and $bound_{ptr}$, as $\langle addr, id \rangle_{ptr}$ are uniquely named symbols in the inserted code.

As depicted in Figure 4.2, the `PBC` and MemSafe's pointer-based metadata are capable of not only ensuring complete spatial safety, but also temporal safety with a *single check*. Whenever a pointer p is assigned the value of the *invalid* pointer, its pointer metadata is updated as $M[addr_p] \leftarrow \langle base, bound, id \rangle_{invalid}$, which will always cause the `PBC` to signal a safety violation since the *invalid* pointer refers to an impossible address range.

[1] These pointers are implicitly type-cast to be pointers to type `unsigned char` because `sizeof(unsigned char)` is defined to always equal one byte [46]. This is required for the pointer arithmetic performed in the body of the bounds check to be valid.

As mentioned above, since addresses in A can be reused, a copy of the *id* associated with a pointer p must be included with *p's* pointer metadata. Whenever the metadata associated with the *invalid* pointer is stored to a particular address, this address is marked for potential reuse. Thus, $addr_p$ may be reused for storing the pointer metadata of another pointer if *p's* referent is deallocated. To ensure that $addr_p$ has not been reused, the PBC checks whether the *id* associated with the dereferenced pointer p, is equal to the *id* located at $addr_p$. If it is not, $addr_p$ has been reused for the pointer metadata of another pointer and *p's* referent is temporally invalid.

However, the PBC is insufficient for ensuring complete temporal safety. Since a nested object (e.g., an array of structures or a structure containing and array field) is deallocated using a pointer to its base address, only pointers that refer to the outer object are assigned the value of the *invalid* pointer upon the object's deallocation. The pointer metadata of any potential sub-object references are not updated in this way (see the rules for metadata propagation in Section 4.3). Thus, object metadata is required to associate pointers to *inner* objects with the base and bound address of their corresponding *outer* object. Object metadata is introduced below.

4.2.3 Object metadata

For every object allocation, MemSafe creates and assigns a unique *id* to the object and records a tuple $\langle base, bound \rangle$ for the allocated region in a global object metadata facility. MemSafe removes entries for objects from the metadata facility when they are deallocated. The object metadata facility maps an object's *id* to its base and bound

address and is formally defined by the partial function:

$$\begin{aligned} omd &: I \rightarrow O \\ & id \mapsto \langle base, bound \rangle_{id} \end{aligned}$$

where I is the set of *ids* and O is the set of object metadata. For notational convenience, the function *omd* can also be represented more generally as the relation $\mathcal{R}_{\mathcal{O}}$, where $(id, \langle base, bound \rangle_{id}) \in \mathcal{R}_{\mathcal{O}}$ if the object associated with id is a valid memory object that has yet to be deallocated. A discussion of the implementation of the object metadata facility is deferred until Section 6.2 in order to separate the presentation of MemSafe's method from its prototype implementation.

4.2.4 Object bounds check

MemSafe utilizes object metadata for performing an Object Bounds Check (OBC). MemSafe inserts an `OBC`, in addition to the `PBC` described above, before each pointer dereference that may access a sub-object if the pointer cannot be statically verified to be temporally safe.[2] `OBC` is the forcibly inlined procedure defined by:

Runtime Check 4.2—Object bounds check:

```
1: inline void OBC(ptr, size, id) {
2:   ⟨base, bound⟩_id = omd(id)
3:   if ((ptr < base_id) || (ptr + size > bound_id)) {
4:     signal_safety_violation();
5:   }
6: }
```

In the definition of the above runtime check, *ptr* is a pointer to type `unsigned char`,

[2]Refer to Section 5.1.3.4 for a discussion of how MemSafe utilizes its pointer data-flow representation for computing the set of potential sub-object references.

id is a component of *ptr's* pointer metadata, and *size* is the size in bytes of *ptr's* referent. An OBC is similar in functionality to the PBC. For example, MemSafe utilizes the object metadata of pointer *ptr's* referent, denoted $\langle base, bound \rangle_{id}$, to ensure the safety of its dereference at runtime:

Example 4.3—OBC insertion:

```
1: PBC(ptr, sizeof(*ptr), addr_ptr, id_ptr);
   OBC(ptr, sizeof(*ptr), id_ptr);
2: ... *ptr ...
3: ▷ some load or store operation involving ptr
```

In this example, MemSafe will abort the program and report a violation of memory safety if the dereference *ptr will access a location outside the range specified by $[base_{id}, bound_{id})$. The OBC uses the *id* field of *ptr's* pointer metadata to retrieve the object metadata of its referent from the object metadata facility. Assuming pointer *ptr* refers to a sub-object, the temporal safety of *ptr's* dereference is ensured because, had *ptr's* referent been previously deallocated, its entry would have been unmapped in the object metadata facility $\mathcal{R}_{\mathcal{O}}$, causing $omd(id)$ to fail and MemSafe to signal a safety violation.

As depicted in Figure 4.2, the OBC and MemSafe's object-based metadata are capable of not only ensuring complete temporal safety, but also partial spatial safety with a *single check*. Thus, if the detection of sub-object overflows is not a requirement, the PBC in the above example can be eliminated since the OBC also verifies *ptr* is within bounds of its outer object.

4.3 Propagation of the Required Metadata

Having presented the runtime checks that MemSafe requires for ensuring memory safety, this section describes MemSafe's translations for creating and propagating the required metadata. In doing so, it is assumed that the program has already been transformed such that it includes the syntax extensions for modeling memory deallocation and pointer stores as explicit pointer assignments (see Section 4.1). In the discussion below, the rules for propagating the required metadata are addressed according to the way in which pointers are defined.

4.3.1 Memory allocation

As described previously, MemSafe creates entries in the global object metadata facility as objects are allocated. For automatic memory allocation (i.e., the allocation of stack variables), MemSafe generates a new *id* for the allocated object and maps it to the object's base and bound address in $\mathcal{R}_{\mathcal{O}}$. MemSafe updates the object metadata facility according to the following metadata rule for automatic memory allocation. Numbered lines indicate original code.

Metadata Rule 4.1—Automatic memory allocation:

```
1:  struct { ...  int array[100]; ...  } s;
```

$$\mathcal{R}_{\mathcal{O}} = \mathcal{R}_{\mathcal{O}} \cup \{(id \in I, \langle \texttt{\&s}, \texttt{\&s+}\textbf{\texttt{sizeof}}\texttt{(s)}\rangle)\} \tag{4.1.1}$$

In this example, a structure containing an array field is allocated on the procedure stack. Therefore, MemSafe obtains a new *id* for the allocated object and maps it to

the base and bound address of the allocated region in $\mathcal{R}_{\mathcal{O}}$ (4.1.1).

For dynamic memory allocation (i.e., the allocation of objects on the heap), MemSafe updates $\mathcal{R}_{\mathcal{O}}$ as it does for automatic memory allocation, but it also creates pointer metadata for the pointer returned by `malloc`, since the `malloc` function is responsible for creating a new object as well as a new pointer to the allocated object. If the pointer returned by `malloc` is equal to the `NULL` pointer, the pointer inherits the metadata of the *invalid* pointer. MemSafe creates the required object and pointer metadata for heap-allocated objects according to the following metadata rule for dynamic memory allocation. Numbered lines indicate original code.

Metadata Rule 4.2—Dynamic memory allocation:

```
1:  int *p;
2:  ...
3:  p = (int*) malloc(size);
```

$$\langle addr, id\rangle_p = \langle addr \in A, id \in I\rangle \tag{4.2.1}$$

$$\langle base, bound\rangle_{id_p} = \begin{cases} \langle base, bound\rangle_{id_{invalid}} & \text{if } p = null, \\ \langle \mathtt{p}, \mathtt{p+}size\rangle & \text{otherwise} \end{cases} \tag{4.2.2}$$

$$\mathcal{R}_{\mathcal{O}} = \mathcal{R}_{\mathcal{O}} \cup \left\{\left(id_p, \langle base, bound\rangle_{id_p}\right)\right\} \tag{4.2.3}$$

$$M[addr_p] \leftarrow \langle base_{id_p}, bound_{id_p}, id_p\rangle \tag{4.2.4}$$

In this example, an object of *size* bytes is allocated dynamically by calling `malloc`, and the address returned by `malloc` is assigned to the pointer p. After the program allocates the object, MemSafe obtains an address for holding the pointer metadata of p and obtains a new unique id for the allocated object (4.2.1). If the value returned by `malloc` is equal to `NULL`, the object metadata associated with id_p is set to the base and bound address of the "invalid" region of memory. Otherwise, the metadata

associated with the object is defined such that it refers to the space occupied by the allocated region of memory (4.2.2). Finally, MemSafe associates the object's metadata with id_p in $\mathcal{R}_\mathcal{O}$ (4.2.3) and stores the metadata of p at its associated address (4.2.4).

For static memory allocation (i.e., the allocation of global variables), MemSafe initializes the object metadata facility to include entries for the base and bound address of each allocated region, since the number and size of global variables in known at compile-time.

4.3.2 Memory deallocation

Whenever an object is deallocated, MemSafe removes its entry from $\mathcal{R}_\mathcal{O}$ and sets the pointer metadata of the pointer that refers to the object to be equal to that of the *invalid* pointer. Stack-allocated objects are deallocated when the function in which they are defined exits. Therefore, MemSafe removes their entries from $\mathcal{R}_\mathcal{O}$ just before the end of the procedure. MemSafe updates object and pointer metadata for automatic memory deallocation according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.3—Automatic memory deallocation:

```
1:  void f() {
2:    struct { ...  int array[100]; ...  } s;
3:    int *p;
4:    ...
5:    p = &(s.array[42]);
6:    ...
```

$$\mathcal{R}_\mathcal{O} = \mathcal{R}_\mathcal{O} \setminus \{(id_s, omd\,(id_s))\} \tag{4.3.1}$$

$$M[addr_p] \leftarrow \langle base, bound, id\rangle_{invalid} \tag{4.3.2}$$

```
7:    p = invalid;        ▷ MemSafe models deallocation as an explicit
```

```
8: }                              pointer assignment of 'invalid'
```

In this example, structure s is an automatic variable of function f and contains an array sub-object that is nested within it. In line 5, pointer p is assigned the address of an element of the structure's *array* field, and it is assumed that p may escape to another procedure. Before the procedure exits, MemSafe removes the entry for s from $\mathcal{R}_{\mathcal{O}}$ (4.3.1) using the unique *id* associated with s ("\" denotes set difference). Since p may escape, it is assigned the value of the *invalid* pointer in line 7, and its pointer metadata is updated to refer to the metadata associated with *invalid* (4.3.2).

Heap-allocated objects are deallocated dynamically with the `free` function. Similar to the above rule for automatic memory deallocation, MemSafe updates object and pointer metadata for dynamic memory deallocation according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.4—Dynamic memory deallocation:

```
1: int *p;
2: ...
```

$$\mathcal{R}_{\mathcal{O}} = \mathcal{R}_{\mathcal{O}} \setminus \{(id_p, omd\,(id_p))\} \tag{4.4.1}$$

$$M[addr_p] \leftarrow \langle base, bound, id\rangle_{invalid} \tag{4.4.2}$$

```
3: free(p);
4: p = invalid;          ▷ MemSafe models deallocation as an explicit
                           pointer assignment of 'invalid'
```

In this example, a pointer p to a heap-allocated object is used to deallocate its referent dynamically by the program calling the `free` function. Before the call to `free` in line 3, MemSafe removes the entry for the deallocated object from $\mathcal{R}_{\mathcal{O}}$ with id_p (4.4.1) and sets the pointer metadata of p to be equal to that of the *invalid* pointer (4.4.2).

Pointer p is assigned the value of *invalid* in line 4.

If id_p had been previously unmapped in the object metadata facility (indicating that *p's* referent was already deallocated before the call to `free`), the lookup operation represented by $omd\,(id_p)$ would fail. In this case, MemSafe would signal a temporal safety violation to indicate the multiple deallocation attempt.

4.3.3 Address-of operator

Like dynamic memory allocation, the address-of operator (`&`) creates a pointer to a new location. Therefore, having already updated $\mathcal{R}_{\mathcal{O}}$ for an object's allocation, MemSafe creates pointer metadata for pointers to the object. MemSafe sets the pointer metadata of a pointer defined in terms of the address-of operator according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.5—Address-of operator:

```
1: struct { ...  int array[100]; ...  } s;
2: int *p;
3: ...
4: p = &(s.array[42]);
```

$$\langle addr, id\rangle_p = \langle addr \in A, id_s\rangle \tag{4.5.1}$$

$$M[addr_p] \leftarrow \langle \texttt{\&s.array[0]}, \texttt{sizeof(s.array)}, id_p\rangle \tag{4.5.2}$$

In this example, as in previous examples, a pointer p is assigned the address of an element of the *array* field of structure s. Because the program creates a new pointer, MemSafe obtains a new address for storing the pointer metadata of p (4.5.1). MemSafe then creates and stores pointer metadata for p to indicate that it refers to the base and bound address of the *array* field of s (4.5.2).

This example also demonstrates MemSafe's ability to detect sub-object overflows. Although p refers to a location within object s (indeed, p inherits the *id* of s), *p's* base and bound address are associated with the *array* field of s.

4.3.4 Pointer copies and arithmetic

Pointers defined as simple pointer copies or in terms of pointer arithmetic (e.g., array and structure indexing) inherit the pointer metadata of the original pointer.[3] MemSafe sets the pointer metadata of pointers defined by simple assignments according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.6—Pointer copies and arithmetic:

```
1: int x, *p0, *p1;
2: ...
3: p1 = p0 + x;
```

$$\langle addr, id \rangle_{p_1} = \langle addr, id \rangle_{p_0} \tag{4.6.1}$$

In this example, since pointer p_1 is defined in terms of pointer arithmetic, it simply inherits the pointer metadata associated with pointer p_0 (4.6.1).

4.3.5 ϱ-functions

Since the value produced by a ϱ-function is not known statically, MemSafe must "disambiguate" it for the returned pointer to inherit the correct metadata. Thus, MemSafe requires an additional metadata facility. Like the object metadata facility,

[3]SSA ϕ-functions that involve pointer values are no different than ordinary pointer copies. For example, `p2 = φ(p0, p1)` copies p_0 to p_2 at the end of the basic block defining p_0 and copies p_1 to p_2 at the end of the basic block defining p_1.

the pointer metadata facility maps the address of an in-memory pointer to its pointer metadata and is defined by the partial function:

$$\begin{aligned} pmd : A &\rightarrow P \\ ptr &\mapsto \langle addr, id \rangle_{*\texttt{ptr}} \end{aligned}$$

where A is the set of addresses, and P is the set of pointer metadata. For notational convenience, the function pmd can also be represented more generally as the relation $\mathcal{R}_{\mathcal{P}}$, where $(ptr, \langle addr, id \rangle_{*\texttt{ptr}}) \in \mathcal{R}_{\mathcal{P}}$.

For pointer loads, MemSafe creates a new definition for the loaded value and assigns it the result of a ϱ-function, which indicates the set of values to which the loaded value may potentially be equal. For a pointer ptr whose pointed-to location is loaded in defining another pointer p, MemSafe retrieves from the pointer metadata facility the required pointer metadata for p with the lookup operation $pmd(ptr)$. MemSafe performs this operation according to the following metadata rule for pointer loads. Numbered lines indicate original code.

Metadata Rule 4.7—Pointer loads:

```
1:  int **ptr1, *p0, *p1, ...;
2:  ...
3:  p0 = *ptr1;                  ▷ MemSafe models in-memory data-flow with
4:  p1 = ϱ(a0, b0, ...);           the ϱ-function
```

$$\langle addr, id \rangle_{p_1} = pmd(ptr_1) \tag{4.7.1}$$

In this example, an in-memory pointer is loaded and assigned to pointer p_0. MemSafe then creates a new pointer p_1 and assigns it the result of a ϱ-function indicating the values the in-memory pointer may potentially equal. The pointer metadata for p_1 is retrieved from the pointer metadata facility with the $pmd(ptr_1)$ lookup operation

(4.7.1), and all uses of p_0 are replaced with uses of p_1.

For each argument of the ϱ-function (including the *invalid* pointer), MemSafe saves their pointer metadata in $\mathcal{R}_\mathcal{P}$ at the locations each pointer is stored to memory. MemSafe updates the pointer metadata facility for pointer stores according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.8—Pointer stores:

```
1: int **ptr2, *a0;
2: ...
3: *ptr2 = a0;      ▷ ptr2 may alias ptr1 from above
```

$$\mathcal{R}_\mathcal{P} = (\mathcal{R}_\mathcal{P} \setminus \{(ptr_2, pmd(ptr_2))\}) \cup \{(ptr_2, \langle addr, id \rangle_{a_0})\} \tag{4.8.1}$$

In this example, pointer ptr_2 is assumed to potentially alias with pointer ptr_1 from the previous example. Thus, pointer a_0 appears in the ϱ-function defined above for pointer p_1 because of the pointer store in line 3. Here, MemSafe maps pointer ptr_2 to the pointer metadata of a_0 in $\mathcal{R}_\mathcal{P}$ (4.8.1). If ptr_1 happens to be equal to ptr_2, the pointer metadata of a_0 would be retrieved in the previous example.

4.3.6 `NULL` and manufactured pointers

Pointer type-casts and unions do not require any additional metadata propagation. The new pointer simply inherits the pointer metadata of the original pointer, as in the rule for pointer copies and arithmetic. However, pointers defined as `NULL` or as a cast from a non-pointer type must inherit the base and bound of the *invalid* pointer. Although this may result in false positives, they have been observed to be rare occurrences in practice. For reading and writing to memory-mapped I/O locations,

MemSafe requires a target's backend to specify the base and bound address of all valid address ranges. MemSafe propagates the metadata of the *invalid* pointer according to the following rule for `NULL` and manufactured pointers. Numbered lines indicate original code.

Metadata Rule 4.9—`NULL` and manufactured pointers:

```
1:  int *p;
2:  ...
3:  p = (int*) 42;
```

$$\langle addr, id \rangle_p = \langle addr, id \rangle_{invalid} \tag{4.9.1}$$

In this example, pointer p is defined as a type-cast from the integer 42. Thus, MemSafe defines the pointer metadata for p to be equal to that of the *invalid* pointer (4.9.1). The result would have been the same if p had been assigned the value of `NULL`.

4.3.7 Function arguments and return values

MemSafe requires an additional metadata facility in order to propagate pointer metadata for pointers passed as arguments to functions or returned from functions. Let *callee* values refer to formal pointer arguments and pointer values that are returned from functions. Similarly, let *caller* values refer to actual pointer arguments and local pointer values to be returned from functions. The function metadata facility maps a *callee* value to the pointer metadata of its corresponding *caller* value and is defined by the partial function:

$$\begin{aligned} fmd &: C \rightarrow P \\ callee &\mapsto \langle addr, id \rangle_{caller} \end{aligned}$$

where C is the set of caller values, P is the set of pointer metadata, and *callee* is a tuple $\langle \texttt{\&f}, i \rangle$ indicating the i^{th} pointer associated with function *f*. Pointers are statically assigned an index i based on their usage: the return value of a function is assigned index zero, and the pointer arguments of a function are assigned an index equal to their offset in the function's argument list, beginning at one. For notational convenience, the function *fmd* can also be represented more generally as the relation $\mathcal{R}_{\mathcal{F}}$, where $(callee, \langle addr, id \rangle_{caller}) \in \mathcal{R}_{\mathcal{F}}$.

For function calls, MemSafe creates an entry in the function metadata facility for pointer arguments passed to the function. Similarly, MemSafe defines the pointer metadata of a pointer returned from the function call by performing a lookup operation of $\mathcal{R}_{\mathcal{F}}$. MemSafe updates and defines pointer metadata for function calls according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.10—Function calls:

```
1: int *p0, *p1;
2: ...
```

$$\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 1 \rangle, fmd(\langle \texttt{\&f}, 1 \rangle))\}) \cup \{(\langle \texttt{\&f}, 1 \rangle, \langle addr, id \rangle_{p_0})\} \tag{4.10.1}$$

```
3: p1 = f(p0);
```

$$\langle addr, id \rangle_{p_1} = fmd(\langle \texttt{\&f}, 0 \rangle) \tag{4.10.2}$$

In this example, a pointer p_0 is passed as an argument to function f and pointer p_1 is assigned the returned value. The return value of f is statically associated with the index "0," and its single pointer argument is given an index of "1." Thus, before the function call, the pointer metadata of p_0 is associated with the tuple $\langle \texttt{\&f}, 1 \rangle$ in $\mathcal{R}_{\mathcal{F}}$ (4.10.1). That is, a key represented by the address of f and the integer "1" is mapped

to the pointer metadata of p_0. Similarly, after the call returns, the pointer metadata for p_1 is retrieved from $\mathcal{R}_{\mathcal{F}}$ with the tuple $\langle \texttt{\&f}, 0 \rangle$ (4.10.2).

For the declaration of a function with pointer arguments, MemSafe retrieves the pointer metadata for each incoming pointer by performing a lookup operation of $\mathcal{R}_{\mathcal{F}}$. Similarly, if a function returns a pointer value, MemSafe creates an entry in the function metadata facility for its pointer metadata just before the function returns. MemSafe updates and defines pointer metadata for function declarations according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule 4.11—Function declarations:

```
1:  int* f(int *q) {
2:    int *r;
3:    ...
```

$$\langle addr, id \rangle_q = fmd(\langle \texttt{\&f}, 1 \rangle) \tag{4.11.1}$$

```
2:    ...
```

$$\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 0 \rangle, fmd(\langle \texttt{\&f}, 0 \rangle))\}) \cup \{(\langle \texttt{\&f}, 0 \rangle, \langle addr, id \rangle_r)\} \tag{4.11.2}$$

```
3:    return r;
4:  }
```

In this example, pointer q is a formal argument of function f, and pointer r is returned at the end of the procedure. Since q is declared to be the first pointer in the function's argument list, MemSafe retrieves the pointer metadata for q from $\mathcal{R}_{\mathcal{F}}$ with the tuple $\langle \texttt{\&f}, 1 \rangle$ at the beginning of the procedure (4.11.1). Similarly, since MemSafe statically assigns pointer return values the index "0," the pointer metadata of r is associated with the tuple $\langle \texttt{\&f}, 0 \rangle$ in $\mathcal{R}_{\mathcal{F}}$ before the procedure exits (4.11.2).

MemSafe's approach for propagating metadata for pointer arguments and return values is quite robust. It is sufficient for interfacing with pre-compiled libraries,

handling variable-argument functions, and passing metadata through poorly-typed function pointers. For complete safety, pre-compiled libraries must have been compiled with MemSafe's safety checks, but a safe application is capable of interfacing with unsafe libraries as well.

4.4 Memory Safety for Multithreaded Programs

Concurrent programming is an increasingly common method for improving application performance. Advantages of concurrency include (1) increased application throughput, since the parallel execution of a concurrent program can increase the total number tasks completed in a given time period and (2) increased application responsiveness, since time spent waiting for input/output operations to complete can be effectively used for another task.

Given the current proliferation of multi-core and multiprocessor CPUs, the multi-threading paradigm has emerged as a widespread—if not the dominant—concurrent programming and execution model. Multithreading refers to the ability of a computer to efficiently execute multiple threads. An execution thread is the smallest unit of processing that is schedulable by an operating system, and it exists within the context of a traditional operating system process. The use of multithreading to exploit task parallelism has two main advantages over multiprocessing, the other primary technique for increasing application throughput. First, context switching between threads in the same process is typically faster than context switching between processes since processes maintain a considerable amount of state information. Multiple threads

existing within the same process share the same state. Second, multiple threads within the same process share the same address space, which allows concurrently-running code to conveniently communicate using shared memory. Processes have separate address spaces and must rely on expensive inter-process communication mechanisms for exchanging data.

However, because of the tight coupling of concurrent threads, the potential exists for race conditions to occur, whereby multiple threads simultaneously attempt to update shared data structures. To prevent such concurrency errors, multithreaded applications must use synchronization primitives that lock shared data structures against concurrent access. A multithreaded piece of code is said to be *thread safe* if it lacks such concurrency errors, which is to say that the piece of code is guaranteed to function correctly during simultaneous execution by multiple threads. Because concurrency errors can be very difficult to reproduce and isolate, thread safety is a major challenge for the multithreaded programming paradigm.

For MemSafe to be an effective method of enforcing memory safety for the increasingly large number of multithreaded applications, MemSafe's inserted safety checks, metadata, and code for propagating the required metadata must be made thread safe. Therefore, since the potential exists for the $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ global metadata facilities to be concurrently operated upon by separate threads of a multithreaded program, a locking mechanism is required so that simultaneous updates to these metadata facilities is prevented.

MemSafe requires read/write locks [37] for controlling access to each global metadata facility. A read/write lock is a synchronization primitive that allows multiple

threads to read from the same shared memory area concurrently, but enforces mutual exclusion for any thread that writes to the shared memory. Therefore, $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ are each able to be simultaneously read by multiple threads, but when a thread attempts to update a particular metadata facility, all other threads wishing to read or write that metadata facility must wait until the write operation is completed.

The POSIX threads (Pthreads) standard [44] defines an implementation of the read/write locks MemSafe requires for managing concurrent accesses of the global metadata facilities. Pthreads is a commonly-used standard for creating and manipulating threads for various Unix-like operating systems, including GNU/Linux and Max OS X, and it defines a set of C programming language types and functions (in the `pthread.h` header file) that, in addition to implementing thread synchronization mechanisms, also provides an API for creating and joining threads.

In the remainder of this section, the checks MemSafe inserts for enforcing memory safety and the rules MemSafe uses for propagating the required object and pointer metadata are updated to ensure thread safety. Note that only the portions of MemSafe's method that requires modification (i.e., the portions that access the $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ global metadata facilities) are repeated in this section. All other components of MemSafe's method remain the same as presented in Sections 4.1–4.3.

4.4.1 Declaration of the required locks

Before describing how the checks and rules for metadata propagation are altered to achieve thread safety, the required locks must first be declared and initialized. The

following code fragment demonstrates the use of the Pthreads API for initializing a read/write lock for each metadata facility at the beginning of a program.

Example 4.4—Declaration of metadata facility locks:

```
1:  #include <pthread.h>
2:  ...
3:  pthread_rwlock_t object_lock, pointer_lock, function_lock;
4:  ...
5:  int main(int argc, char *argv[]) {
6:    if (pthread_rwlock_init(&object_lock,   NULL) |
7:        pthread_rwlock_init(&pointer_lock,  NULL) |
8:        pthread_rwlock_init(&function_lock, NULL) != 0) {
9:          abort("Unable to create locks for metadata facilities");
10:   }
11:   ...
12: }
```

In this example, three global read/write locks are declared in line 3 for the three metadata facilities $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$. The locks are initialized at the beginning of `main` so that they will be available for managing access to the metadata facilities before execution enters the concurrent sections of the code. The program aborts in line 9 if any one of the locks could not be initialized.

4.4.2 Object bounds check

Recall that if a pointer may potentially refer to a sub-object, MemSafe inserts an object bounds check (`OBC`) to enforce the temporal safety of the pointer's dereference, in addition to the pointer bounds check (`PBC`) required for enforcing spatial safety. The `OBC` uses the *id* field of a pointer's pointer metadata to retrieve from the object metadata facility $\mathcal{R}_{\mathcal{O}}$ the object metadata $\langle base, bound \rangle_{id}$ associated with the pointer's referent. The `OBC` then determines whether the dereferenced pointer refers to a location

that is within the base and bound address of its referent, and if it is not, the OBC signals a safety violation.

Since the OBC retrieves metadata with the lookup operation $omd(id)$, if a thread allocates an object and inserts its base and bound address into $\mathcal{R}_{\mathcal{O}}$ at the same moment that another thread is executing an OBC, a race could occur and incorrect or undefined program behavior could result. Therefore the lookup operation is a critical section of execution that must be protected by an appropriate locking mechanism, which is shown below in the following revised definition of the object bounds check. Numbered lines indicate code present in the original definition of OBC.

Runtime Check 4.3—Object bounds check (thread safe):

```
1:  inline void OBC(ptr, size, id) {
      if (pthread_rwlock_rdlock(&object_lock) != 0) {
        abort("Unable to acquire read lock for metadata facility");
      }
2:    ⟨base, bound⟩_id = omd(id)
      pthread_rwlock_unlock(&object_lock);
3:    if ((ptr < base_id) or (ptr+size > bound_id))
4:      signal_safety_violation();
5:  }
```

Before the lookup operation in line 2, MemSafe acquires a read lock for $\mathcal{R}_{\mathcal{O}}$ or aborts the program if it is unable to successfully obtain the lock. After the lookup operation is complete, MemSafe releases the read lock for $\mathcal{R}_{\mathcal{O}}$, and performs the bounds check.

4.4.3 Memory allocation

MemSafe creates entries in the object metadata facility as objects are allocated. Since inserting entries into $\mathcal{R}_{\mathcal{O}}$ changes the state of this global data structure, memory

allocation is a critical section of execution. Therefore, a thread that updates $\mathcal{R}_{\mathcal{O}}$ to include the metadata of a newly allocated object must obtain a write lock before performing the operation.

For automatically allocated stack variables, MemSafe generates a new *id* for the allocated object, and maps it to the base and bound address of the allocated region of memory. MemSafe inserts thread synchronization primitives for automatic memory allocation according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for automatic memory allocation.

Metadata Rule 4.12—Automatic memory allocation (thread safe):

```
1: struct { ...  int array[100]; ...  } s;
   if (pthread_rwlock_wrlock(&object_lock) != 0) {
     abort("Unable to acquire write lock for metadata facility");
   }
```

2: $\mathcal{R}_{\mathcal{O}} = \mathcal{R}_{\mathcal{O}} \cup \{(id \in I, \langle \texttt{\&s}, \texttt{\&s+}\textbf{sizeof}\texttt{(s)}\rangle)\}$

```
   pthread_rwlock_unlock(&object_lock);
```

Before the update operation in line 2, MemSafe acquires a write lock for $\mathcal{R}_{\mathcal{O}}$ or aborts the program if is unable to successfully obtain the lock. After the update operation is complete, MemSafe releases the write lock for $\mathcal{R}_{\mathcal{O}}$.

For dynamically allocated heap variables, MemSafe updates the object metadata facility as it does for automatic allocation, and it also creates pointer metadata for the pointer returned by `malloc` since `malloc` is responsible for creating a new object as well as a new pointer to the allocated object. Thus, the object metadata facility must again be protected with a write lock. MemSafe inserts thread synchronization primitives for

dynamic memory allocation according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for dynamic memory allocation.

Metadata Rule 4.13—Dynamic memory allocation (thread safe):

```
1: int *p;
2: ...
3: p = malloc(size);
```

4: $\langle addr, id \rangle_p = \langle addr \in A, id \in I \rangle$

5: $\langle base, bound \rangle_{id_p} = \begin{cases} \langle base, bound \rangle_{id_{invalid}} & \text{if } p = null, \\ \langle \text{p}, \text{p+}size \rangle & \text{otherwise} \end{cases}$

```
if (pthread_rwlock_wrlock(&object_lock) != 0) {
  abort("Unable to acquire write lock for metadata facility");
}
```

6: $\mathcal{R}_{\mathcal{O}} = \mathcal{R}_{\mathcal{O}} \cup \left\{ \left(id_p, \langle base, bound \rangle_{id_p} \right) \right\}$

7: $M[addr_p] \leftarrow \langle base_{id_p}, bound_{id_p}, id_p \rangle$

```
pthread_rwlock_unlock(&object_lock);
```

Before the update operation in line 6, MemSafe acquires a write lock for $\mathcal{R}_{\mathcal{O}}$ or aborts the program if it is unable to successfully obtain the lock. MemSafe releases the write lock for $\mathcal{R}_{\mathcal{O}}$ after the update operation is complete. Note that unlike the creation of object metadata, the creation of pointer metadata is local to a thread and, therefore, does not require the use of thread synchronization primitives.

For statically allocated global variables, their number and size are known at compile-time. Therefore, MemSafe initializes $\mathcal{R}_{\mathcal{O}}$ to include entries for the base and bound address of each allocated region at the beginning of a program's `main` procedure. The initialization of $\mathcal{R}_{\mathcal{O}}$ does not require thread synchronization since it occurs when

the program is executing sequential code and has not yet spawned additional threads of execution.

4.4.4 Memory deallocation

Whenever an object is deallocated—whether it was allocated automatically on the stack or dynamically on the heap—MemSafe removes its entry from $\mathcal{R}_{\mathcal{O}}$ and sets the pointer metadata of the pointer that refers to the object to be equal to that of the *invalid* pointer. Since the removal of entries from $\mathcal{R}_{\mathcal{O}}$ changes the state of this data structure (as was the case for the insertion of new entries) memory deallocation is also a critical section of execution, and a thread that updates $\mathcal{R}_{\mathcal{O}}$ to remove the metadata of a deallocated object must obtain a write lock before performing the operation.

Stack-allocated objects are deallocated when the function in which they are defined exits. Therefore, MemSafe removes from $\mathcal{R}_{\mathcal{O}}$ the entries for a function's locally allocated objects just before the end of the function. MemSafe inserts thread synchronization primitives for the deallocation of stack-allocated objects according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for automatic memory deallocation.

Metadata Rule 4.14—Automatic memory deallocation (thread safe):

```
1:  void func() {
2:    struct { ...  int array[100]; ...  } s;
3:    int *p;
4:    ...
5:    p = &(s.array[42]);
6:    ...
      if (pthread_rwlock_wrlock(&object_lock) != 0) {
        abort("Unable to acquire write lock for metadata facility");
      }
```

```
7:          R_O = R_O \ {(id_s, omd(id_s))}
8:    M[addr_p] ← ⟨base, bound, id⟩_invalid

      pthread_rwlock_unlock(&object_lock);
9:    p = invalid;          ▷ MemSafe models deallocation as an explicit
10: }                         pointer assignment of 'invalid'
```

Before the lookup and update operations in line 7, MemSafe acquires a write lock for $\mathcal{R}_\mathcal{O}$ or aborts the program if is unable to successfully obtain the lock. After completing the operations, MemSafe releases the write lock for $\mathcal{R}_\mathcal{O}$.

Heap-allocated objects are deallocated by a program calling the `free` function. Therefore, MemSafe removes from $\mathcal{R}_\mathcal{O}$ the entry for the object being deallocated just before a call to `free`. MemSafe inserts thread synchronization primitives for the deallocation of heap-allocated objects according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for dynamic memory deallocation.

Metadata Rule 4.15—Dynamic memory deallocation (thread safe):

```
1:  int *p;
2:  ...
    if (pthread_rwlock_wrlock(&object_lock) != 0) {
      abort("Unable to acquire write lock for metadata facility");
    }
3:        R_O = R_O \ {(id_p, omd(id_p))}
4:  M[addr_p] ← ⟨base, bound, id⟩_invalid

    pthread_rwlock_unlock(&object_lock);
5:  free(p);
6:  p = invalid;          ▷ MemSafe models deallocation as an explicit
                            pointer assignment of 'invalid'
```

Like the rule for stack-allocated objects, MemSafe acquires a write lock for $\mathcal{R}_\mathcal{O}$ before the lookup and update operations in line 3 or aborts the program if is unable to

successfully obtain the lock. After completing the operations, MemSafe releases the write lock for $\mathcal{R}_{\mathcal{O}}$.

Since statically allocated object have a global scope and a lifespan equal to that of the program, they are not deallocated until the program terminates. Thus, an update of $\mathcal{R}_{\mathcal{O}}$ is not required to remove entries for global variables.

4.4.5 ϱ-functions

MemSafe uses the global pointer metadata facility $\mathcal{R}_{\mathcal{P}}$ to "disambiguate" the value produce by ϱ-functions. Since inserting entries into $\mathcal{R}_{\mathcal{P}}$ changes the state of this global data structure, pointer loads and stores are critical sections of execution, and a thread that reads or updates $\mathcal{R}_{\mathcal{P}}$ must obtain an appropriate read/write lock before performing these operations.

For pointer loads, MemSafe creates a new definition for the produced value and assigns it the result of a ϱ-function, which indicates the set of values to which the loaded value may potentially be equal. For a pointer *ptr* whose pointed-to location is loaded in the definition of another pointer p, MemSafe retrieves from the pointer metadata facility the required pointer metadata for p with the lookup operation $pmd(ptr)$. MemSafe inserts thread synchronization primitives for pointer loads according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for pointer loads.

Metadata Rule 4.16—Pointer loads (thread safe):

```
1:  int **ptr1, *p0, *p1, ...;
2:  ...
```

```
3:  p0 = *ptr1;            ▷ MemSafe models in-memory data-flow with
4:  p1 = ϱ(a0, b0, ...);     the ϱ-function
    if (pthread_rwlock_rdlock(&pointer_lock) != 0) {
      abort("Unable to acquire read lock for metadata facility");
    }
```

5: $\langle addr, id \rangle_{p_1} = pmd(ptr_1)$

```
    pthread_rwlock_unlock(&pointer_lock);
```

Before the lookup operation in line 5, MemSafe acquires a read lock for $\mathcal{R}_\mathcal{P}$ or aborts the program if it is unable to successfully obtain the lock. MemSafe releases the read lock for $\mathcal{R}_\mathcal{P}$ after the lookup operation is complete.

For each argument of a ϱ-function, MemSafe updates $\mathcal{R}_\mathcal{P}$ at the location the pointer is stored to memory. Since this operation involves the insertion of a new entry into the pointer metadata facility, a write lock is required for a thread to obtain exclusive access to $\mathcal{R}_\mathcal{P}$. MemSafe inserts thread synchronization primitives for pointer loads according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for pointer stores.

Metadata Rule 4.17—Pointer stores (thread safe):

```
1:  int **ptr2, *a0;
2:  ...
3:  *ptr2 = a0;            ▷ ptr2 may alias ptr1 above
    if (pthread_rwlock_wrlock(&pointer_lock) != 0) {
      abort("Unable to acquire write lock for metadata facility");
    }
```

4: $\mathcal{R}_\mathcal{P} = (\mathcal{R}_\mathcal{P} \setminus \{(ptr_2, pmd(ptr_2))\}) \cup \{(ptr_2, \langle addr, id \rangle_{a_0})\}$

```
    pthread_rwlock_unlock(&pointer_lock);
```

Before the lookup and update operations in line 4, MemSafe acquires a write lock for $\mathcal{R}_\mathcal{P}$ or aborts the program if is unable to successfully obtain the lock. After completing

the operations, MemSafe releases the write lock for $\mathcal{R}_{\mathcal{P}}$.

4.4.6 Function calls

Like the pointer metadata facility is used to disambiguate the value produced by ϱ-functions, MemSafe uses the global function metadata facility $\mathcal{R}_{\mathcal{F}}$ in an analogous role for disambiguating pointers passed as arguments to and returned from functions. Since the state of $\mathcal{R}_{\mathcal{F}}$ is modified when a program makes a call to a function that requires pointer arguments or returns a pointer value, threads must obtain an appropriate read/write lock before performing such an operation.

For function calls that pass a pointer p as an actual argument, MemSafe maps a tuple—composed of the address of the callee and the offset of p in the argument list—to the pointer metadata of p. Since this operation involves the insertion of a new entry into the function metadata facility, a write lock is required for a thread to obtain exclusive access to $\mathcal{R}_{\mathcal{F}}$. Similarly, a read lock is required to obtain the metadata of a returned pointer value. MemSafe inserts thread synchronization primitives for function calls involving pointer arguments or returned pointer values according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for function calls.

Metadata Rule 4.18—Function calls (thread safe):

```
1:  int *p0, *p1;
2:  ...
    if (pthread_rwlock_wrlock(&function_lock) != 0) {
      abort("Unable to acquire write lock for metadata facility");
    }
```

3: $\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 1\rangle, \mathit{fmd}(\langle \texttt{\&f}, 1\rangle))\}) \cup \{(\langle \texttt{\&f}, 1\rangle, \langle \mathit{addr}, \mathit{id}\rangle_{p_0})\}$

```
   pthread_rwlock_unlock(&function_lock);
4: p1 = f(p0);
   if (pthread_rwlock_rdlock(&function_lock) != 0) {
        abort("Unable to acquire read lock for metadata facility");
   }
```

5: $\langle addr, id \rangle_{p_1} = fmd(\langle \texttt{\&f}, 0 \rangle)$

```
   pthread_rwlock_unlock(&function_lock);
```

Before the lookup and update operations in line 3, MemSafe acquires a write lock for $\mathcal{R}_{\mathcal{F}}$ or aborts the program if is unable to successfully obtain the lock. After completing the operations, MemSafe releases the write lock for $\mathcal{R}_{\mathcal{F}}$. Similarly, MemSafe acquires and releases a read lock for $\mathcal{R}_{\mathcal{F}}$ for the lookup operation in line 5.

For the definition of a function f that declares a pointer p as a formal argument at offset one in its argument list, MemSafe retrieves from the function metadata facility the required pointer metadata for p with the lookup operation $fmd(\langle \texttt{\&f}, 1 \rangle)$. Similarly, for functions that return a pointer value, MemSafe updates $\mathcal{R}_{\mathcal{F}}$ with the pointer metadata of the returned value. MemSafe inserts thread synchronization primitives for function declarations having formal pointer arguments or returning a pointer value according to the following revised rule. Numbered lines indicate code present in the original definition of the metadata propagation rule for function declarations.

Metadata Rule 4.19—Function declarations (thread safe):

```
1: int* f(int *q) {
2:   int *r;
3:   ...
     if (pthread_rwlock_rdlock(&function_lock) != 0) {
       abort("Unable to acquire read lock for metadata facility");
     }
```

2: $\langle addr, id \rangle_{q} = fmd(\langle \texttt{\&f}, 1 \rangle)$

```
      pthread_rwlock_unlock(&function_lock);
3:    ...
      if (pthread_rwlock_wrlock(&function_lock) != 0) {
        abort("Unable to acquire write lock for metadata facility");
      }

4:    R_F = (R_F \ {(<&f, 0>, fmd(<&f, 0>))}) ∪ {(<&f, 0>, <addr, id>_r)}

      pthread_rwlock_unlock(&function_lock);
5:    return r;
6:  }
```

Before the lookup operation in line 4, MemSafe acquires a read lock for $\mathcal{R}_{\mathcal{F}}$ or aborts the program if it is unable to successfully obtain the lock. MemSafe releases the read lock for $\mathcal{R}_{\mathcal{F}}$ after the lookup operation is complete. Similarly, MemSafe acquires and releases a write lock for $\mathcal{R}_{\mathcal{F}}$ for the update and lookup operations in line 6.

4.5 Example Application

Having presented MemSafe's unoptimized runtime checks and metadata, it is useful to consider a real-world code example in order to better understand how the metadata propagation code is actually implemented. A small, widely-used piece of code that can serve this purpose is the merge sort algorithm. The merge sort algorithm, first proposed in 1945 by John von Neumann [52, p. 159], is a comparison-based sorting algorithm. Having a worst case performance of $\Theta(n \log n)$, the merge sort algorithm easily scales to large data sets and is typically stable, meaning that most implementations preserve the input order of equal elements in the sorted output. Merge sort is a classical example of a divide-and-conquer algorithm [23], and is becoming an increasingly popular sorting algorithm given its scalability and ease of implementation. For example, merge sort is

```
1:  #include <stdlib.h>
2:  #include <limits.h>
3:
4:  ▷ Merge 2 sorted sequences.  A2 is an array
5:    and p, q, and r are indices numbering
6:    elements of the array such that p ≤ q < r.
7:
8:  void merge(int *A2, int p, int q, int r) {
9:
10:   int n1 = q - p + 1;
11:   int n2 = r - q;
12:
13:   int *L = (int*) malloc((n1+1)*sizeof(int));
14:
15:   for (int i=0; i < n1; i++) {
16:     L[i] = A2[p+i-1];
17:   }
18:
19:   int *R = (int*) malloc((n2+1)*sizeof(int));
20:
21:   for (int i=0; i < n2; i++) {
22:     R[i] = A2[q+i];
23:   }
24:
25:   L[n1] = INT_MAX;
26:   R[n2] = INT_MAX;
27:
28:   int i = 0;
29:   int j = 0;
30:   for (int k=p; k <= r; k++) {
31:     if (L[i] <= R[j]) {
32:       A2[k] = L[i];
33:       i     = i+1;
34:     } else {
35:       A2[k] = R[j];
36:       j     = j+1;
37:     }
38:   }
39:   free(L);
40:   free(R);
41: }
42:
43: ▷ Sort the elements in the subarray A1[p..r].
44:   If p ≥ r, the subarray has at most one
45:   element and is therefore already sorted.
46:
47: void merge_sort(int* A1, int p, int r) {
48:   if (p < r) {
49:     int q = (p+r)/2;
50:     merge_sort(A1, p, q);
51:     merge_sort(A1, q+1, r);
52:     merge(A1, p, q, r);
53:   }
54: }
```

Figure 4.3: Merge sort algorithm. An implementation of the merge sort algorithm [23] is shown for sorting an unordered array of integers.

the standard sorting routine for the Perl, Python, and Java interpreted programming languages. Because the merge sort algorithm is used so frequently and is relatively small in size, it provides a convenient and illustrative example of the application of MemSafe's method.

Figure 4.3 shows the C language source code for a simple implementation [23] of the merge sort algorithm for sorting integer arrays. Intuitively, the algorithm operates by dividing an n-element sequence of items to be sorted into two subsequences, each containing $n/2$ elements. It then recursively sorts the two subsequences. Finally the algorithm merges the two sorted subsequences to produce the final sorted sequence. The recursion is terminated when the subsequence to be sorted contains only one element. When this is the case, there is nothing to be done since a sequence of length

one is, by definition, already sorted.

The primary operation of the merge sort algorithm is the actual merging of the sorted sequences. The `merge` procedure, shown in line 8 of Figure 4.3, takes four arguments as input. A is an array of integers and p, q, and r are indices enumerating elements of A such that $p \leq q < r$. The procedure assumes that the sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are in sorted order. It then copies the sub-arrays to temporary storage (the L and R dynamically allocated arrays) and merges the sub-arrays to form a single sorted sub-array that replaces $A[p \ldots r]$.

The `merge` procedure is used as a subroutine of the `merge_sort` procedure defined in line 47, which takes three arguments as input. A is the sub-array to be sorted and p and r are array indices. If $p \geq r$, the sub-array has at most one element and is, therefore, already sorted. If $p < r$, the algorithm computes an array index q that partitions $A[p \ldots r]$ into two sub-arrays. The sub-array $A[p \ldots q]$ contains $\lceil n/2 \rceil$ elements and $A[q+1 \ldots r]$ contains $\lfloor n/2 \rfloor$ elements. Thus, to sort an entire sequence A using merge sort, the algorithm is initialized as `merge_sort(A, 1, n)` where n is the length of array A.

Figure 4.4 shows the source code for the two procedures used in the merge sort algorithm after (1) conversion to the SSA form and (2) insertion of MemSafe's syntax extensions for modeling memory deallocation and pointer stores. Note that the `for` loops from the original source code are implemented here as equivalent `do while` structures to show the inserted ϕ-functions. Complex expressions are also shown separated into elementary operations, and it is assumed that all variables are SSA temporary variables except L and R which require storage in memory. Because the

```
1:  #include <stdlib.h>
2:  #include <limits.h>
3:
4:  ▷ Merge 2 sorted sequences.  A2 is an array
5:    and p, q, and r are indices numbering
6:    elements of the array such that p ≤ q < r.
7:
8:  void merge(int *A2, int p, int q, int r) {
9:
10:   int n1 = q - p + 1;
11:   int n2 = r - q;
12:
13:   int *L = (int*) malloc((n1+1)*sizeof(int));
14:
15:   int i9;
16:   do {
17:     int i0 = φ(0, i9);
18:     if (i0 < n1) {
19:       int *tmp2 = A2+p+i0-1;
20:       int *tmp4 = L+i0;
21:       *tmp4 = *tmp2;
22:     } else {
23:       break;
24:     }
25:     i9 = i0+1;
26:   } while (1);
27:
28:   int *R = (int*) malloc((n2+1)*sizeof(int));
29:
30:   int i2;
31:   do {
32:     int i1 = φ(0, i2);
33:     if (i1 < n2) {
34:       int *tmp7 = A2+q+i1;
35:       int *tmp9 = R+i1;
36:       *tmp9 = *tmp7;
37:     } else {
38:       break;
39:     }
40:     i2 = i1+1;
41:   } while (1);
42:
43:   int *tmp12 = L+n1;
44:   int *tmp14 = R+n2;
45:   *tmp12 = INT_MAX;
46:   *tmp14 = INT_MAX;
47:
48:   int i4, j2, k1;
49:   do {
50:     int i3 = φ(0, i4);
51:     int j0 = φ(0, j2);
52:     int k0 = φ(p, k1);
53:     if (k0 <= r) {
54:       int *tmp16 = L+i2;
55:       int *tmp18 = R+j0;
56:       if (*tmp16 <= *tmp18) {
57:         int *tmp20 = L+i2;
58:         int *tmp22 = A2+k0;
59:         *tmp22 = *tmp20;
60:         int i5 = i2+1;
61:       } else {
62:         int *tmp25 = R+j0;
63:         int *tmp27 = A2+k0;
64:         *tmp27 = *tmp25;
65:         int j1 = j0+1;
66:       }
67:     } else {
68:       break;
69:     }
70:     i4 = φ(i5, i2);
71:     j2 = φ(j1, j0);
72:     k1 = k0+1;
73:   } while (1);
74:
75:   free(L);
76:   free(R);
      L = invalid;
      R = invalid;
77: }
78:
79: ▷ Sort the elements in the subarray A1[p..r].
80:   If p ≥ r, the subarray has at most one
81:   element and is therefore already sorted.
82:
83: void merge_sort(int* A1, int p, int r) {
84:   if (p < r) {
85:     int q = (p+r)/2;
86:     merge_sort(A1, p, q);
87:     merge_sort(A1, q+1, r);
88:     merge(A1, p, q, r);
89:   }
90: }
```

Figure 4.4: Merge sort in SSA form. The implementation of the merge sort algorithm presented in Figure 4.3 is shown here in SSA form [26] with MemSafe's extensions for modeling memory deallocation and pointer stores.

program dynamically deallocates the L and R arrays at the end of the `merge` procedure, MemSafe inserts the assignments of the *invalid* pointer after lines 75 and 76. However, since there are no pointer store or load operations in this example, MemSafe does not insert any ϱ-functions.

Figure 4.5 shows an implementation of MemSafe's unoptimized checks and metadata for a portion of the merge sort algorithm shown in Figure 4.4. Since MemSafe's code insertions for the entire algorithm would be significantly long and complex, only the first 25 lines of the algorithm (in SSA form) are shown with the required checks and metadata for ensuring memory safety. Although this fragment of code is only a portion of the complete algorithm, it includes the allocation of the L array and the copying of the first sorted subsequence of A_2 into L.

Before line 4, MemSafe creates type definitions to represent both pointer and object metadata. The type `pmdb_t` is a structure for representing the 3-tuple pointer metadata $\langle base, bound, id \rangle$, and the type `pmda_t` is a structure for representing pointer metadata of the form $\langle addr, id \rangle$. Finally, the type `omd_t` is a structure that represents the tuple $\langle base, bound \rangle$ of object metadata.

Since pointer A_2 is an argument of the `merge` procedure, it's pointer metadata $\langle addr, id \rangle_{A_2}$ must be defined according to Metadata Rule 4.11, which is used for handling function declarations. Pointer metadata for A_2 is retrieved from the function metadata facility with the `fmd_get` procedure.

After the allocation of the L array in line 13, MemSafe creates additional metadata. Since storage for L is allocated dynamically, MemSafe defines both pointer and object metadata according to Metadata Rule 4.2, which is used for handling dynamic memory

```
   ▷ Type for ⟨base, bound, id⟩ pointer metadata
   typedef struct {
     unsigned char *base, *bound; int id;
   } pmdb_t;

   ▷ Type for ⟨addr, id⟩ pointer metadata
   typedef struct {
     pmdb_t *addr; int id;
   } pmda_t;

   ▷ Type for ⟨base, bound⟩ object metadata
   typedef struct {
     unsigned char *base, *bound;
   } omd_t;

 4: ▷ Merge 2 sorted sequences.  A2 is an array
 5:    and p, q, and r are indices numbering
 6:    elements of the array such that p ≤ q < r.
 7:
 8: void merge(int *A2, int p, int q, int r) {

     ▷ Retrieve pointer metadata for A2
       according to Metadata Rule 4.11.1
     pmda_t pmda_A2 = fmd_get(&merge, 1);

     ...
13:  int *L = (int*) malloc((n1+1)*sizeof(int));

     ▷ Define pointer metadata for L
       according to Metadata Rule 4.2.1
     pmda_t pmda_L = {
       (pmdb_t*) malloc(sizeof(pmdb_t),
       getUniqueId()
     };

     ▷ Define object metadata for L
       according to Metadata Rule 4.2.2
     omd_t *omd_L = (omd_t*)malloc(sizeof(omd_t));
     if (L == NULL) {
       *omd_L = *omd_invalid
     } else {
       *omd_L = (omd_t) {
         L, L+((n1+1)*sizeof(int))
       };
     }

     ▷ Map object metadata of L in the object
       metadata facility according to Metadata
       Rule 4.2.3
     omd_set(pmda_L.id, omd_L);

     ▷ Store pointer metadata of L to memory
       according to Metadata Rule 4.2.4
     pmdb_t pmdb_L = {
       omd_L->base, omd_L->bound, pmda_L.id
     };
     *(pmda_L.addr) = pmdb_L
14:
15:  int i9;
16:  do {
17:    int i0 = φ(0, i9);
18:    if (i0 < n1) {
19:      int *tmp2 = A2+p+i0-1;

         ▷ Define pointer metadata for tmp2
           according to Metadata Rule 4.6.1
         pmda_t pmda_tmp2 = pmda_A2;

20:      int *tmp4 = L+i0;

         ▷ Define pointer metadata for tmp4
           according to Metadata Rule 4.6.1
         pmda_t pmda_tmp4 = pmda_L;

         ▷ Perform PBC for dereference of tmp4
         PBC(tmp4, sizeof(int),
             pmda_tmp4.addr, pmda_tmp4.id);

         ▷ Perform PBC for dereference of tmp2
         PBC(tmp4, sizeof(int),
             pmda_tmp2.addr, pmda_tmp2.id);

21:      *tmp4 = *tmp2;
22:    } else {
23:      break;
24:    }
25:    i9 = i0+1;
26:  } while (1);
     ...
```

Figure 4.5: Merge sort fragment with metadata and checks. A portion of the merge sort algorithm presented in Figure 4.4 in SSA form [26] is shown here with the required checks and metadata MemSafe inserts to ensure memory safety.

allocation. First, MemSafe inserts a call to `malloc` to create an address $addr_L$ for storing the pointer metadata associated with L, and MemSafe creates the unique key id_L for the allocated object with the `getUniqueId` procedure. Second, MemSafe defines the object metadata $\langle base, bound \rangle_{id_L}$ associated with the allocated region of memory. If the pointer returned by `malloc` is equal to `NULL`, the region's object metadata is set to that of the "invalid" region. Otherwise, the region's object metadata is defined to be the base and bound of the allocated object. Third, MemSafe maps the allocated object's key id_L to it's metadata $\langle base, bound \rangle_{id_L}$ in the object metadata facility with the `omd_set` procedure. Finally, MemSafe defines the pointer metadata of the returned pointer $\langle base, bound, id \rangle_L$ and stores it to memory at location $addr_L$.

Since pointers tmp_2 and tmp_4 are defined in lines 19 and 20 in terms of pointer arithmetic, MemSafe creates their pointer metadata $\langle addr, id \rangle_{tmp_2}$ and $\langle addr, id \rangle_{tmp_4}$ according to Metadata Rule 4.6, which is used for pointer copies and arithmetic. This metadata is then utilized to perform the pointer bounds checks before the pointers are dereferenced in line 21. MemSafe inserts metadata and runtime checks for the remaining portion of the merge sort algorithm in a similar way.

Chapter 5

Reducing the Runtime Cost of Enforcing Memory Safety

Because the runtime overhead of MemSafe's basic approach can be prohibitively expensive for use in real systems, this chapter develops several tools and optimizations for reducing the cost associated with the inserted checks for memory safety and the code required for propagating metadata. First, a novel pointer data-flow representation is presented that is made possible by the modeling of both memory deallocation and pointer store operations as explicit pointer assignments. Then, this data-flow representation is used as the foundation of several optimizations that identify and eliminate unneeded runtime checks and code for propagating unused metadata.

5.1 A Data-flow Graph for Pointers

By utilizing the abstractions developed in Section 4.1 for memory deallocation and pointer stores, MemSafe creates a whole-program Data-Flow for Pointers Graph ($DFPG$). The $DFPG$ is a definition-use graph for the flow of all pointer metadata in a program. Since the *invalid* pointer creates a direct pointer assignment for each

deallocated object, and since the ϱ-function creates a pointer assignment for indirect pointer stores, every pointer assignment, whether it is an explicit or implicit assignment, is given a unique name in the SSA representation of the program. Therefore, if the pointer metadata associated with a pointer p is copied to that of another pointer q, there is a directed edge from p to q in the $DFPG$.[1] Similarly, if the pointer metadata of q is loaded from memory, a ϱ-function is inserted that indicates the pointers whose metadata may equal that of q, and these pointers are represented in the $DFPG$ as predecessors of q. In general, since the data-flow of pointers may flow recursively, cycles are possible in the $DFPG$. Recall that since the data-flow associated with pointer loads and stores is represented by the ϱ-function, the definitions and uses associated with these operations are not included in the $DFPG$.

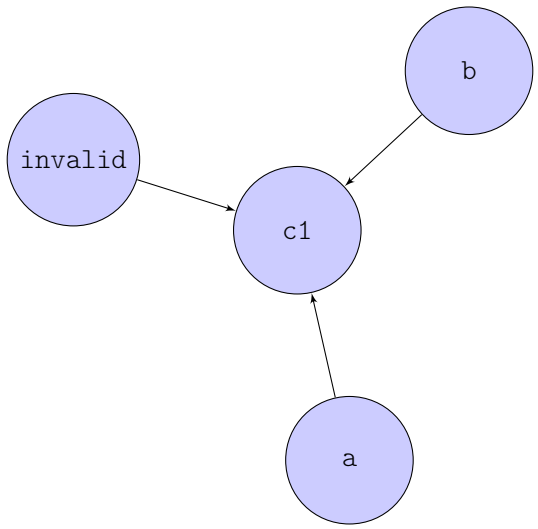
Figure 5.1 shows (a) an example code fragment and (b) its associated $DFPG$. In this code fragment, which was introduced during the discussion of the *invalid* pointer and ϱ-function (see Section 4.1.2), the only pointer definition that is not a pointer load or store operation is that of c_1, which occurs after line 9. Since the definition of c_1 uses the value of three other pointers (a, b, and *invalid*), there is an edge in the $DFPG$ from each of these pointers to c_1. More complicated pointer data-flow is possible, and the $DFPG$ for a real-world program is presented in Section 5.1.4.

The remainder of this section describes the process by which the $DFPG$ is constructed, the issue of $DFPG$ connectivity, and important properties of the $DFPG$ that will be utilized for performing the optimizations that eliminate unneeded runtime

[1]For simplicity, nodes in the $DFPG$ are identified by named pointer values. However, in actuality, $DFPG$ nodes represent the pointer metadata associated with these pointers, and not the pointers themselves.

```
1:  int  *a,  *b,  *c;
2:  int **p, **q;
    ▷ assume p and q may alias
    ...
3:  *q = a;
    ...
4:  if (condition) {
5:     *p = b;
6:  } else {
7:     free(*p);
       *p = invalid;
8:  }
    ...
9:  c0 = *q
    c1 = ϱ(a, b, invalid);
```

(a) Syntax Extensions



(b) $DFPG$

Figure 5.1: DFPG construction. A code fragment with MemSafe's syntax extensions (a) and its corresponding DFPG (b). Numbered lines indicate original code.

checks and metadata propagation. The section ends by discussing the construction of the $DFPG$ for a real-world application.

5.1.1 Construction

The construction of the $DFPG$ is a straightforward process that involves analyzing pointer definitions and uses and reasoning about the propagation of their associated pointer metadata, according to the rules presented in Section 4.3, were it to be inserted for runtime checks. The construction process takes place after (1) a program has been converted to SSA form, (2) pointers assignments for the *invalid* pointer have been inserted for memory deallocation, and (3) ϱ-functions have been inserted to represent the data-flow of pointer stores and loads. The $DFPG = (V, E)$ is a directed graph

consisting of a set of vertices V that represent the pointers in a program (i.e., the $\langle base, bound, id \rangle$ metadata associated with the pointers) and a set of directed edges E that connect pointer uses to pointer definitions. The pseudocode for the $DFPG$ construction algorithm is given below. Pseudocode conventions follow those of Cormen et al. [23].

Algorithm 5.1—Pseudocode for constructing the DFPG:

```
1:  V ← {}
2:  E ← {}
3:  for each instruction i in the program
4:      do if i is a function call
5:            then F ← set of all potential callees of i
6:                 for each potential callee f ∈ F
7:                     do for each actual pointer argument p of i
8:                           do if p ∉ V
9:                                 then V ← V ∪ {p}
10:                          q ← the formal argument of f corresponding to p
11:                          if q ∉ V
12:                             then V ← V ∪ {q}
13:                          E ← E ∪ {(p, q)}
14:                if i defines a pointer p
15:                   then if p ∉ V
16:                           then V ← V ∪ {p}
17:                        for each potential callee f ∈ F
18:                           q ← the return value of f
19:                           if q ∉ V
20:                              then V ← V ∪ {q}
21:                           E ← E ∪ {(q, p)}
22:           else if i defines a pointer p
23:                  then if p ∉ V
24:                          then V ← V ∪ {p}
25:                       for each pointer q that i uses
26:                          do if q ∉ V
27:                                then V ← V ∪ {q}
28:                             E ← E ∪ {(q, p)}
29: DFPG ← GRAPH(V, E)
```

Intuitively, the $DFPG$ construction algorithm operates as follows. For each instruction in the program, if the instruction is a function call, the algorithm first determines

the potential set of functions the call instruction may be calling. This is done by analyzing a conservative call graph that was created using alias analysis to determine the set of potential functions to which each function pointer may refer. The algorithm then connects each actual pointer argument of the call instruction with the formal argument of each potential callee, adding the pointers to the graph in the process if they are not already present. Similarly, pointers that are function return values are connected with the pointer defined by each potential caller of the function. Finally, for instructions that are not function calls, if the instruction defines a pointer value, the pointer is added to the graph and an edge is created to it from each pointer value the definition uses.

5.1.2 Connectivity

While the number of regular vertices in the $DFPG$ (i.e., all vertices excluding those representing ϱ-functions) is fixed for a given program, the number of edges is largely dependent on the precision of the alias analysis that was used for inserting ϱ-functions. For example, assume a naïve alias analysis that responds with *may alias* for every alias query. The analysis would declare that any pointer stored to memory may alias with any pointer loaded from memory. Thus, if this analysis was used for inserting ϱ-functions, there would be an edge from every stored pointer to the ϱ-function representing each loaded pointer. The large number of $DFPG$ edges, in this case, would indicate that a high amount of uncertainty exists in the data-flow of pointer values. This ambiguity is detrimental for the optimizations described in Section

5.2 as the removal of unneeded checks and metadata propagation relies on knowing statically—with certainty—that particular conditions must or must not occur.

MemSafe uses Andersen's alias analysis [4] as the default analysis for ϱ-function insertion (see Section 4.1.2). Andersen's analysis is widely-regarded as the most precise flow- and context- insensitive analysis. However, MemSafe is not tied to one particular analysis, and it can benefit from improved alias analysis precision, although the improved precision would likely come at the expensive of longer compile times.

5.1.3 Properties

Several properties can be derived for the pointers in the $DFPG$ based on the structure of the graph. Examples include the potential for a pointer to be spatially unsafe and the potential for it to be temporally unsafe. Such properties are essential for performing the optimizations that eliminate unneeded runtime checks and code for propagating metadata. The properties required for the optimizations in Section 5.2 are described below.

5.1.3.1 DFPG transpose

The transpose of the $DFPG$ is a new graph $DFPG^T$ on the same set of vertices but with all the edges reversed compared to their original orientation in the $DFPG$. Formally, the transpose of $DFPG = (V, E)$ is $DFPG^T = (V, E^T)$ where $E^T = \{(u, v) : (v, u) \in E\}$. $DFPG^T$ is computed efficiently in $\Theta(|V| + |E|)$ time using adjacency lists. The transpose of the $DFPG$ is useful when considering a reverse data-flow analysis. That is, while the $DFPG$ is a definition-use graph, meaning there

is a directed edge from the definition of a pointer to its uses, the $DFPG^T$ is the corresponding use-definition graph.

5.1.3.2 Undirected DFPG

Similarly, the undirected version of the $DFPG$ is a new graph $DFPG^U = (V', E^U)$ where $V' = V \setminus \{invalid\}$. E^U is formed by (1) replacing all directed edges in E with undirected edges and (2) removing all edges incident with the vertex representing the *invalid* pointer. *invalid* is removed when constructing the $DFPG^U$ to simplify the graph and reveal features that are discussed below. Additionally, since the removal of *invalid* may leave vertices representing ϱ-functions with only a single immediate predecessor, such vertices are also removed to simplify the graph, and edges are inserted to connect the predecessor of each removed ϱ-function with its immediate successors. For example, assume p is defined as a ϱ-function and edges (u, p) and (p, v) are in E^U. If p has only the single immediate predecessor u after the removal of the *invalid* pointer, then p, (u, p), and (p, v) are also removed and replaced with a new edge (u, v). The $DFPG^U$ is also efficiently computed in $\Theta(|V| + |E|)$ time using adjacency lists.

5.1.3.3 Potential referents

A vertex in $DFPG^T$ having no children represents a newly created pointer. That is, pointers in $DFPG^T$ with no children represent pointers returned from the `malloc` function, pointers created with the address-of operator (`&`), or pointers representing statically allocated objects. A newly created pointer, by definition, refers to the object

of its initial assignment. Therefore, the set of potential referents R of a pointer p (represented in the $DFPG^T$ by a vertex v) is given by the set of vertices that are reachable from v and have no children. Formally, the potential referents of v are given by $R_v = \left\{u \in V : v \rightsquigarrow u \wedge \deg^+(u) = 0\right\}$, where $\rightsquigarrow$ denotes reachability and $\deg^+(u)$ denotes the outdegree of vertex u. For any given pointer, its set of potential referents can be computed efficiently in $O(|V| + |E|)$ time using a breadth- or depth-first search.

5.1.3.4 Potential sub-object referents

A pointer created with the address-of operator refers to a sub-object if it has a composite type, and its value is the result an indexing operation through a composite type. The address of a sub-object is computed by indexing through the fields of an aggregate object beginning at the aggregate's base address b. Therefore, a pointer p refers to a sub-object if its definition has the form $p = \&(b + \sum x_i)$ where p and b are pointers to composite types and x_i are offsets that index through the elements of the aggregate pointed to by b. For indexing into a structure type, an offset x_i must be a compile-time constant, but when indexing through an array, x_i is not required to be constant. For any given pointer represented in the $DFPG$, the pointer may potentially refer to a sub-object if there exists a sub-object reference in the pointer's set of potential referents R. Formally, the set of potential sub-object referents $R_v^{(s)} \subseteq R_v$ for a vertex v is given by $R_v^{(s)} = \{r \in R_v : \textsc{is-sub}(r)\}$ where $\textsc{is-sub}(r)$ denotes the proposition that the definition of r matches the indexing pattern $r = \&(b + \sum x_i)$.

5.1.3.5 Temporally unsafe pointers

If the pointer metadata associated with the *invalid* pointer may potentially propagate to another pointer p, then p may be temporally unsafe. Therefore, all vertices in the $DFPG$ reachable from *invalid* represent pointers that may be temporally unsafe. Formally, the set of pointers T that may be temporally unsafe is given by $T = \{v \in V : invalid \rightsquigarrow v\}$. The set of all pointers that may be temporally unsafe can be efficiently computed in $O(|V| + |E|)$ time using a breadth- or depth-first search beginning at the vertex representing the *invalid* pointer.

5.1.3.6 Spatially unsafe pointers

Pointers in the $DFPG$ reachable from `NULL` or manufactured pointers or from pointers defined in terms of pointer arithmetic may be spatially unsafe. Additionally, a pointer that is not physically sub-typed [18] with *each* of its potential referents may also be spatially unsafe since the object to which the pointer refers is unknown statically. A type τ is a physical sub-type of type τ', denoted $\tau \preceq \tau'$, if a value of type τ' may be operated upon as if it had type τ, which is to say that in the memory layout of the two types, the values stored in corresponding locations are compatible for assignment. The vertices representing `NULL` and manufactured pointers and pointers defined in terms of pointer arithmetic can be identified efficiently in $\Theta(|V| + |E|)$ time by performing a breadth- or depth-first search. If these vertices are represented by the set A, then the set S of pointers that may be spatially unsafe is given by $S = \{v \in V : (\exists u \in A)(u \rightsquigarrow v) \vee (\exists r \in R_v)(\tau(v) \not\preceq \tau(r))\}$ where $\tau(v)$ denotes the type

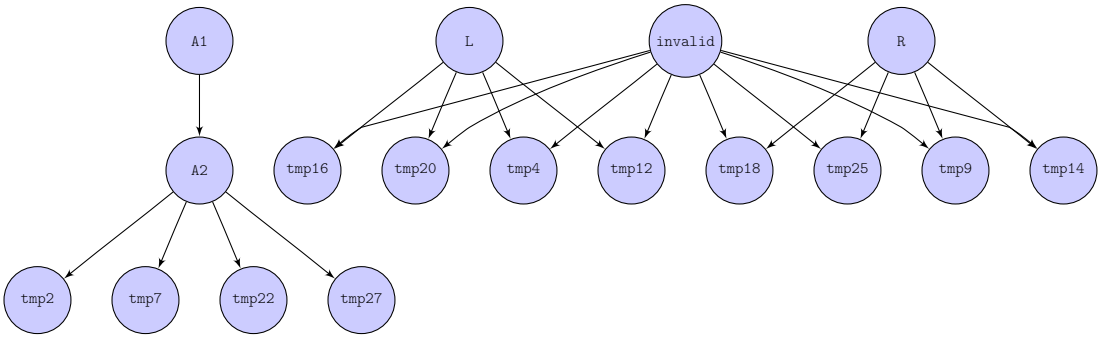
of the pointer represented by vertex v.

5.1.3.7 Pointers that may alias

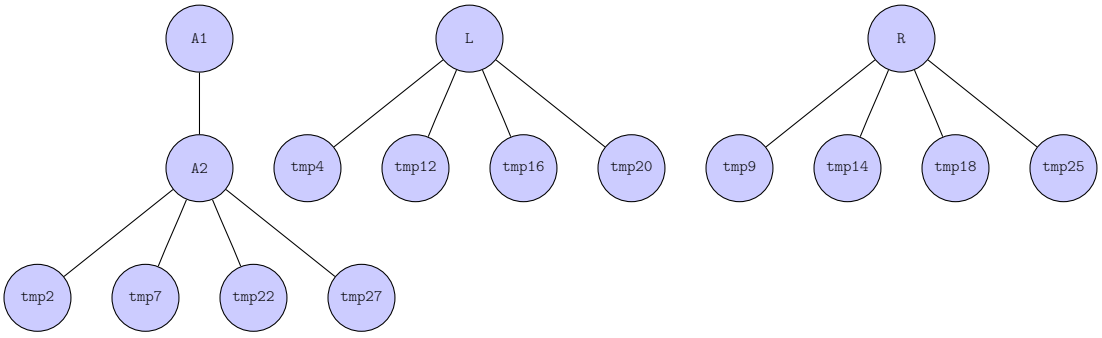
Pointers that may alias are represented in $DFPG^U$ as connected components. Intuitively, since pointers that may alias might refer to the same objects, there is a path between all vertices in $DFPG^U$ that have overlapping sets of potential referents. Thus, pointers that may alias are in the same connected component in $DFPG^U$, and pointers that must not alias are in different connected components. The connected components of an undirected graph can be computed efficiently in $\Theta(|V| + |E|)$ time by performing a breadth- or depth-first search. A search beginning at vertex v will find the entire single connected component containing v (and no additional vertices) in $O(|V| + |E|)$ time. The remaining connected components are found by initiating a similar search beginning at the next unvisited vertex and continuing until all vertices have been visited.

5.1.4 Example application

It is again useful to consider a real-world example to better understand how a program's $DFPG$ is actually constructed. Recall the merge sort procedure described in Section 4.5. Figure 5.2 shows (a) the $DFPG$ of the merge sort algorithm presented in Figure 4.4 and (b) its corresponding $DFPG^U$. Crucially, these data-flow graphs simulate the flow of *pointer metadata*, which was shown for a portion of the algorithm: in Figure 4.5, pointer metadata is given type `pmdb_t`. Note that the nodes in the graph are named according to the pointer with which the metadata is associated.



(a) $DFPG$ for the merge sort algorithm



(b) $DFPG^U$ for the merge sort algorithm

Figure 5.2: Merge Sort DFPG. The $DFPG$ (a) and corresponding $DFPG^U$ (b) are shown for the merge sort algorithm [23]. The $DFPG^U$ shows three connected components.

Because the original L and R arrays are deallocated with calls to `free`, the pointer metadata of each is updated to be equal to that of the *invalid* pointer, indicating that L and R now refer to an "invalid" region of memory. MemSafe uses Andersen's alias analysis to insert ϱ-nodes for every loaded pointer that may alias with the locations used in storing pointers (including stores of the metadata associated with the *invalid* pointer and the loads of pointer metadata performed by the pointer bounds checks, which are not shown in Figure 4.5). Because Andersen's analysis is flow-insensitive, the analysis identifies pointer loads of locations that may alias with stored-to locations regardless of where they occur in the control-flow of the program. Therefore, even though the L and R arrays are not deallocated until the end of the `merge` procedure, *invalid* is propagated to other pointers in the algorithm.[2] Note that since there are no pointer loads involving pointers derived from the original array A, and since A is not deallocated, the pointer metadata of *invalid* does not propagate to A.

The example's $DFPG^U$ in Figure 5.2 is formed by removing the *invalid* pointer, and all edges incident with it, from the $DFPG$ and replacing all directed edges with undirected edges. The graph is also simplified by removing ϱ-functions that have only a single remaining argument (after having removed the *invalid* pointer). The $DFPG^U$ shows three connected components that correspond to the disjoint alias sets associated with the original L, R, and A arrays.

[2]As mentioned in Section 4.1, MemSafe performs a simple reachability analysis to avoid the insertion of ϱ-functions for impossible pointer data-flow. However, that analysis was not used in constructing this example in order to show the propagation of the pointer metadata associated with the *invalid* pointer.

5.2 Optimizations of the Basic Approach

MemSafe utilizes the previously described properties of a program's $DFPG$ to perform several optimizations that reduce the cost of enforcing memory safety at runtime. Since the $DFPG$ blurs the distinction between spatial and temporal errors by encoding memory deallocation and pointer stores as direct pointer assignments, the optimizations described below effect the runtime overhead of achieving both spatial and temporal memory safety.

In the discussion below, the following C source code example will be used to demonstrate the application of MemSafe's optimizations.

Example 5.1—Original Code for Demonstrating Optimizations:

```
1:  struct { ...  int array[100]; ...  } s;
2:  ...
3:  int *p = &(s.array[42]);
4:  int *q = &(s.array[0]);
5:  ...
6:  *p = x;
7:  ...
8:  for (int i=0; i < 42-n; i++) {
9:    q[i] = *p;
10: }
```

In this example, a nested object (a structure containing an array) is allocated on the procedure stack in line 1. In line 3, a pointer p is created that refers to a location within the *array* field of the structure s, and in line 4, another pointer q is created that refers to the beginning of *array*. The element to which p refers (element 42 of *array*) is then assigned some value x in line 6. Finally, in lines 8–10, pointer q is used to iterate over the first part of *array* (elements 0 to $42 - n$) and assign each element

the value to which p refers, which in this case, is the value x.

Although this code fragment is somewhat contrived, it contains several features that are useful for demonstrating the application and functionality of MemSafe's optimizations. These features include (1) multiple dereferences of the same pointer, (2) sub-object pointer accesses, (3) pointer arithmetic involving both constant and non-constant offsets, and (4) array accesses within the body of a loop.

In the remainder of this section, several effective optimizations for eliminating unneeded safety checks and code for the propagation of metadata are described. Throughout the discussion, the above code fragment, instrumented with the code transformations necessary for enforcing spatial and temporal memory safety, will be reproduced in order to demonstrate the application of each optimization.

5.2.1 Dominated dereferences optimization

Multiple dereferences of the same pointer require safety checks only for the dereference that dominates the others. Dominated dereferences do not require checking. Well-known in compiler theory, a dereference d_1 *dominates* [7] another dereference d_2 if every path from the beginning of the program to d_2 includes d_1. Given the naming invariant of the SSA form—that is extended to include assignments of the *invalid* pointer and ϱ-functions (see Section 4.1)—all variables with the same name are required to have the same value. Thus, if the dereference of a pointer is guaranteed to always be executed before another dereference of the same pointer, memory safety checks for the second dereference are redundant, since the second pointer must have the same value as the first.

Within a basic block, a dereference d_1 dominates d_2 if d_1 comes before d_2 in the order of instructions. Across basic blocks, d_1 dominates d_2 if the basic block containing d_1 dominates the basic block containing d_2. The dominators of a basic block n are defined on the control-flow graph of the function f containing n and given by the maximal solution to the following data-flow equations:

$$\begin{aligned} \mathrm{Dom}(n_0) &= n_0 \\ \mathrm{Dom}(n) &= \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p) \right) \bigcup \{n\} \end{aligned}$$

where n_0 is the entry block of *f*. Thus, the dominator of the first basic block in f is itself, and the set of dominators for any other basic block n in f is the intersection of the set of dominators for all predecessors p of n. Since a basic block also dominates itself, n is also in the set of dominators for n.

The concept of dominance is integral to the SSA form, since all definitions must dominate their uses, and the computation of dominators is required for converting a program into SSA form. Although, the direct solution to the above equations is computed in $O(N^2)$ time, where N is the number of basic blocks in function *f*, Lengauer and Tarjan [57] developed an efficient near-linear-time algorithm that relies on properties of the *depth-first spanning tree* [23] of *f's* control-flow graph.

The following code fragment demonstrates the application of the Dominated Dereferences Optimization (DDO) using the running code example introduced above. Below, the source code is shown after conversion to SSA form and after the insertion of all necessary checks and metadata required for enforcing memory safety. Note that the original `for` loop is replaced here with an equivalent `do while` loop to show the

inserted ϕ-functions. Unnumbered lines indicate MemSafe's insertions for performing runtime checks and propagating the required metadata.

Example 5.2—Dominated dereferences optimization:

```
1:  struct { ...  int array[100]; ...  } s;
    R_O = R_O ∪ {(id ∈ I, ⟨&s, &s + sizeof(s)⟩)}
2:  ...
3:  int *p0 = &(s.array[42]);
4:  int *q0 = &(s.array[0]);
    ⟨addr, id⟩_p0 = ⟨addr ∈ A, id_s⟩
     M[addr_p0] ← ⟨&s.array[0], sizeof(s.array), id_p0⟩
    ⟨addr, id⟩_q0 = ⟨addr ∈ A, id_s⟩
     M[addr_q0] ← ⟨&s.array[0], sizeof(s.array), id_q0⟩
5:  ...
    PBC(p0, sizeof(int), addr_p0, id_p0);
    OBC(p0, sizeof(int), id_p0);
6:  *p0 = x;
7:  ...
8:  int i1;
9:  do {
10:   int i0 = φ(0, i1);
11:   if (i0 < 42-n) {
12:     int *q1 = q0 + i0;
        ⟨addr, id⟩_q1 = ⟨addr, id⟩_q0
        ~~PBC(p0, sizeof(int), addr_p0, id_p0);~~     ▷ Dominated dereferences do
        ~~OBC(p0, sizeof(int), id_p0);~~                not require checking
        PBC(q1, sizeof(int), addr_q1, id_q1);
        OBC(q1, sizeof(int), id_q1);
13:     *q1     = *p0;
14:   } else {
15:     break;
16:   }
17:   i1 = i0+1;
18: } while (1);
```

Since the dereference of p_0 in line 6 dominates the dereference of p_0 in line 13, the object and pointer bounds checks before the dereference of p_0 in line 13 are redundant and are eliminated.

5.2.2 Temporally safe dereferences optimization

Dereferences of pointers that are guaranteed to be temporally safe do not require an object bounds check. Recall that since temporal errors are modelled as spatial errors, with the introduction of assignments of the *invalid* pointer, the pointer bounds check is capable of ensuring complete spatial *and* temporal safety for object-level references. However, if a pointer may refer to a sub-object (and may be temporally unsafe), its dereference requires an `OBC` in addition to a `PBC` to ensure temporal safety.

To understand why an `OBC` is needed to ensure the temporal safety of potential sub-object references, consider a pointer p that refers to the base address of a dynamically allocated structure containing a nested sub-object. If the structure is deallocated with `free(p)`, the statement `p = invalid` is inserted into the program in order to propagate the metadata of the *invalid* pointer. MemSafe updates the pointer metadata of p according to the metadata propagation rule for dynamic memory deallocation (see Section 4.3.2):

$$M[addr_p] \leftarrow \langle base, bound, id \rangle_{invalid}$$

However, if before the deallocation of s, there exists a sub-object reference of the form `q = (p)->`*`field`* (which is represented as `q = &((*p).`*`field`*`)`), the pointer metadata for q is defined according to the rule for the address-of operator (see Section 4.3.3) and is represented as

$$\langle addr, id \rangle_q = \langle addr \in A, id \rangle M[addr_q] \quad \leftarrow \langle \texttt{\&s.}\textit{field}, \textbf{sizeof}\texttt{(s.field)}, id \rangle$$

where s is equal to `*p` and id refers to the unique id assigned to s when it was allocated.

Thus, q is assigned a new address for storing metadata, and its base and bound are equal to the "narrowed" sub-range of a particular field of the nested structure. Only if q had been defined as an object-level reference of the form `q = p` would it inherit $addr_p$ and the metadata associated with the *invalid* pointer when p is used to deallocate s. Therefore, a PBC is not capable of enforcing temporal safety for sub-object references and an OBC is instead required.

However, the OBC can be eliminated for pointers that are guaranteed to be temporally safe. Pointers that may be temporally unsafe are determined by traversing the $DFPG$ (see Section 5.1.3.5). In addition, pointers that may be temporally unsafe but cannot be sub-object references (see Section 5.1.3.4) are, by the above discussion, capable of being checked with a PBC. Since an OBC requires that a lookup operation be performed of the object metadata facility $\mathcal{R}_{\mathcal{O}}$ (and any required blocking due to thread synchronization primitives), MemSafe preferentially removes an OBC where a PBC is sufficient for enforcing memory safety.

The following code fragment demonstrates the application of the Temporally safe Dereferences Optimization (TDO) using the running code example. Below, the source code is shown after eliminating the redundant checks for the dereference of p_0 in line 12 with the dominated dereferences optimization.

Example 5.3—Temporally safe dereferences optimization:

```
1:  struct { ...  int array[100]; ...  } s;
    R_O = R_O ∪ {(id ∈ I, ⟨&s, &s + sizeof(s)⟩)}
2:  ...
3:  int *p0 = &(s.array[42]);
4:  int *q0 = &(s.array[0]);
    ⟨addr, id⟩_p0 = ⟨addr ∈ A, id_s⟩
     M[addr_p0] ← ⟨&s.array[0], sizeof(s.array), id_p0⟩
```

```
    ⟨addr, id⟩_{q0} = ⟨addr ∈ A, id_s⟩
     M[addr_{q0}] ← ⟨&s.array[0], sizeof(s.array), id_{q0}⟩
5:  ...
    PBC(p0, sizeof(int), addr_{p0}, id_{p0});
    ~~OBC(p0, sizeof(int), id_{p0});~~              ▷ Temporally safe dereferences
6:  *p0 = x;                                          do not require an object check
7:  ...
8:  int i1;
9:  do {
10:   int i0 = φ(0, i1);
11:   if (i0 < 42-n) {
12:     int *q1 = q0 + i0;
        ⟨addr, id⟩_{q1} = ⟨addr, id⟩_{q0}
        PBC(q1, sizeof(int), addr_{q0}, id_{q1});
        ~~OBC(q1, sizeof(int), id_{q1});~~
13:     *q1     = *p0;
14:   } else {
15:     break;
16:   }
17:   i1 = i0+1;
18: } while (1);
```

Although p_0 refers to a sub-object (the *array* field of structure s), p_0 is guaranteed to be temporally safe in this example, and the object bounds checks before the dereference of p_0 in line 6 and the dereference of q_1 in line 13 are eliminated.

5.2.3 Non-incremental dereferences optimization

A pointer may be spatially unsafe if the vertex representing it in the $DFPG$ is reachable from `NULL` or manufactured pointers or pointers defined in terms of pointer arithmetic. Additionally, a pointer is may also spatially unsafe if it is not physically sub-typed with each of its potential referents (see Section 5.1.3.6). All other pointers are guaranteed to be spatially safe. If a spatially safe pointer is also temporally safe (see Section 5.1.3.5), a `PBC` before its dereference is not required and is removed. Recall that the pointer bounds check is capable of ensuring the temporal safety of object

references in addition to the spatial safety of object and sub-object references. Since MemSafe preferentially removes (with TDO) an `OBC` for the dereference of pointer that may be temporally unsafe, but which must not be a sub-object reference, a `PBC` cannot be removed if the dereferenced pointer may be temporally unsafe.

As a refinement of the above discussion, compile-time bounds checking is possible for pointers that are reachable in the $DFPG$ from pointers defined in terms pointer arithmetic if every such path in the $DFPG$ only involves the arithmetic of constant values. Every possible constant-offset address expression for a pointer p can be determined by performing a depth-first search of the $DFPG^T$ beginning at the vertex representing p. If each potential address expression is within bounds of, and physically sub-typed with, the type associated with each potential referent of p, a `PBC` for the dereference of p is not required and is eliminated.

The following code fragment demonstrates the application of the Non-incremental Dereferences Optimization (NDO) using the running code example. Below, the source code is shown after eliminating the object bounds checks for pointers p_0 and q_1 with the temporally safe dereferences optimization.

Example 5.4—Non-incremental Dereferences Optimization:

```
1:  struct { ...  int array[100]; ...  } s;
    R_O = R_O ∪ {(id ∈ I, ⟨&s, &s + sizeof(s)⟩)}
2:  ...
3:  int *p0 = &(s.array[42]);
4:  int *q0 = &(s.array[0]);
    ⟨addr, id⟩_p0 = ⟨addr ∈ A, id_s⟩
     M[addr_p0] ← ⟨&s.array[0], sizeof(s.array), id_p0⟩
    ⟨addr, id⟩_q0 = ⟨addr ∈ A, id_s⟩
     M[addr_q0] ← ⟨&s.array[0], sizeof(s.array), id_q0⟩
```

```
5:  ...
    PBC(p0, sizeof(int), addr_p0 , id_p0 );      ▷ Non-incremental dereferences
6:  *p0 = x;                                       do not require a pointer check
7:  ...
8:  int i1;
9:  do {
10:   int i0 = φ(0, i1);
11:   if (i0 < 42-n) {
12:     int *q1 = q0 + i0;
        ⟨addr, id⟩_q1 = ⟨addr, id⟩_q0
        PBC(q0, sizeof(int), addr_q0 , id_q0 );
13:     *q1     = *p0;
14:   } else {
15:     break;
16:   }
17:   i1 = i0+1;
18: } while (1);
```

Since the definition of pointer p_0 does not involve any pointer arithmetic—it only involves the address-of operator (&) after expanding the complex expression—and p_0 is not equal to the NULL pointer or a manufactured pointer, p_0 is spatially safe and must refer to the address of its assignment in line 3. In addition, since p_0 is also temporally safe, the pointer bounds check before the dereference of p_0 in line 6 is not required and is eliminated.

5.2.4 Monotonically addressed ranges optimization

A pointer whose value is a monotonic function of a loop induction variable is said to refer to a monotonically addressed range of memory [42]. Induction variable increments of the form $i_n = i_{n-1} + 1$ and $i_n = i_{n-1} - 1$ are examples of monotonic functions that are common in real-world applications. In general, if a pointer refers to a monotonically addressed range of memory, and if the value of the loop terminating

condition is loop-invariant, then it is possible to determine an expression for the range of memory to which the pointer refers throughout the execution of the loop.

For example, let p_0 be the initial value of a pointer p that is dereferenced within the body of loop having a monotonically increasing induction variable. The range of memory to which p can refer is given by $[p_0 + f(i_0), p_0 + f(i_n))$, where f is some monotonic function of the induction variable i. The initial and final values of i are represented by i_0 and i_n, respectively, and are determined by the loop starting and terminating conditions. For the sake of discussion, assume that i has been identified with a suitable loop induction variable recognition analysis [7], and that the loop has been subsequently transformed such that i is the single *canonical* induction variable, meaning that i is initialized to zero and incremented by one on every iteration of the loop [58]. The range of memory to which p can refer then simplifies to $[p_0, p_0 + n]$, where n is the terminating condition of the loop.

For the dereference of a pointer referring to a monotonically addressed range of memory, MemSafe removes the pointer's PBC from within the loop body, and instead inserts a Monotonically Addressed Range Check (MARC) in the loop pre-header. If the pointer's dereference also requires an OBC, this check is placed in the loop pre-header as well. MARC is a runtime check defined by the following forcibly inlined procedure.

Runtime Check 5.1—Monotonically addressed range check:

```
1: inline void MARC(ptr, size, addr, id, trip_count) {
2:   ⟨base, bound, id⟩_ptr ← M[addr]
3:   ptr_max = ptr + trip_count;
4:   if ((id != id_ptr) || (ptr < base_ptr) || (ptr_max + size > bound_ptr)) {
5:     signal_safety_violation();
6:   }
7: }
```

For a loop having a canonical induction variable i, MemSafe signals a safety violation during the execution of the loop pre-header if `*ptr + i` will access a location outside the range specified by $[base_{ptr}, bound_{ptr})$ on any iteration of the loop. Since i is canonical, `trip_count` is determined by the loop terminating condition. Additionally, because a `MARC` utilizes pointer metadata, the *id* of the dereferenced pointer must be compared to the *id* located at the metadata's address since the address may have been reused (see Section 4.2.2). Finally, Note that a `MARC` does not eliminate the need to perform an object bounds check for potential sub-object references that may be temporally unsafe. However, if the dereference of a pointer also requires an `OBC`, this check can be placed in the loop pre-header as well.

A basic block p is the *pre-header* [7] of a loop if it is the *immediate dominator* [7] of the loop header h. p is the immediate dominator of h if (1) $p \neq h$, (2) p dominates h, and (3) p does not dominate any other dominator of h. The basic block h is the *header* [7] of a loop if (1) from any basic block in the loop there is a path of directed control-flow edges leading to h, (2) there is a path of directed edges from h to any other basic block in the loop, and (3) there is no edge from any block outside the loop to any block within the loop other than h.

The following code fragment demonstrates the application of the Monotonically addressed Ranges Optimization (MRO) using the running code example. Below, the source code is shown after eliminating the `PBC` before line 6 with the non-incremental dereferences optimization.

Example 5.5—Monotonically Addressed Ranges Optimization:

```
1:  struct { ...  int array[100]; ...  } s;
    R_O = R_O ∪ {(id ∈ I, ⟨&s, &s + sizeof(s)⟩)}
2:  ...
3:  int *p0 = &(s.array[42]);
4:  int *q0 = &(s.array[0]);
    ⟨addr, id⟩_p0 = ⟨addr ∈ A, id_s⟩
     M[addr_p0] ← ⟨&s.array[0], sizeof(s.array), id_p0⟩
    ⟨addr, id⟩_q0 = ⟨addr ∈ A, id_s⟩
     M[addr_q0] ← ⟨&s.array[0], sizeof(s.array), id_q0⟩
5:  ...
6:  *p0 = x;
7:  ...
8:  int i1;
    MARC(q0, sizeof(int), addr_q0, id_q0, 42-n);
9:  do {
10:   int i0 = φ(0, i1);
11:   if (i0 < 42-n) {
12:     int *q1 = q0 + i0;
        ⟨addr, id⟩_q1 = ⟨addr, id⟩_q0
        ~~PBC(q0, sizeof(int), addr_q0, id_q0);~~      ▷ Monotonically addressed
13:     *q1     = *p0;                                   ranges can be checked in
14:   } else {                                           loop pre-header
15:     break;
16:   }
17:   i1 = i0+1;
18: } while (1);
```

Since the definition of q_1 in line 12 occurs within the body of a loop having a loop-invariant terminating condition $(42 - n)$, and q_1 is a function of the canonical loop induction variable i_0, q_1 refers to a monotonically addressed range of memory. Therefore, the PBC for the dereference of q_1 in line 13 is not required, and MemSafe instead inserts a MARC in the loop pre-header.

The monotonically addressed ranges optimization is similar to array bounds check elimination [15, 41, 53, 73, 81] and loop-invariant code motion [7] of compiler theory. Loop-invariant code motion is a compiler optimization that automatically moves loop-invariant code from within a loop to a location outside the loop in order to avoid

unnecessarily repeating computations. However, since the pointer dereferences that are targeted by MRO are not loop-invariant, loop-invariant code motion is unable to hoist a `PBC` within a loop and move the check to the loop's pre-header. The `PBC` must first be converted to a `MARC` by the monotonically addressed ranges optimization before it becomes loop-invariant and capable of being hoisted.

Array bounds check elimination is a well-researched technique whereby the unneeded bounds checks for affine array accesses can be eliminated. However, MRO does not require a memory reference to be an affine array access. Indeed, a dereferenced pointer could itself be an induction variable. In certain cases, though, techniques such as affine conversion [36] can be used to convert pointer accesses of array elements into semantically equivalent array representations. Thereafter, the array access would be subject to bounds check elimination. MRO is more general than array bounds check elimination, but for affine array accesses, bounds check elimination could be used with MemSafe to completely eliminate some of the required runtime checks.

5.2.5 Partitioned metadata optimization

In a multithreaded program, concurrent threads must acquire read/write locks when attempting to update or retrieve metadata from the global metadata facilities $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ (see Section 4.4). In order to avoid some of the runtime cost associated with the required thread synchronization primitives, MemSafe can partition each metadata facility into separate structures and thereby reduce metadata facility access contention. Furthermore, thread synchronization primitives can be removed entirely if concurrent

threads access disjoint data structures after the contentious metadata facilities have been partitioned. The code constituting a "thread" can be conservatively determined automatically through an analysis of a program's call graph.

As discussed in Section 5.1.3.7, connected components in the $DFPG^U$ represent disjoint alias sets. Since pointers that must not alias are guaranteed to never refer to the same objects, the metadata for a pointer in one particular connected component of the $DFPG^U$ will never propagate to a pointer in another connected component. To understand why, consider two pointers p and q. If metadata propagates from p to q, the pointers are required to alias since the metadata associated with each would indicate that both p and q refer to a region of memory bounded by the same base and bound address. By definition, pointers that refer to the same region of memory, or overlapping regions of memory, must alias.

Therefore, since the metadata for pointers in one connected component cannot propagate to the pointers of another connected component (an exception being the metadata for the *invalid* pointer which is discussed below), the global metadata facilities $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ can be partitioned into N disjoint structures, where N is the number of connected components in the $DFPG^U$. Thus, all pointers in a connected component n are assigned the metadata facilities ${\mathcal{R}_{\mathcal{O}}}^{(n)}$, ${\mathcal{R}_{\mathcal{P}}}^{(n)}$, and ${\mathcal{R}_{\mathcal{F}}}^{(n)}$, and all update and retrieval operations involving pointers in the n^{th} connected component make use of the n^{th} set of metadata facilities.

When computing the $DFPG^U$, the *invalid* pointer and edges incident with it are removed from the graph in order to avoid representing an artificial flow of data introduced by MemSafe's analysis. Therefore, after partitioning the metadata facilities,

a separate *invalid* pointer, represented by $invalid^{(n)}$, is created for each connected component. This ensures that the metadata associated with the original *invalid* pointer will not propagate to multiple connected components of the $DFPG^U$.

The following code fragment demonstrates the application of the Partitioned Metadata Optimization (PMO) using the running code example. Below, the source code is shown after converting the PBC within the loop body to a MARC in the loop pre-header by applying the monotonically addressed ranges optimization. Also, since a multithreaded program is assumed for this example, thread synchronization primitives have been added to control access to the metadata facilities.

Example 5.6—Partitioned Metadata Optimization:

```
1:  struct { ...  int array[100]; ...  } s;
    if (pthread_rwlock_wrlock(&object_lock) != 0) {
      abort("Unable to acquire write lock for metadata facility");
    }
    R_O^(n) = R_O^(n) ∪ {(id ∈ I, ⟨&s, &s + sizeof(s)⟩)}   ▷ Object metadata facility
    pthread_rwlock_unlock(&object_lock);                    partitioned into N
2:  ...                                                     disjoint structures
3:  int *p0 = &(s.array[42]);
4:  int *q0 = &(s.array[0]);
    ⟨addr, id⟩_p0 = ⟨addr ∈ A, id_s⟩
     M[addr_p0] ← ⟨&s.array[0], sizeof(s.array), id_p0⟩
    ⟨addr, id⟩_q0 = ⟨addr ∈ A, id_s⟩
     M[addr_q0] ← ⟨&s.array[0], sizeof(s.array), id_q0⟩
5:  ...
6:  *p0 = x;
7:  ...
8:  int i1;
    MARC(q0, sizeof(int), addr_q0, id_q0, 42-n);
9:  do {
10:   int i0 = φ(0, i1);
11:   if (i0 < 42-n) {
12:     int *q1 = q0 + i0;
        ⟨addr, id⟩_q1 = ⟨addr, id⟩_q0
13:     *q1     = *p0;
14:   } else {
15:     break;
```

```
16:     }
17:     i1 = i0+1;
18:   } while (1);
```

After the allocation in line 1, an entry for the object metadata associated with s is added to the n^{th} object metadata facility ${\mathcal{R}_{\mathcal{O}}}^{(n)}$. Here, it is assumed that the address of s does not flow into any pointers other than the ones in the example. Thus, p_0, q_0, and q_1 are represented in the same connected component of the $DFPG^U$. Since it is also assumed that no other threads of execution access ${\mathcal{R}_{\mathcal{O}}}^{(n)}$, the thread synchronization primitives surrounding the update operation after line 1 are eliminated.

5.2.6 Unused metadata optimization

Metadata is unused if it is no longer required for performing safety checks that have been removed with the above optimizations. Therefore, MemSafe removes such unused metadata and the code required for its propagation. Metadata is deemed unused and removed according to the following three rules, which are applied iteratively until all unused metadata has been identified and eliminated:

1. Object metadata that is not directly used for a safety check is considered unused and is removed. This includes the code that inserts object metadata into the object metadata facility $\mathcal{R}_{\mathcal{O}}$.

2. Pointer metadata that is not directly used for a safety check, stored in the $\mathcal{R}_{\mathcal{P}}$ or $\mathcal{R}_{\mathcal{F}}$ metadata facilities, or copied to other pointers as a result of pointer assignments or casts (see Section 4.3) is considered unused and is removed. This includes pointer metadata retrieved from $\mathcal{R}_{\mathcal{P}}$ or $\mathcal{R}_{\mathcal{F}}$ with lookup operations.

3. If all lookup operations for the $\mathcal{R}_{\mathcal{P}}$ or $\mathcal{R}_{\mathcal{F}}$ metadata facilities have been eliminated for all pointers in a particular connected component of the $DFPG^U$, all store operations to that metadata facility involving the pointer metadata of any pointer in the same connected component are also eliminated.

Since connected components in the $DFPG^U$ represent disjoint alias sets, the last rule simply states that for a disjoint set of pointers that may alias, if none of these pointers are ever used to retrieve metadata from $\mathcal{R}_{\mathcal{P}}$ or $\mathcal{R}_{\mathcal{F}}$, then metadata does not need to be associated with these pointers in the same metadata facility.

The following code fragment demonstrates the application of the Unused Metadata Optimization (UMO) using the running code example. Below, the source code is shown after converting the PBC within the loop body to a MARC in the loop pre-header by applying the monotonically addressed ranges optimization.

Example 5.7—Unused Metadata Optimization:

```
1:  struct { ...  int array[100]; ...  } s;
    ~~R_O = R_O ∪ {(id ∈ I, ⟨&s, &s + sizeof(s)⟩)}~~        ▷ Unused metadata can be
2:  ...                                                      eliminated
3:  int *p0 = &(s.array[42]);
4:  int *q0 = &(s.array[0]);
    ~~⟨addr, id⟩_{p0} = ⟨addr ∈ A, id_s⟩~~
    ~~M[addr_{p0}] ← ⟨&s.array[0], sizeof(s.array), id_{p0}⟩~~
    ⟨addr, id⟩_{q0} = ⟨addr ∈ A, id_s⟩
     M[addr_{q0}] ← ⟨&s.array[0], sizeof(s.array), id_{q0}⟩
5:  ...
6:  *p0 = x;
7:  ...
8:  int i1;
    MARC(q0, sizeof(int), addr_{q0}, id_{q0}, 42-n);
9:  do {
10:   int i0 = φ(0, i1);
11:   if (i0 < 42-n) {
12:     int *q1 = q0 + i0;
        ~~⟨addr, id⟩_{q1} = ⟨addr, id⟩_{q0}~~
```

```
13:       *q1     = *p0;
14:     } else {
15:       break;
16:     }
17:     i1 = i0+1;
18:   } while (1);
```

Since all object bounds checks have been eliminated previously, the update of the object metadata facility $\mathcal{R}_{\mathcal{O}}$ after the allocation in line 1 is eliminated. Similarly, since the pointer bounds checks for the dereference of p_0 and q_1 have been eliminated, and since the pointer metadata for p_0 and q_1 (defined after lines 4 and 12, respectively) is not propagated through either $\mathcal{R}_{\mathcal{P}}$ or $\mathcal{R}_{\mathcal{F}}$, it is removed as well. After applying all of the above optimizations, the only code remaining that is required to enforce memory safety is the definition of the pointer metadata for q_0 after line 4, and the `MARC` before line 9, which uses this metadata.

The unused metadata optimization is similar to, but not the same as, the well-known dead code elimination optimization [7] in compiler theory. Dead code elimination is used to remove code that is guaranteed to be never executed (unreachable code), and code that operates on dead variables. Unreachable code is identified by inspecting a program's call and control-flow graphs. For example, a basic block that has no immediate predecessors in the control-flow graph represents unreachable code. Dead variables are variables that are defined but never used. Such unused variables are trivial to identify in a program's SSA form since each assignment is given a unique name. If a defined value never appears on the right-hand side of an expression, the value is never used, and it is safe to eliminate the assignment.

However, the unused metadata optimization cannot be accomplished using dead code elimination since the metadata, and the code inserted for its propagation, is neither necessarily unreachable nor does it represent dead code. The inserted metadata is assumed reachable since it is a requirement of the inserted safety checks, which themselves are assumed reachable, and the code for metadata propagation, even if all safety checks have been eliminated, is not dead code since $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ must maintain state throughout the execution of a program. Thus, the unused metadata optimization is an enhancement of traditional dead code elimination achieved through the use of application-specific knowledge.

Chapter 6

MemSafe Implementation

Having described MemSafe's approach for inserting and optimizing the runtime checks needed for ensuring memory safety, this chapter describes the implementation of the MemSafe compiler and the global data structures it requires for maintaining metadata. Additionally, implementation issues related to the C language and the typical C programming development process are also considered.

6.1 MemSafe's Analysis and Transformation

MemSafe is implemented within the Low Level Virtual Machine (LLVM) [56] compiler infrastructure. LLVM's intermediate representation is a low-level, typed SSA [26] form that is language-independent and also independent of any instruction set architecture. Thus, the implementation of MemSafe's transformation for ensuring memory safety is not specific to a particular computer architecture, and in theory, could also be used to enforce safety for languages other than C. However, MemSafe has not been tested for this purpose. By default, MemSafe uses Andersen's analysis [4] for interprocedural flow- and context-insensitive points-to information, but MemSafe is compatible with

any alias analysis implementation that operates within the LLVM infrastructure.

MemSafe consists of a collection of analyses and transformations that each contribute a portion of the overall approach described in Chapter 4. These include:

1. An analysis and transformation that creates a temporary global variable for representing the *invalid* pointer and assigns it to each pointer whose referent is deallocated, as presented in Section 4.1.1. Pointers to deallocated objects are identified by locating calls to `free` and by determining the local variables whose addresses may escape the function in which they are defined. While MemSafe conservatively assumes that the address of a variable escapes if it is saved in another variable, an escape analysis (implemented as a client of alias analysis) can be used to obtain more precise information.

2. An analysis and transformation that uses the results of alias analysis to insert ϱ-functions, as described in Section 4.1.2. For simplicity, ϱ-functions are implemented as calls to a variable-argument function since ϱ-function arguments can be of any (pointer) type and number.

   Additionally, a simple reachability analysis is used to improve upon the results of alias analysis. For example, consider the pointer store operation `*ptr1 = p0` and the pointer load operation `p1 = *ptr2`. If alias analysis indicates that the pointers ptr_1 and ptr_2 may alias, p_0 would be added to the ϱ-function inserted for p_1. However, if there is no control-flow path from the store operation to the load operation, this is unnecessary since there is no program execution in which the stored value can modify the loaded value. MemSafe does not include

stored pointers in the ϱ-functions inserted for loaded pointers if the store cannot reach the load. Note that the imprecision of this approach is the result of the flow-insensitivity of Andersen's analysis and that MemSafe's reachability analysis does not result in a flow-sensitive alias analysis.

3. An analysis that constructs the $DFPG$ for the transformed code, as described in Section 5.1. The $DFPG$ is made available to the remaining transformations.

4. An analysis and transformation that inserts the required runtime checks and metadata for ensuring memory safety, according to the check insertion and metadata propagation rules in Sections 4.2–4.3 and the optimizations in Section 5.2. The metadata facilities $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$ are declared as global data structures and code for their initialization is placed in `main`.

5. A pass that aggressively removes the inserted assignments of the *invalid* pointer (but not its associated metadata) and all ϱ-functions, since these are only used for MemSafe's analysis.

Each of the above analyses and transformations are run sequentially on a program after it as been translated into LLVM's intermediate form, linked, and optimized with LLVM's standard set of compiler optimizations. Applying MemSafe's transformation after LLVM's optimizations improves the results of alias analysis and ensures that MemSafe avoids inserting unnecessary checks. The same optimizations are run after MemSafe has completed its transformation to further optimize the inserted checks and to eliminate any dead code MemSafe may have introduced during its analysis.

6.2 Metadata Facilities

The global facilities ($\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$ and $\mathcal{R}_{\mathcal{F}}$) MemSafe requires for maintaining object and pointer metadata can be implemented using any data structure that supports efficient insertion, deletion, and retrieval operations. For simplicity and ease of implementation, MemSafe uses dynamically resized hash tables for all three metadata facilities. Collisions are resolved using separate chaining. Hash functions are a modulo of the key with the size of the table, which becomes an efficient bitwise `and` operation by restricting table sizes to powers of two.

The prototype implementation of MemSafe makes two simplifications in the process described for creating pointer and object metadata. Fist, MemSafe acquires addresses $addr \in A$ for storing a pointer's pointer metadata using `malloc`, and this storage is released back to the system using `free` when a pointer's metadata is updated to be that of the *invalid* pointer. Second, MemSafe acquires a unique $id \in I$, which is used as a key for the object metadata facility by incrementing a global counter. While this ensures that each generated id is unique, this places a finite limitation on the number of objects that can be allocated by a program. However, note that a 4GHz computer would take 136 years to overflow a 64-bit counter allocating a new object on every clock cycle.

6.2.1 Implementation alternatives

Although the prototype implementation of MemSafe utilizes hash tables for maintaining the required metadata, other implementations are possible. In particular, tries [38]

can often be used as an alternative to hash tables. A trie is a prefix tree data structure in which the position of a node in the tree indicates the key with which the node's data is associated. Tries are useful for implementing associative containers because the time required for performing insertion, deletion, and retrieval operations is $O(m)$, where m is the length of a key. Although a hash table is capable of performing these operations in constant time, complex hash functions, key collisions, and table resizing can, in the worst case, degrade their performance to $O(n)$ time, where n is the number of keys. Tries are also simpler than hash tables to implement effectively, and they are commonly used for implementing dictionaries and spell checking algorithms.

Splay trees can also be used instead of hash tables for implementing MemSafe's metadata facilities. A splay tree [78] is a self-optimizing binary search tree in which the frequently accessed nodes are moved towards the root of the tree, where they become quick to access again. The insertion, deletion, and retrieval operations of a splay tree are each performed in $O(\log n)$ amortized time, where n is the number of nodes in the tree. When accessing a node, a special *splaying* operation is performed that uses a sequence of tree rotations to place the accessed node at the root of the tree. Because of this self-optimizing property, if used to maintain metadata, splay trees can mimic a program's locality of reference and ensure the base and bound information of the most frequently dereferenced pointers is kept near the root of the tree. Another advantage of splay trees is that, because their elements are sorted, they can be used to perform range lookups, which can be useful for some forms of metadata propagation (see Section A.1 for an example). Splay trees are commonly used to implement software caches and garbage collection algorithms.

While the above design alternatives apply to all three metadata facilities, the function metadata facility $\mathcal{R}_{\mathcal{F}}$ could also be implemented using a stack instead of an associate container. Just like parameters are passed to functions by pushing them onto the call stack, the $\langle addr, id \rangle$ pointer metadata associated with pointer arguments could be pushed onto a disjoint *shadow stack* before procedure calls. A callee could then pop the required metadata from the top of the shadow stack at the beginning of the procedure. The advantage of this approach is its simplicity: although the hash table is an effective implementation of the function metadata facility, a stack-based approach would likely require less memory overhead and the insertion, deletion, and retrieval operations would be more predictable in runtime.

As a final implementation alternative, pointer metadata could be passed to functions through the use of additional arguments. In order to do so, each function accepting pointer arguments must be replaced with an equivalent function accepting an additional metadata argument for each of its original pointer arguments. While this approach is essentially a stack-based model, it eliminates the need for maintaining a separate data structure for $\mathcal{R}_{\mathcal{F}}$. However, the requirement to modify function prototypes would complicate interfacing with variadic functions and pre-compiled libraries.

6.3 Metadata Allocation

Since the metadata associated with pointers is propagated at pointer assignments, it can be efficiently allocated locally using automatic memory that is disjoint from the pointers with which it is associated. However, the metadata that is maintained

in the global metadata facilities requires dynamic storage. To avoid the runtime cost associated with dynamic memory management, MemSafe initially sizes the $\mathcal{R}_{\mathcal{P}}$ and $\mathcal{R}_{\mathcal{F}}$ metadata facilities large enough such that the need to dynamically allocate additional storage is a rare occurrence.

MemSafe makes the following improvement to the above approach for lowering the dynamic allocation overhead of the object metadata facility $\mathcal{R}_{\mathcal{O}}$. Since object metadata exists in $\mathcal{R}_{\mathcal{O}}$ only for as long as the object it is associated with is allocated, the storage space required for object metadata can be located with the objects. For stack-allocated objects, their corresponding object metadata can be efficiently allocated on the stack, and a pointer to this structure can be maintained in $\mathcal{R}_{\mathcal{O}}$. For an object allocated on the heap, instead of requiring the program to perform an additional call to `malloc` to allocate the storage needed for the metadata, MemSafe transforms the program such that the object's metadata is maintained in a header structure appended to the beginning of the allocated region.

Figure 6.1 demonstrates MemSafe's transformation for lowering the runtime cost associated with dynamically allocating object metadata. In the original code fragment (a), an array of ten of structures is allocated dynamically with `malloc`, and then this object is later deallocated with `free`. The metadata associated with *array* (not shown) must be dynamically allocated and inserted into $\mathcal{R}_{\mathcal{O}}$.

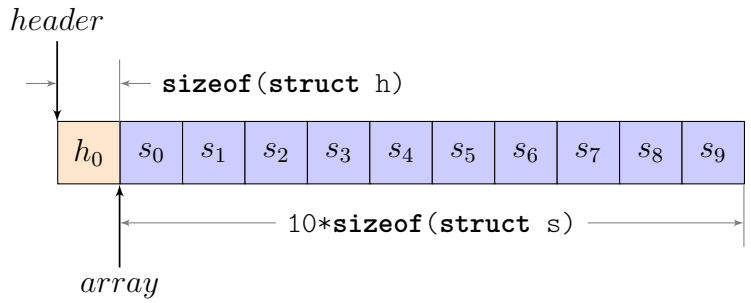
In the transformed code (b), line 1 computes the storage space required for the original array plus one header structure h for storing *array's* object metadata. In line 2, a region of memory is allocated and the pointer *header* is created to point to the metadata structure. This pointer is incremented by the size of one header structure in

```
1: array = (struct s*) malloc(10 * sizeof(struct s));
   ...
2: free(array);
```

(a) Original source code

```
1: size   = 10 * sizeof(struct s) + sizeof(struct h);
2: header = (struct h*) malloc(size);
3: array  = (struct s*) (header + 1);
4: header->base  = (void*) array;
5: header->bound = (void*) (array + 10);
6: header->id    = id;
   ...
7: header = ((struct h*) array) - 1;
8: free(header);
```

(b) Transformed source code



(c) Memory layout

Figure 6.1: Header allocation. The original source code (a) is transformed (b) to efficiently allocate memory for the header and initialize the object metadata. The memory layout (c) of the resulting allocation contains enough padding to hold the header in addition to the original data.

line 3 to obtain a pointer to the array. Lines 4–6 store the metadata of *array* in the header structure, and a pointer to this structure is maintained in $\mathcal{R}_{\mathcal{O}}$ (not shown). In order to deallocate the array, the base address of the allocated object is computed in line 7, and in line 8, this address is used to deallocate the entire region. Note that the object metadata for *array* would have been unmapped in $\mathcal{R}_{\mathcal{O}}$ before the call to `free` (see Section 4.3). The resulting memory layout (c) of the entire structure shows the location of the two pointers.

6.4 Limitations

Although MemSafe's method of ensuring the memory safety of C is complete and compatible with most programs, given C's weak typing guarantees and the typical application development process, in practice, MemSafe is not free from limitations. For example, the implementation of MemSafe currently does not support inline assembly instructions and does not allow self-modifying code. For programs requiring assembly, MemSafe could be extended with the appropriate rules for handling these instructions, but this would likely limit the effectiveness of MemSafe's optimizations. Self-modifying code is now commonly disabled by default in most modern operation systems. Additional limitations of MemSafe's implementation are discussed below.

6.4.1 Separate compilation

MemSafe's most significant limitation is its use of whole-program analysis to limit the number of required checks and to avoid unnecessary metadata propagation. Although

analyzing the entire program is essential for reducing the cost of software-provided memory safety, it negates the advantages of separate compilation, and can be problematic for use in common build environments. However, MemSafe's whole-program analysis, which is based the construction of the $DFPG$, is not required for enforcing safety. The checks and metadata propagation described in Sections 4.2–4.3 are fully compatible with separate compilation, and MemSafe's optimizations can be turned off for programs where whole-program analysis is infeasible. Chapter 7 presents performance overheads with and without using whole-program analysis.

6.4.2 `NULL` and manufactured pointers

Pointers that are `NULL` or defined as a type-cast from a non-pointer type cannot be associated with a valid object. Therefore, MemSafe sets the base and bound of such pointers to be equal to that of the *invalid* pointer. Although this may result in false positives, they have been observed to be rare occurrences in practice. For reading and writing to memory-mapped I/O locations, MemSafe requires a target's backend to specify the base and bound address of all valid address ranges.

Chapter 7

Results

Having discussed the prototype implementation of the MemSafe compiler, this chapter presents a thorough evaluation of MemSafe's approach for ensuring the memory safety of C programs at runtime. Specifically, this chapter evaluates (1) MemSafe's completeness by demonstrating that it is capable of detecting known memory safety violations in several large programs, (2) MemSafe's runtime cost by measuring its runtime overhead on a variety of programs and comparing this slowdown with that of prior methods, and (3) the effectiveness of MemSafe's static analysis by measuring quantities related to MemSafe's data-flow representation and the number of required checks and performed optimizations.

In performing the above evaluation, it will be demonstrated that MemSafe is compatible with a variety of C programs and that it does not require any source code modifications or programmer intervention. Additionally, it will be shown that MemSafe's key contributions—namely, the modeling of temporal violations as spatial violations, the use of a hybrid metadata representation, and MemSafe's data-flow representation—are effective tools for reducing the runtime cost of dynamically ensuring memory safety.

Benchmark		Size		Detected All
Suite	Program	LOC	Derefs	
BugBench	099.go	29246	16632	yes
	129.compress	1934	232	yes
	bc-1.06	14288	2474	yes
	gzip-1.2.4	9076	1722	yes
	ncompress-4.2.4	1922	838	yes
	polymorph-0.4.0	716	65	yes

Table 7.1: Violations detected in BugBench. MemSafe's ability to detect all memory violations in the BugBench [60] programs is demonstrated. Program size is measured in lines of code (LOC) and the number of static dereferences.

7.1 Effectiveness in Detecting Errors

To provide evidence of its completeness and ability to detect real errors, MemSafe was evaluated on programs containing known memory errors from the BugBench [60] suite of programs. BugBench is a collection of programs containing various documented software bugs that was expressly created to evaluate the effectiveness of error detection tools. Table 7.1 shows that MemSafe is capable of detecting all known memory errors in six programs from BugBench. BugBench programs that were excluded from Table 7.1 include programs that only contain errors that are *not* related to spatial or temporal safety (e.g., memory leaks and race conditions). Thus, the programs in Table 7.1 are representative of all memory safety violations in BugBench. The size of each program is given in lines of code (LOC) and the number of static dereferences.

MemSafe's ability to detect real-world memory violations was further validated by it compiling two large applications and successfully detecting the known memory errors. Table 7.2 summarizes the memory safety violations detected by MemSafe in various versions of the Apache HTTP server [6] and the GNU Core Utilities [39] software

Application	Version	LOC	Component	Detected Violation
Apache HTTP Server*	2.0.39	262487	mod_ext_filter	null dereference
	2.0.40	266741	mod_env	null dereference
	2.0.46	282682	mod_ssl	dangling pointer
	2.0.48	284627	mod_ssl	null dereference
	2.0.50	262266	mod_rewrite	buffer overflow
	2.0.52	263513	mod_auth_ldap	null dereference
	2.0.54	265243	mod_auth_ldap	null dereference
	2.0.59	267783	mod_rewrite	uninitialized pointer
	2.2.0	310283	mod_proxy	double free
	2.2.2	311235	mod_dbd	double free
	2.2.6	314531	mod_proxy_balancer	buffer overflow
	2.2.8	316713	mod_log_config	null dereference
	2.2.9	332867	mod_ldap	null dereference
	2.3.4	206590	mod_proxy	null dereference
GNU Core Utilities†	5.2.1	103659	fts	double free
	5.2.1	103659	copy	buffer overflow
	5.2.1	103659	who	buffer overflow
	5.3.0	107147	cut	double free
	5.9.0	112781	regexec	buffer overflow
	6.10	69491	mkfifo	null dereference
	6.10	69491	mknod	null dereference
	6.10	69491	ptx	buffer overflow

Table 7.2: Violations detected in real-world applications. MemSafe's ability to detect known real-world memory violations in the Apache HTTP Server [6] and GNU Core Utilities [39] software package is demonstrated. Program size is measured in lines of code (LOC).

*Source of violations: https://issues.apache.org/bugzilla/

†Source of violations: http://lists.gnu.org/archive/html/bug-coreutils/

package. The Apache HTTP server is a widely-used open source web server, and the GNU Core Utilities is a GNU software package that provides the basic text, shell, and file utilities (e.g., `cat`, `expr`, and `cp`) common among virtually all Unix-like operating systems. To reproduce the known errors in these programs, the online development archive and bug database of each was consulted in order to identify particular versions of the software that contain memory safety violations and the runtime conditions necessary for producing them. Having discovered the known violations, MemSafe was then used to compile each version of the software, and the programs were executed to verify that the inserted runtime checks successfully detected the violations. The size of each program in Table 7.2 is given in lines of code.

7.2 Runtime Performance

MemSafe's increase in runtime and memory consumption was measured on a total of 30 programs from the Olden [71], PtrDist [9] and SPEC [79] benchmark suites. Programs from Olden and PtrDist suites are known for being memory allocation intensive, while those from SPEC are larger and generally more computationally intensive. The programs were executed on a system running the Ubuntu 8.04 LTS Desktop operating system with Linux kernel version 2.6.24. The system contains a single 3GHz Pentium 4 processor and 2GB of main memory. Program execution times were determined by taking the lowest of three times obtained using the GNU/Linux `time` command, and memory usage was measured by instrumenting all allocation and deallocation instructions to record the number of allocated objects and their sizes.

Due in part to LLVM's research-quality implementation of Andersen's analysis, the current implementation of MemSafe is not yet robust enough to compile the entire set of SPEC benchmarks. The results presented in this section pertain to the subset that MemSafe correctly compiles.

7.2.1 Increase in runtime

Table 7.3 summarizes the runtime and memory consumption overheads of MemSafe's fully optimized approach. While this section discusses the increase in runtime of programs compiled with MemSafe's safety checks, the discussion of their increase in memory consumption is deferred until Section 7.2.2.

The "Runtime" and "Slowdown" columns of Table 7.3 show that MemSafe ensured complete spatial and temporal safety for all 30 programs with an average overhead of 88%. In general, MemSafe's overhead was observed to be comparable to that of CCured [64]: on the allocation intensive Olden benchmarks, MemSafe's overhead was 29% versus CCured's 30%, and on CCured's entire set of reported benchmarks, MemSafe overhead was 69% versus CCured's 80%. Not including *bc* (on which CCured's overhead was particularly high) reduces these to 65% and 30%, respectively. While the runtime cost of MemSafe is similar to that of CCured, MemSafe does not incur the drawbacks associated with the use of CCured—the need for manual modifications and the compatibility issues arising from the use of "fat pointer." Due to CCured's need for manual code modifications, results for CCured on additional programs were not obtained.

Benchmark		Size		Runtime (s)		Memory (MB)		Slowdown		
Suite	Program	LOC	Derefs	Base	MemSafe	Base	MemSafe	MemSafe	CCured	MSCC
Olden	bh	2073	284	4.64	5.34	14.67	17.70	1.15	1.44	2.82
	bisort	350	76	1.31	1.59	214.73	275.12	1.21	1.45	1.76
	em3d	688	187	5.11	6.95	54.84	55.15	1.36	1.87	1.79
	health	502	236	0.47	0.70	36.63	53.61	1.48	1.29	2.72
	mst	428	57	0.31	0.36	24.90	24.92	1.17	1.06	1.76
	perimeter	484	258	0.36	0.48	37.33	80.12	1.34	1.09	3.37
	power	622	285	4.09	4.70	2.91	5.14	1.15	1.07	1.22
	treeadd	245	26	0.38	0.59	48.00	182.92	1.55	1.10	3.23
	tsp	582	194	3.83	4.44	144.00	278.65	1.16	1.15	2.28
	average	*716*	*178*	*2.28*	*2.79*	*64.22*	*108.15*	*1.29*	*1.30*	*2.33*
PtrDist	anagram	650	113	1.56	2.96	0.24	0.25	1.90	1.43	–
	bc	7297	3927	1.34	3.15	0.72	0.72	2.35	9.91	–
	ft	1766	246	2.04	3.61	3.24	9.20	1.77	1.03	–
	ks	782	239	1.54	2.97	0.02	0.09	1.93	1.11	–
	yacr2	3986	1000	1.96	5.98	29.88	30.17	3.05	1.56	–
	average	*2896*	*1105*	*1.69*	*3.73*	*6.82*	*8.09*	*2.20*	*3.01*	–
SPEC'95	099.go	29246	16632	0.62	1.26	0.00	0.01	2.03	1.22	2.60
	129.compress	1934	232	0.01	0.02	0.00	0.00	2.20	1.17	1.85
	130.li	7597	4905	0.06	0.12	19.91	19.91	1.93	1.70	–
	147.vortex	67202	25135	0.00	0.00	96.41	96.41	–	–	–
	average	*26495*	*11726*	*0.17*	*0.35*	*29.08*	*29.08*	*2.05*	*1.36*	–
SPEC'00	164.gzip	8605	1499	20.72	43.10	187.93	187.94	2.08	–	1.46
	175.vpr	17729	5386	8.34	16.26	44.35	44.37	1.95	–	3.53
	181.mcf	2412	534	11.34	21.89	99.86	99.46	1.93	–	2.85
	186.crafty	24975	7579	14.93	34.94	7.09	7.14	2.34	–	–
	255.vortex	67213	25134	3.96	8.28	96.46	96.46	2.09	–	–
	256.bzip2	4649	1254	22.33	45.55	191.95	191.96	2.04	–	–
	300.twolf	20459	11741	7.52	15.34	6.39	12.74	2.04	–	–
	average	*20863*	*7590*	*12.73*	*26.48*	*90.58*	*91.49*	*2.07*	–	–
SPEC'06	401.bzip2	8293	4013	6.20	17.55	855.79	855.79	2.83	–	–
	445.gobmk	197215	27614	0.29	0.66	28.50	28.66	2.27	–	–
	456.hmmr	35992	7582	7.82	17.52	59.82	59.89	2.24	–	–
	458.sjeng	13847	5832	10.12	22.26	179.63	179.64	2.20	–	–
	473.astar	5842	1873	0.00	0.00	313.15	313.15	–	–	–
	average	*52238*	*9383*	*4.89*	*11.60*	*287.38*	*287.43*	*2.39*	–	–
Average		**18394**	**5136**	**4.77**	**9.62**	**93.31**	**106.92**	**1.88**	–	–

Table 7.3: Dynamic results with whole-program analysis. Program size is measured in lines of code and the number of static dereferences, runtime is measured in seconds, and memory consumption is measured in megabytes. Slowdown is computed as the ratio of the execution time of the instrumented program to that of the original program. Slowdown for MemSafe with all optimizations is shown in comparison with CCured [64] and MSCC [82] where results are available.

Benchmark		Size		Runtime (s)		Memory (MB)		Slowdown		
Suite	Program	LOC	Derefs	Base	MemSafe	Base	MemSafe	MemSafe	CCured	MSCC
Olden	bh	2073	284	5.00	9.35	14.67	19.52	1.87	1.44	2.82
	bisort	350	76	1.32	5.77	214.73	294.74	4.37	1.45	1.76
	em3d	688	187	5.22	7.41	54.84	55.39	1.42	1.87	1.79
	health	502	236	0.46	2.36	36.63	57.96	5.12	1.29	2.72
	mst	428	57	0.30	0.38	24.90	24.93	1.26	1.06	1.76
	perimeter	484	258	0.36	1.83	37.33	90.67	5.09	1.09	3.37
	power	622	285	4.09	4.83	2.91	5.94	1.18	1.07	1.22
	treeadd	245	26	0.38	2.09	48.00	208.00	5.51	1.10	3.23
	tsp	582	194	3.83	19.04	144.00	304.00	4.97	1.15	2.28
	average	*716*	*178*	*2.33*	*5.89*	*64.22*	*117.91*	*3.42*	*1.30*	*2.33*
PtrDist	anagram	650	113	1.57	2.97	0.24	0.25	1.89	1.43	–
	bc	7297	3927	1.34	4.03	0.72	0.73	3.01	9.91	–
	ft	1766	246	2.05	3.63	3.24	11.15	1.77	1.03	–
	ks	782	239	1.54	3.33	0.02	0.09	2.16	1.11	–
	yacr2	3986	1000	1.97	6.15	29.88	30.20	3.12	1.56	–
	average	*2896*	*1105*	*1.69*	*4.02*	*6.82*	*8.49*	*2.39*	*3.01*	–
SPEC'95	099.go	29246	16632	0.62	1.25	0.00	0.01	2.02	1.22	2.60
	129.compress	1934	232	0.01	0.02	0.00	0.01	1.78	1.17	1.85
	130.li	7597	4905	0.06	0.20	19.91	19.92	3.32	1.70	–
	147.vortex	67202	25135	0.00	0.00	96.41	96.42	–	–	–
	average	*26495*	*11726*	*0.17*	*0.37*	*29.08*	*29.09*	*2.37*	*1.36*	–
SPEC'00	164.gzip	8605	1499	20.75	43.99	187.93	187.94	2.12	–	1.46
	175.vpr	17729	5386	8.39	27.27	44.35	44.37	3.25	–	3.53
	181.mcf	2412	534	11.28	30.79	99.86	99.86	2.73	–	2.85
	186.crafty	24975	7579	14.95	42.91	7.09	7.15	2.87	–	–
	255.vortex	67213	25134	3.95	21.73	96.46	96.46	5.50	–	–
	256.bzip2	4649	1254	22.30	53.74	191.95	191.96	2.41	–	–
	300.twolf	20459	11741	7.51	30.94	6.39	14.71	4.12	–	–
	average	*20863*	*7590*	*12.73*	*35.91*	*90.58*	*91.78*	*3.29*	–	–
SPEC'06	401.bzip2	8293	4013	6.23	24.23	855.79	855.79	3.89	–	–
	445.gobmk	197215	27614	0.30	0.99	28.50	28.77	3.30	–	–
	456.hmmr	35992	7582	7.82	28.00	59.82	59.89	3.58	–	–
	458.sjeng	13847	5832	10.11	30.23	179.63	179.64	2.99	–	–
	473.astar	5842	1873	0.00	0.00	313.15	313.15	–	–	–
	average	*52238*	*9383*	*4.89*	*16.69*	*287.38*	*287.45*	*3.44*	–	–
Average		**18394**	**5136**	**4.79**	**13.65**	**93.31**	**109.99**	**3.09**	–	–

Table 7.4: Dynamic results with separate compilation. Program size is measured in lines of code and the number of static dereferences, runtime is measured in seconds, and memory consumption is measured in megabytes. Slowdown is computed as the ratio of the execution time of the instrumented program to that of the original program. Slowdown for MemSafe with all optimizations is shown in comparison with CCured [64] and MSCC [82] where results are available.

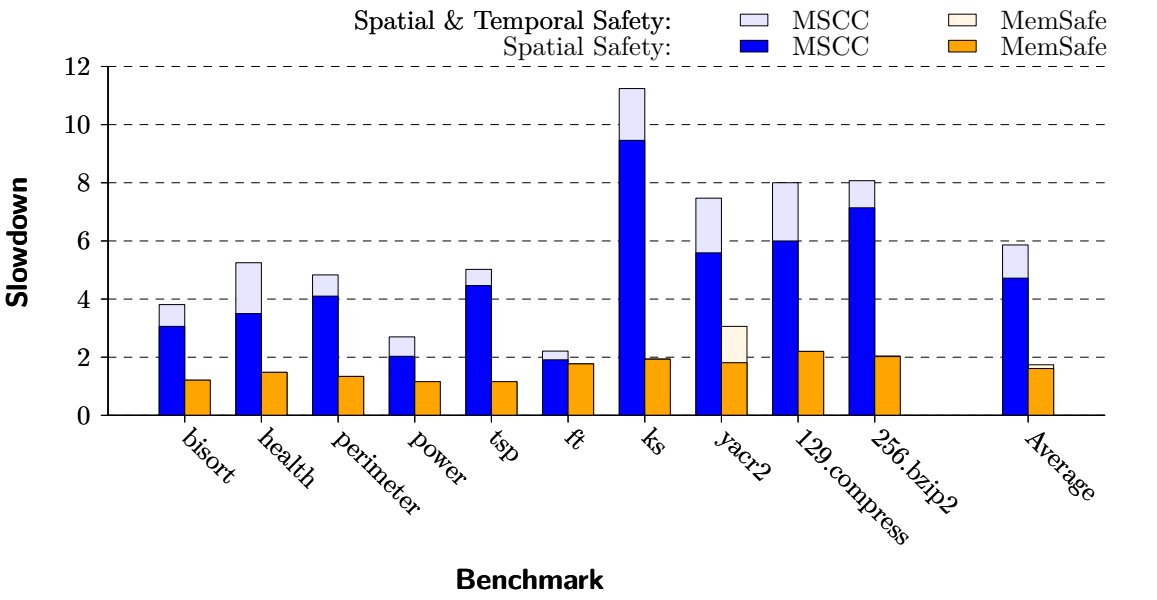


Figure 7.1: Runtime comparison with MSCC. Slowdown for MemSafe and MSCC [82] is shown for spatial and temporal and spatial-only safety.

Additionally, MemSafe demonstrated a significant and consistent improvement over the reported performance of MSCC [82], the tool with the lowest overhead among all existing complete and automatic methods that detect both spatial and temporal errors. On the Olden benchmarks, MemSafe's average overhead (29%) was roughly 1/4 that of MSCC (133%), and on the entire set of MSCC's reported benchmarks, MemSafe's overhead (44%) was roughly 1/3 that of MSCC (137%).

In order to provide a direct comparison with MSCC (instead of relying on published results) on the same computer hardware, an attempt was made to compile our entire set of benchmarks with MSCC. However, perhaps due to MSCC having not been actively maintained since its publication, it was found to be difficult to compile MemSafe's entire set of 30 benchmarks with MSCC. Figure 7.1 compares the slowdown of MemSafe's fully optimized approach for spatial and temporal safety with that of MSCC on the set of benchmarks MSCC compiled correctly. MemSafe's average

overhead for these benchmarks (74%) was roughly 1/6 that of MSCC (486%). While these results show a dramatic increase in runtime overhead for MSCC, the overall trend is similar to the reported results for MSCC shown in Table 7.3. Comparisons with additional methods on the Olden benchmarks is presented in Section 1.2.

MemSafe's optimized approach improves the runtime cost required for memory safety in comparison to that of prior work for the following reasons: (1) MemSafe's data-flow representation enables performance-enhancing optimizations that reduce overhead from 253% to 88% (explained later). (2) MemSafe's modeling of temporal errors as spatial errors, combined with a hybrid metadata representation, enables MemSafe to ensure temporal safety with only a 10% increase in the overhead of spatial safety alone (also explained later). In particular, MemSafe's large improvement versus MSCC on the Olden benchmarks is due to the fact that these programs deallocate all dynamically allocated memory at once before terminating. Thus, by determining that there is no control-flow path from the deallocation to other points in the program, MemSafe is able to eliminate the propagation of the metadata associated with the *invalid* pointer and remove all object bounds checks. Deallocated memory at the end of a program is a common programming style when objects are required to have an unlimited lifespan or when memory reallocation is not needed.

Table 7.4 summarizes, in the same way as Table 7.3, the runtime performance of MemSafe's fully optimized approach, but when limited by disabling whole-program-analysis. Moreover, MemSafe was instructed to not use interprocedural information when inserting the required runtime checks and code for propagating metadata. Thus, these results represent the worse-case execution time associated with separate

compilation, since each function, in a sense, was processed as if it were contained in a separate module. The "Slowdown" column of Table 7.4 shows that MemSafe's average runtime overhead increases to 209% overall and to 242% on the Olden benchmark suite by disallowing interprocedural analysis.

To explain the seemingly large increase in MemSafe's runtime overhead when restricted to separate compilation, note that while not being true whole-program analyses themselves, both CCured and MSCC link the intermediate representation of separately processed files together to form one monolithic representation of the program before generating object code. Thus, the compiler backend of each of these tools benefits from interprocedural information when performing optimizations. Interprocedural optimization was completely disabled to obtain the runtime overheads for MemSafe's approach presented in Table 7.4.

7.2.2 Increase in memory consumption

The "Memory" column of Table 7.3 reports the memory consumption of each program when compiled with the LLVM compiler and when compiled with MemSafe. MemSafe ensured complete spatial and temporal safety for all 30 programs with an average increase in memory of 13.61MB, which is equal to 48.60% of the programs' original memory requirements. MemSafe's average memory consumption overhead is significantly higher for the Olden (73.52%) and PtrDist (130.32%) benchmark suites compared to the three SPEC (8.46%) benchmark suites. Since MemSafe requires the metadata of each allocated object to be mapped in the object metadata facility,

allocation intensive programs, like those in the Olden and PtrDist suites, can be expected to require more memory for maintaining metadata than computationally intensive programs. In particular, the memory required for MemSafe's metadata is determined by the number of allocated objects rather than their total size.

Table 7.4 shows MemSafe's increase in memory consumption when whole-program-analysis is disabled. Without the full effectiveness of MemSafe's optimizations for eliminating unnecessary checks and metadata, MemSafe's average memory consumption increases from 13.61MB to 16.67MB (57.77%).

7.2.3 Effectiveness of optimizations

Figure 7.2 shows that MemSafe's optimizations and whole-program analysis are effective tools for reducing the runtime overhead required for ensuring memory safety. Shown in the "Average" histogram, MemSafe's optimizations reduced its average runtime overhead from 253% to 88%. Since the optimization for dominated dereferences (DDO) is minimally effective, it is presented in Figure 7.2 as the baseline. The optimization for temporally-safe dereferences (TDO) reduced overhead by 102%, and the optimization for non-incremental dereferences (NDO) reduced overhead by 37%. Combined with the optimization for unused metadata, which is included with both, NDO and TDO accounted for the greatest reduction in overhead. The optimization for monotonically addressed ranges (MRO) was marginally effective and reduced overhead by only 1%.

Figure 7.3 shows the effectiveness of MemSafe's optimizations without utilizing

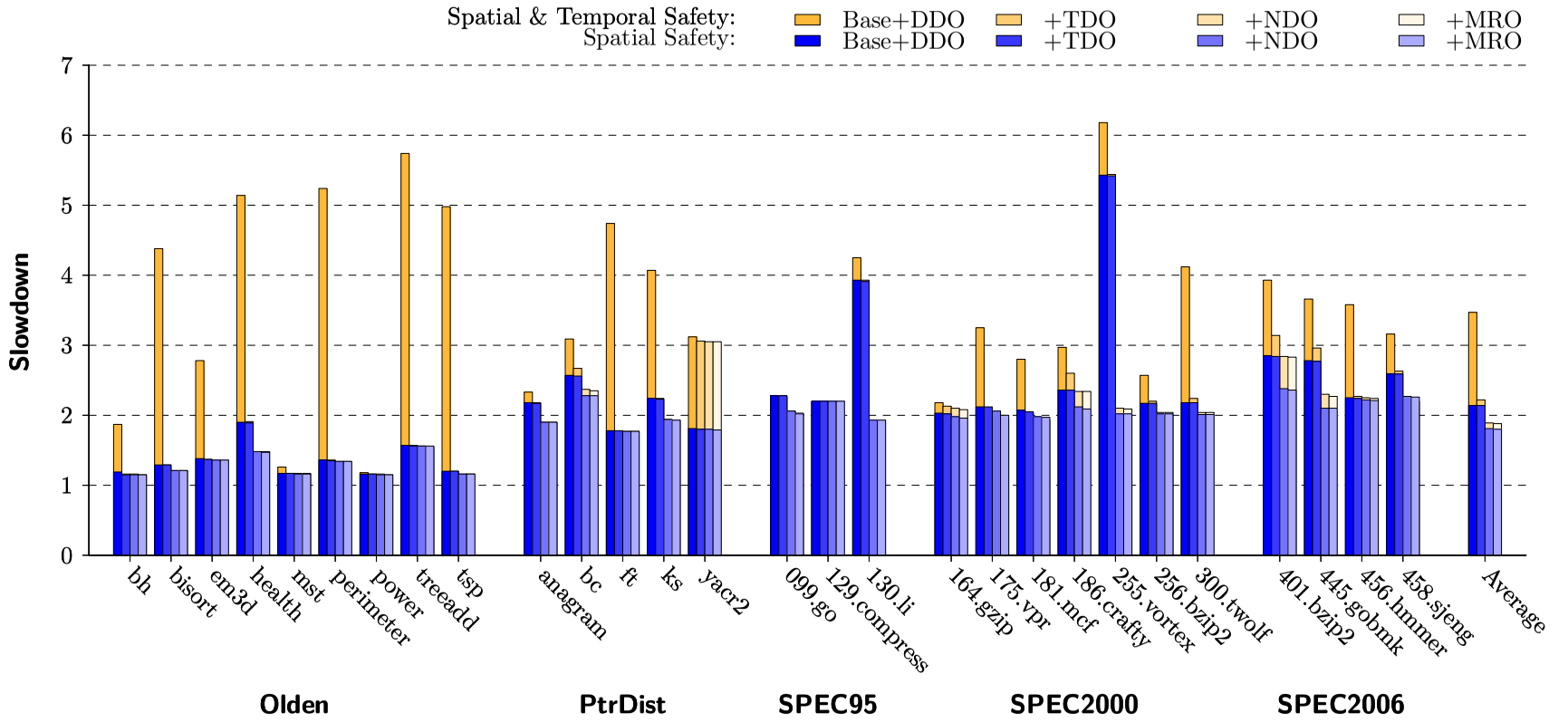


Figure 7.2: Optimization effectiveness with whole-program analysis. Slowdown is shown for spatial and temporal and spatial-only safety. Optimizations include dominated dereferences (DDO), temporally-safe dereferences (TDO), non-incremental dereferences (NDO), and monotonically addressed ranges (MRO).

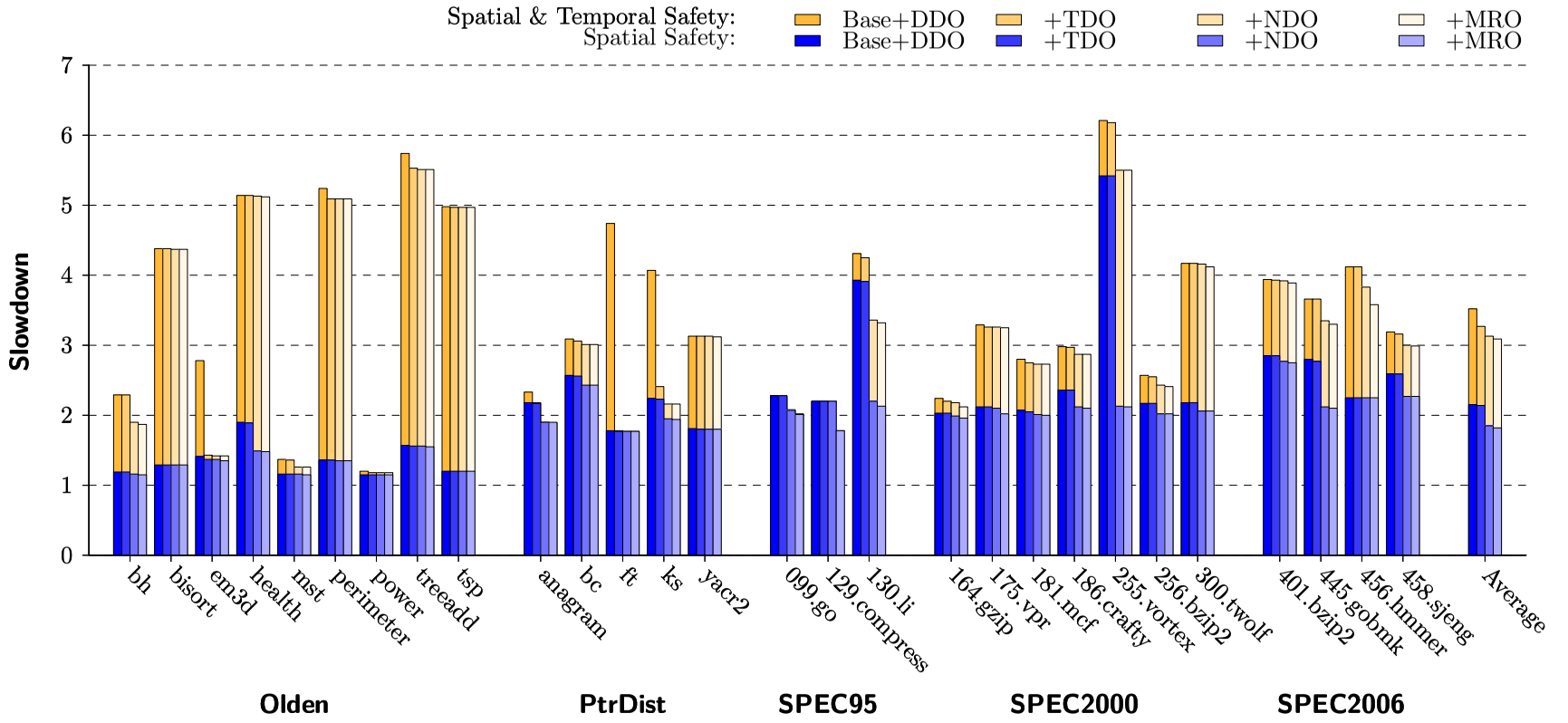


Figure 7.3: Optimization effectiveness without whole-program analysis. Slowdown is shown for spatial and temporal and spatial-only safety. Optimizations include dominated dereferences (DDO), temporally-safe dereferences (TDO), non-incremental dereferences (NDO), and monotonically addressed ranges (MRO).

whole-program analysis. When restricted to not use interprocedural information, MemSafe's optimizations reduced the overhead from 253% to 209%. TDO reduced overhead by 11%, NDO reduced overhead by 7%, and MRO reduced overhead by an additional 2%. Hence, MemSafe's average overhead with separate compilation was 209% versus 88% with whole-program analysis. MemSafe's seemingly large improvement in runtime overhead when given the ability to perform interprocedural optimizations is not by chance: By representing memory deallocation and pointer stores as direct assignments, MemSafe makes whole-program optimizations much more effective. Thus, MemSafe's overheads are lower than those of existing methods that cannot benefit in this way.

7.2.4 Additional cost of temporal safety

Figure 7.2 also quantifies the additional runtime cost required for MemSafe to ensure temporal safety. The last bar in the "Average" histogram shows that MemSafe's overhead for both spatial and temporal safety (88%) is comparable to the runtime overhead MemSafe requires for ensuring spatial safety alone (80%). Thus, for the 30 programs tested, MemSafe ensured complete temporal safety with a modest 10% increase in the average overhead for achieving spatial safety alone.

Finally, the additional cost of ensuring temporal safety with MemSafe is a significant reduction in the cost of achieving temporal safety with MSCC. On MSCC's set of reported benchmarks, the additional cost of ensuring temporal safety with MemSafe (1%) is a reduction in the additional runtime cost required for MSCC to ensure tempral

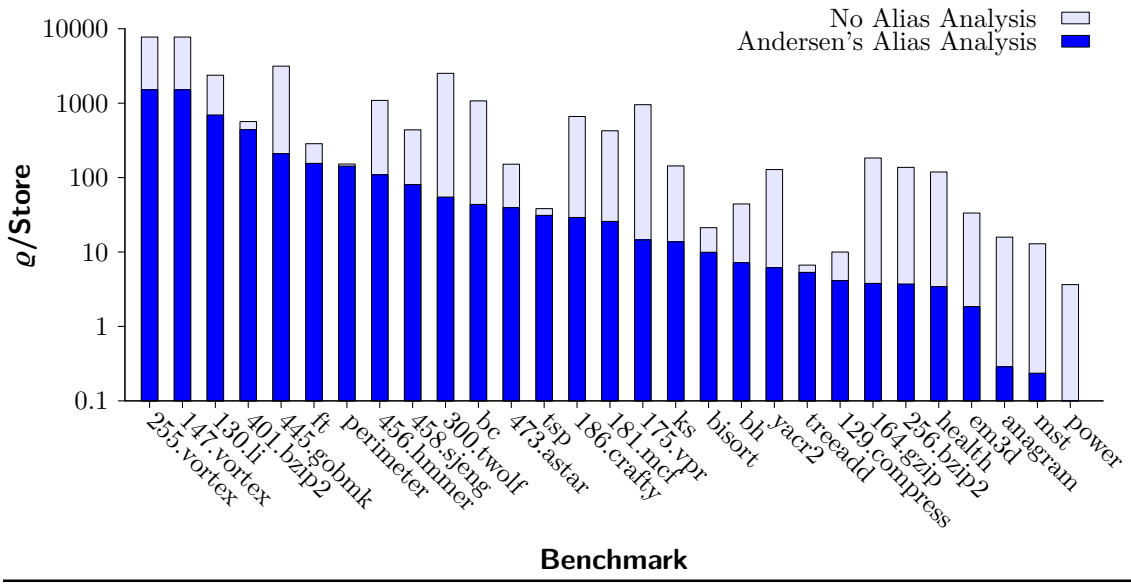


Figure 7.4: Effect of aliasing. The average number of ϱ-nodes modifiable by each pointer store is shown with Andersen's alias analysis versus no alias analysis.

safety (62%) by a factor of 62. For the set of programs that were successfully compiled with MSCC on the same platform as MemSafe (shown in Figure 7.1), the additional cost of ensuring temporal safety with MemSafe (13%) is a reduction in the additional runtime cost required for MSCC to ensure temporal safety (114%) by a factor of nearly 9. Thus, while the overheads for MemSafe are much lower than that of MSCC, the difference in the additional overhead required to achieve temporal safety is further evidence that by modeling temporal errors as spatial errors, MemSafe's optimizations are effective tools for reducing the additional cost of temporal safety.

7.3 Static analysis

The "Checks," "Opts.," and "DFPG" columns of Table 7.5 describe results related to MemSafe's whole-program analysis. First, the "Checks" column shows the static

Benchmark		Size		Checks (%)			Opts. (%)			DFPG	
Suite	Program	LOC	Derefs	PBC	OBC	MARC	DDO	TDO	NDO	Invalid (%)	ϱ/Store
Olden	bh	2073	284	19.01	0.00	1.76	27.11	70.07	52.11	0.00	7.19
	bisort	350	76	9.21	0.00	0.00	35.53	64.47	55.26	0.00	9.93
	em3d	688	187	31.55	0.00	3.21	7.49	82.35	57.75	0.00	1.84
	health	502	236	22.46	0.00	0.00	15.25	84.32	62.29	0.00	3.42
	mst	428	57	19.30	0.00	5.26	12.28	77.19	63.16	0.00	0.24
	perimeter	484	258	19.77	0.00	0.00	0.00	100.00	80.23	0.00	142.61
	power	622	285	37.89	0.00	0.00	22.46	75.79	39.65	0.00	0.00
	treeadd	245	26	39.47	0.00	0.00	0.00	100.00	60.53	0.00	5.33
	tsp	582	194	15.98	0.00	0.00	31.96	68.04	52.06	0.00	31.07
	average	*716*	*178*	*23.85*	*0.00*	*1.14*	*16.90*	*80.25*	*58.12*	*0.00*	*22.40*
PtrDist	anagram	650	113	33.63	0.00	0.00	21.24	52.21	45.13	0.00	0.29
	bc	7297	3927	15.76	3.79	1.12	32.85	57.70	50.27	8.99	43.62
	ft	1766	246	30.49	0.00	0.00	24.80	70.33	44.72	3.92	155.43
	ks	782	239	28.03	0.00	0.00	27.62	49.37	44.35	0.00	13.71
	yacr2	3986	1000	34.70	5.00	3.90	34.60	51.20	26.80	4.85	6.15
	average	*2896*	*1105*	*28.52*	*1.76*	*1.00*	*28.22*	*56.16*	*42.25*	*3.55*	*43.84*
SPEC'95	099.go	29246	16632	57.54	0.00	5.96	25.44	11.06	11.06	0.00	–
	129.compress	1934	232	16.38	0.00	4.74	40.95	37.93	37.93	0.00	4.13
	130.li	7597	4905	14.92	0.00	0.06	27.26	70.89	57.76	0.00	694.18
	147.vortex	67202	25135	7.35	0.25	0.04	34.36	64.25	58.25	13.18	1511.59
	average	*26495*	*11726*	*24.05*	*0.06*	*2.70*	*32.00*	*46.03*	*41.25*	*3.30*	*736.63*
SPEC'00	164.gzip	8605	1499	21.35	0.47	4.34	44.70	30.02	29.62	0.96	3.79
	175.vpr	17729	5386	21.59	0.32	2.32	22.08	71.67	54.01	7.07	14.52
	181.mcf	2412	534	7.30	0.19	1.31	23.60	69.10	67.79	9.61	25.74
	186.crafty	24975	7579	38.49	11.11	0.01	15.64	46.15	45.86	23.77	29.08
	255.vortex	67213	25134	7.35	0.25	0.04	34.37	64.24	58.24	13.18	1511.61
	256.bzip2	4649	1254	43.46	0.40	3.03	41.23	14.83	12.28	1.12	316.95
	300.twolf	20459	11741	16.73	0.88	0.25	20.82	72.24	62.21	9.39	3.71
	average	*20863*	*7590*	*22.32*	*1.95*	*1.61*	*28.92*	*52.61*	*47.14*	*9.30*	*54.72*
SPEC'06	401.bzip2	8293	4013	17.29	8.07	1.20	12.09	75.18	69.42	27.83	440.48
	445.gobmk	197215	27614	38.51	8.52	1.96	19.49	41.80	40.05	12.70	209.98
	456.hmmr	35992	7582	27.13	8.76	1.58	18.40	65.63	52.89	14.21	108.66
	458.sjeng	13847	5832	26.29	2.54	0.22	28.21	48.47	45.28	18.37	80.29
	473.astar	5842	1873	7.90	2.62	0.32	19.38	75.71	72.40	18.91	39.38
	average	*52238*	*9383*	*23.42*	*6.10*	*1.06*	*19.51*	*61.36*	*56.01*	*18.40*	*133.89*
Average		**18394**	**5136**	**24.23**	**1.77**	**1.42**	**24.04**	**62.07**	**50.31**	**6.48**	**177.68**

Table 7.5: Static results with whole-program analysis. Program size is measured in lines of code and the number of static dereferences. The static number of required checks and optimizations are measured as a percentage of dereferences. The DFPG is measured by the percentage of nodes reachable from *invalid* and the average number of ϱ-nodes modifiable by each pointer store.

Benchmark		Size		Checks (%)			Opts. (%)		
Suite	Program	LOC	Derefs	PBC	OBC	MARC	DDO	TDO	NDO
Olden	bh	2073	284	42.96	33.10	1.76	27.11	36.97	28.17
	bisort	350	76	46.05	38.16	0.00	35.53	26.32	18.42
	em3d	688	187	40.11	20.86	3.21	7.49	62.57	49.20
	health	502	236	60.17	38.14	0.00	15.25	46.61	24.58
	mst	428	57	31.58	14.04	5.26	12.28	64.91	50.88
	perimeter	484	258	98.06	89.53	0.00	0.00	10.47	1.94
	power	622	285	48.42	11.23	0.00	22.46	64.56	29.12
	treeadd	245	26	63.16	31.58	0.00	0.00	68.42	36.84
	tsp	582	194	64.95	59.28	0.00	31.96	8.76	3.09
	average	*716*	*178*	*55.05*	*37.32*	*1.14*	*16.90*	*43.29*	*26.92*
PtrDist	anagram	650	113	33.63	0.00	0.00	21.24	52.21	45.13
	bc	7297	3927	39.62	31.65	1.12	32.85	30.00	26.41
	ft	1766	246	30.49	0.00	0.00	24.80	70.33	44.72
	ks	782	239	34.73	6.69	0.00	27.62	42.68	37.66
	yacr2	3986	1000	42.00	22.40	3.90	34.60	33.90	19.50
	average	*2896*	*1105*	*36.09*	*12.15*	*1.00*	*28.22*	*45.82*	*34.68*
SPEC'95	099.go	29246	16632	57.93	0.42	5.96	25.44	10.67	10.67
	129.compress	1934	232	21.98	5.60	4.74	40.95	32.33	32.33
	130.li	7597	4905	50.38	48.60	0.06	27.26	22.32	22.30
	147.vortex	67202	25135	50.10	49.13	0.04	34.36	15.50	15.50
	average	*26495*	*11726*	*45.10*	*25.94*	*2.70*	*32.00*	*20.21*	*20.20*
SPEC'00	164.gzip	8605	1499	23.08	2.54	4.34	44.70	28.09	27.89
	175.vpr	17729	5386	53.47	46.05	2.32	22.08	26.90	22.13
	181.mcf	2412	534	25.84	19.10	1.31	23.60	50.19	49.25
	186.crafty	24975	7579	57.63	31.73	0.01	15.64	26.82	26.72
	255.vortex	67213	25134	50.10	49.15	0.04	34.37	15.49	15.49
	256.bzip2	4649	1254	45.06	3.67	3.03	41.23	11.56	10.69
	300.twolf	20459	11741	58.08	48.09	0.25	20.82	25.14	20.86
	average	*20863*	*7590*	*44.75*	*28.62*	*1.61*	*28.92*	*26.31*	*24.72*
SPEC'06	401.bzip2	8293	4013	83.85	79.79	1.20	12.09	3.46	2.87
	445.gobmk	197215	27614	58.85	31.96	1.96	19.49	19.95	19.70
	456.hmmr	35992	7582	63.60	55.01	1.58	18.40	21.79	16.42
	458.sjeng	13847	5832	48.13	28.07	0.22	28.21	23.56	23.44
	473.astar	5842	1873	54.40	51.90	0.32	19.38	26.70	25.89
	average	*52238*	*9383*	*61.77*	*49.35*	*1.06*	*19.51*	*19.09*	*17.66*
Average		**18394**	**5136**	**49.28**	**31.58**	**1.42**	**24.04**	**32.63**	**25.26**

Table 7.6: Static results with separate compilation. Program size is measured in lines of code and the number of static dereferences. The static number of required checks and optimizations are measured as a percentage of dereferences.

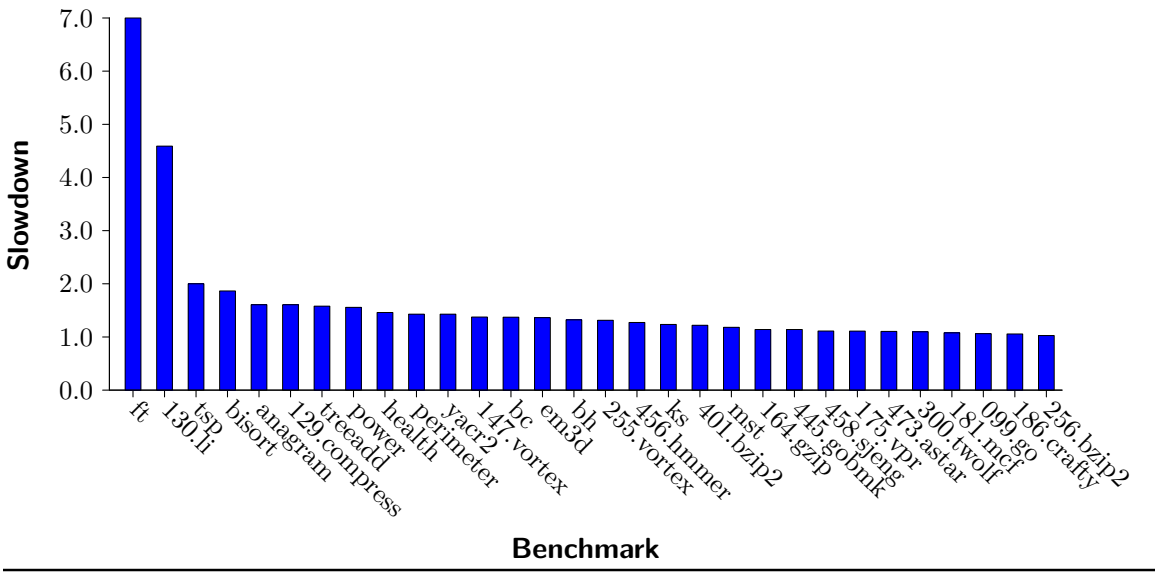


Figure 7.5: Compile-time slowdown. For each program, slowdown is computed as the ratio of the compilation time required by MemSafe to that of the base LLVM compiler using the default set of optimizations.

number of required checks, organized by check type, as a percentage of the static number of total pointer dereferences. Second, the "Opts." column shows the static number of checks (i.e., `PBC` and `OBC`) that were eliminated by MemSafe's optimizations, organized by optimization type. Finally, the "DFPG" column summarizes the DFPG with the percentage of nodes reachable from the node representing the *invalid* pointer and with the $\varrho/store$ quantity. The former indicates the portion of pointers that may refer to temporally invalid objects, and the latter indicates the average number of loaded memory locations that each pointer store may potentially modify. Thus, these two quantities are a static estimate of the uncertainty in pointer data-flow. Figure 7.4 demonstrates that Andersen's alias analysis [4] is often capable of reducing $\varrho/store$ by several orders of magnitude.

Table 7.6 shows the static number of required checks and the number of checks that were eliminated with MemSafe's optimizations when MemSafe's whole-program

analysis was disabled. Thus this represents the results of MemSafe's static analysis in a separate compilation development environment where it is unable to make use of interprocedural information. Since the DFPG is only useful for interprocedural optimization, Table 7.6 does not report the size or complexity of the DFPG.

Finally, Figure 7.5 shows MemSafe's compile-time slowdown for each program. Slowdown is computed as the ratio of the compilation time required by MemSafe to that of the base LLVM compiler using default optimizations. In general, the compile-time requirements of MemSafe are modest. For 93% of the benchmarked programs (i.e., 28 out of 30), MemSafe was able to ensure memory safety with an increase in compile-time by less than a factor of two. The compilation time required by *ft* and *130.li* surpassed this threshold due to the time required to query alias analysis for each pair of pointer load and store instructions, which is needed for inserting ϱ-functions (see the algorithm for ϱ-function insertion in Section 4.1.2.1). Despite these two programs, MemSafe's average increase in compile-time over all 30 benchmarked programs was 62%.

Chapter 8

Related Work

Most prior techniques related to the enforcement of memory safety were presented in Section 3.2 after having described the various types of spatial and temporal memory errors. Therefore, this chapter will not repeat that content, but it will discuss additional details for methods capable of detecting both spatial and temporal violations as well as techniques that can only detect one type of error. This chapter also presents a discussion of previous work related to MemSafe's data-flow analysis.

8.1 Spatial and Temporal Safety

While generally not enforcing complete memory safety, several methods are capable of detecting both spatial and temporal errors. Purify [43] operates on binaries, but only ensures the safety of heap-allocated objects. Yong and Horwitz [83] present a similar approach and improve its cost with static analysis, but this method only checks store operations. Safe C [9] ensures complete safety but is incompatible due to its use of fat-pointers. Patil and Fischer [68] address these issues by maintaining disjoint metadata and performing checks in a separate "shadow process," but this requires

an additional CPU. CCured [64] utilizes a type system to eliminate checks for safe pointers and reduce metadata bookkeeping. However, CCured's use of fat-pointers causes compatibility issues, and some programs require code modifications to lower cost. MSCC [82] is highly compatible and complete but is unable to handle some downcasts. Fail-Safe C [66] maintains complete compatibility with ANSI C but incurs significant runtime overhead. Finally, Clause et al. [20] describe an efficient technique for detecting memory errors, but it requires custom hardware.

8.2 Spatial Safety

Methods that primarily detect bounds violations are numerous. Notable is the work by Jones and Kelly [49] since it maintains compatibility with pre-compiled libraries. However, this method has high overhead and results in false positives. Ruwase and Lam [74] extend this method to track out-of-bounds pointers to avoid false positives. Additionally, Dhurjati and Adve [30] utilize Automatic Pool Allocation [55] to improve cost, and Akritidis et al. [3] constrain the size and alignment of allocated regions to further improve cost. However, these methods do not detect temporal violations and are unable to detect sub-object overflows.

HardBound [28] is a hardware-assisted approach for ensuring spatial safety with low overhead. This method encodes fat-pointers in a special "shadow space" and provides architectural support for checking and propagating metadata. SoftBound [62] is a related technique that records pointer metadata in disjoint data structures similar to MemSafe's representation. However, while these methods ensure complete

spatial safety, they do not ensure temporal safety, and HardBound requires custom hardware to achieve low overhead.

8.3 Temporal Safety

Few methods are designed primarily for detecting temporal violations. Dhurjati and Adve [29] describe a technique based on the Electric Fence [69] `malloc` debugger: Their system assigns a unique virtual page to every dynamically allocated object and relies on hardware page protection to detect dangling pointer dereferences. This approach is improved with Automatic Pool Allocation [55] and a customized address mapping. However, this method does not detect spatial violations and only detects temporal violations of heap objects. CETS [63] inserts temporal safety checks before pointer dereferences and utilizes an efficient lock-and-key mechanism, instead of hash tables, for accessing the required temporal metadata. However, this method also does not detect spatial violations and must be combined with an existing spatial safety mechanism in order to guarantee complete temporal safety.

8.4 Software Debugging Tools

While not intended for deployment in production-quality applications, automated debugging tools can be used to detect some memory safety violations during software development and testing. Valgrind [65] is a heavyweight dynamic binary instrumentation framework providing the Memcheck [75] tool for debugging memory accesses and

leaks, and Mudflap [33] is a compiler approach for debugging memory accesses implemented in the GCC [40] compiler infrastructure. However, these tools are incapable of ensuring complete spatial and temporal memory safety and incur significant runtime overheads. For example, both Memcheck and Mudflap are unable to detect spatial safety errors where an out-of-bounds pointer to one object happens to fall within bounds of another object. They are also unable to detect temporal safety errors when the runtime system reallocates memory to a previously deallocated location. Moreover, Memcheck does not aim to ensure spatial or temporal safety for stack-allocated objects and increases runtime by a factor of 10–30.

8.5 Other Methods of Memory Protection

Several methods utilize software checks to enforce various security-related policies. Abadi et al. [1] describe a technique to prevent software attacks by enforcing control-flow integrity. Similarly, Castro et al. [17] enforce data-flow integrity with an analysis based on reaching definitions, and WIT [2] enforces write-integrity by ensuring each write operation accesses an object from a static set of legally modifiable objects. Although these techniques are capable of preventing many memory access violations, they do not ensure complete spatial and temporal safety.

DieHard [12] is a memory allocator capable of preventing many heap-related errors. It uses random object placement within a larger-than-normal heap to prevent invalid frees and probabilistically avoid heap buffer overflows. However, this method is incapable of ensuring complete spatial and temporal safety.

Other methods seek to provide minimal memory protection guarantees to programs executed on systems lacking hardware virtual memory. Simpson et al. [76] developed a low-overhead method for achieving memory segmentation using compiler-inserted runtime checks. Like paging, segmented virtual memory is a common approach for providing coarse-grained memory protection, and Appendix B shows how MemSafe's metadata propagation rules can be modified to achieve segment protection instead of full memory safety. In another method, Biswas et al. [14] developed a technique for avoiding out-of-memory errors with compiler-inserted runtime checks, memory reuse, and the compression of unused data. Finally, Middha et al. [61] developed a similar method for avoiding out-of-memory errors in embedded systems by sharing stack space among the executing tasks of multitasking workloads.

8.6 SSA Extensions

Various methods have extended SSA [26] to incorporate alias information. IPSSA [59] is an interprocedural, Gated SSA [67] that uses alias analysis to replace indirect stores with ϕ-like functions whose semantics are similar to our ϱ-function. However, IPSSA represents *all* indirect stores as direct assignments, whereas MemSafe's ϱ-function is only used for *pointer* stores. Other extensions include the χ- and μ-extensions [19], which model may-def and may-use information, but unlike the ϱ-function, do not keep track of the defining values. Finally, Cytron and Gershbein [25] describe a demand-driven algorithm that incrementally incorporates alias information with SSA to avoid a large increase in program size.

Chapter 9

Future Work

MemSafe is a whole-program compiler analysis and transformation for ensuring the spatial and temporal memory safety of C programs. Although it is complete, compatible, does not require any manual code modifications, and has lower runtime cost than other methods, MemSafe is not without limitations, and additional work can enhance it further. This chapter identifies several aspects of MemSafe in which additional research is warranted. These areas include (1) MemSafe's performance overheads and the scope of its evaluation, (2) the specification and verification of MemSafe's transformations and optimizations, and (3) additional uses of MemSafe's core technology for tasks not directly related to ensuring the memory safety of C programs.

9.1 Performance Enhancements and Evaluation

Although MemSafe's performance overheads are, on average, lower than that of existing complete and automatic methods that are capable of detecting both spatial and temporal errors, they are not yet low enough for MemSafe to be used with performance-

critical applications. In addition, while MemSafe has been evaluated on a set of thirty benchmark programs and two large and widely-used open source applications, performance overheads are, in general, application-dependent, and MemSafe's current results may be a poor indicator of its results for other commonly-used applications. The following is a list of future work related to lowering MemSafe's performance overheads and improving its evaluation.

1. The implementation alternatives discussed in Section 6.2 for constructing MemSafe's global metadata facilities (i.e., $\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$, and $\mathcal{R}_{\mathcal{F}}$) should each be evaluated in terms of their runtime overhead and memory usage. MemSafe's prototype implementation should then be updated to use the most efficient data structures, which would lower the overall cost of maintaining the required metadata.

2. Additional optimizations for lowering MemSafe's runtime overhead and memory usage should be considered. The six optimizations presented in Section 5.2 are not exhaustive, and additional optimizations could have a significant impact on MemSafe's average performance overheads. An effective means of discovering potential optimization opportunities is to instrument the applications exhibiting high runtime with profiling instructions that help to determine the source of their overhead.

3. Sophisticated static analysis techniques should be investigated for lowering MemSafe's runtime overhead and memory usage. In particular, while MemSafe's static analysis currently only relies on a flow- and context-insensitive alias analysis, more precise analyses [e.g. 13] may yield fewer runtime checks and

metadata propagation.

4. MemSafe's performance overheads should be evaluated on multithreaded programs. Given the current proliferation of multi-core and multiprocessor CPUs, the multithreading paradigm has emerged as the dominant concurrent programming and execution model. While Section 4.4 ensures MemSafe's metadata facilities and runtime checks are thread safe, the runtime cost of the required thread synchronization primitives has yet to be evaluated in a multithreaded environment.

9.2 Specification and Verification

In addition to lowering MemSafe's performance overheads and improving its evaluation, additional work would be useful for increasing the number of programs to which MemSafe can be applied and guaranteeing the correctness of MemSafe's transformations. The following is a list of future work related to the specification of MemSafe's metadata propagation rules and their verification.

1. Additional metadata propagation rules should be devised for handling inline assembly instructions and self-modifying code. Although programs utilizing these features are uncommon, extending MemSafe with the necessary rules would increase the number of programs to which MemSafe could be applied. However, as mentioned in Section 6.4, MemSafe's optimizations would likely be of little benefit to such programs, so the runtime cost of ensuring their memory safety is

expected to be high.

2. MemSafe's program transformations and optimizations should be formally verified in order to prove their correctness. Related work in this regard has provided formalisms of spatial safety [62, 64], temporal safety [63], and static memory safety analyses based on the results of flow- and context-insensitive alias analysis [22], which could serve as a starting point.

9.3 Additional Uses

Although MemSafe has been demonstrated to be an effective tool for ensuring the memory safety of C programs, the technology developed for MemSafe could be useful in other contexts. The following is a list of future work that could potentially reuse some or all of MemSafe's main components for performing tasks not directly related to ensuring the memory safety of C programs.

1. MemSafe should be evaluated on its ability to prevent the exploitation of security vulnerabilities and to thwart attempts at malicious attacks. As mentioned in Section 1, many, if not all, security vulnerability exploits rely on some form of memory access violation (e.g., a buffer overflow error) in order for an attacker to perform a malicious act. Since MemSafe can ensure a program's memory safety, it could also be used as a defense against such attacks.

2. The current implementation of MemSafe is capable of ensuring memory safety for programming languages other than C, and its effectiveness in doing so should

be evaluated. Since LLVM's intermediate representation is source language independent, MemSafe can ensure the memory safety of any language that can be compiled by LLVM. LLVM's compiler frontend currently supports several languages in the C family of programming languages including C, C++, Objective-C, and Objective-C++.

3. While MemSafe's analysis and transformation passes operate on LLVM's intermediate representation, it is possible that MemSafe could be made to ensure memory safety by directly analyzing application binaries. Ensuring the safety of binaries would require a new set of metadata propagation rules for assembly language instructions, but given the low-level nature of LLVM's instruction set, this is likely an achievable task. However, without accurate source-level information, MemSafe's optimizations would be ineffective, and MemSafe's spatial safety guarantees would be limited to object-level safety. Sophisticated binary translation techniques and composite type reconstruction methods could improve MemSafe's ability to directly ensure the memory safety of binaries.

4. Several of MemSafe's key components—in particular, the use of the *invalid* pointer to represent memory deallocation and the use of the ϱ-function to represent indirect pointer assignments—may have general applicability, and their usefulness should be explored outside the context of ensuring memory safety. It is conceivable that MemSafe's $DFPG$, which incorporates both of these tools, could simplify or enable other compiler optimizations as it does for optimizations of memory safety checks and metadata.

Chapter 10

Conclusion

This dissertation describes MemSafe, an automatic compiler analysis and transformation technique that is capable of ensuring the memory safety of C programs at runtime. MemSafe transforms a program such that it detects all spatial and temporal memory violations before they occur, while remaining compatible with existing code and requiring lower runtime overhead than previous techniques. The major contributions of this dissertation are summarized below.

The motivation behind this research is the realization that use of the C programming language is likely going to remain common despite the many well-known memory safety violations it allows. The features of C that make it a desirable language for systems-level programming—including weak typing, low-level access to computer memory, and unchecked pointer use—are the very features whose misuse cause the variety of hard-to-detect memory errors that plague today's software. The detection or prevention of memory errors in C is a challenging problem because, while these violations often cause a program to crash immediately, their symptoms frequently go undetected long after they occur, resulting in data corruption and incorrect results and making software debugging particularly difficult.

As evidence of this problem, a variety of methods exist for retrofitting C programs with software checks to detect memory errors at runtime. However, these techniques generally suffer from one or more practical drawbacks that have thus far limited their adoption. These weaknesses include (1) the inability to detect all spatial and temporal violations, (2) the use of incompatible metadata (the bounds information required for performing runtime checks), (3) the need for manual code modifications, and (4) the tremendous runtime cost of providing complete safety.

MemSafe addresses the above drawbacks and ensures the memory safety of C programs by utilizing a whole-program compiler analysis and transformation that performs a limited amount of static analysis to prove memory safety whenever possible and inserts runtime checks to ensure the safety of the remaining memory accesses. MemSafe is complete, compatible, requires no manual code modifications, and generally has lower runtime overhead than prior techniques achieving a similar level of safety. In this regard, MemSafe makes several novel contributions to the research community, which are each summarized below.

First, MemSafe uniformly handles all memory violations by modeling temporal errors as spatial errors. By doing so, the use of separate mechanisms for detecting temporal errors is no longer required. In particular, MemSafe avoids the drawbacks associated with using conservative garbage collection and explicit temporal checks. MemSafe achieves this uniformity of errors by modeling memory deallocation as an explicit pointer assignment, thereby enabling spatial safety mechanisms to be reused to enforce temporal safety.

Second, MemSafe captures the most salient features of object and pointer meta-

data in a hybrid spatial metadata representation. MemSafe's metadata representation exploits the strengths of both approaches, while simultaneously avoiding their weaknesses, in order to ensure its completeness and compatibility. Additionally, object and pointer metadata create a synergy that allows properties related to temporal safety to be represented using spatial metadata.

Finally, MemSafe uniformly handles pointer data-flow in a representation that simplifies several performance-enhancing optimizations. Unlike previous methods that require runtime checks for all pointer dereferences and the expensive propagation of metadata at every pointer assignment, MemSafe eliminates redundant checks and the propagation of unused metadata. MemSafe achieves this uniformity of pointer data-flow by modeling indirect pointer assignments as explicit pointer assignments, which enables MemSafe's whole-program analysis and optimizations to reason solely about the ways in which pointers are defined.

Experimental results indicate that MemSafe is capable of detecting memory safety violations in real-world programs with a lower runtime overhead than previous methods. Results show that MemSafe detects all known memory errors in multiple versions of two large and widely-used open source applications as well as six programs from a benchmark suite specifically designed for the evaluation of error detection tools. MemSafe enforces complete safety with an average overhead of 88% on 30 widely-used performance evaluation benchmarks. In comparison with previous work, MemSafe's average runtime overhead for one common benchmark suite (29%) is a fraction of that associated with the previous technique (133%) that, until now, had the lowest overhead among all existing complete and automatic methods that are capable of

detecting both spatial and temporal violations.

Since MemSafe's performance overheads cannot necessarily be considered "low," MemSafe is likely only permanently deployable in applications where memory safety is the primary design concern. In practice, it has been observed that many runtime checks can be avoided with MemSafe's simple optimizations, and for safety-critical applications, MemSafe's moderate runtime overheads can be an acceptable trade-off compared to redesigning systems in a safe language. However, for performance-critical applications, MemSafe is primarily useful as a dynamic bug detection tool.

Appendix A

Metadata Propagation for the C Standard Library

The metadata propagation rules presented in Section 4.3 are sufficient for MemSafe to ensure the spatial and temporal memory safety of many C programs. However, the C standard library routines provide programmers with the ability to manipulate memory and pointers in ways that are not covered by those metadata rules. In this appendix, MemSafe is extended with the rules required for a program to correctly propagate metadata when using the C standard library routines. Specifically, Section A.1 describes metadata propagation for the memory copying functions of `string.h`, and Section A.2 describes metadata propagation for the variadic function macros of `stdarg.h`.

A.1 Memory Copying Functions of `string.h`

The memory copying functions (e.g. `memcpy` and `memmove`) defined in the `string.h` standard library generally copy a specified number of bytes from a location indicated by a source pointer *src* to the location referred to by a destination pointer *dest*.

Although these procedures result in multiple read and write operations, MemSafe only needs to perform bounds checks for the source and destination buffers once before the operation begins.

However, any in-memory pointer values that are located in the source buffer and copied to the destination buffer must also have their associated pointer metadata copied. Recall that pointer metadata that is associated with in-memory pointers is maintained in the pointer metadata facility $\mathcal{R}_{\mathcal{P}}$. Thus, any mapped address in $\mathcal{R}_{\mathcal{P}}$ that is within the range $[base, bound)$ of the source buffer, must have its metadata copied and associated with a new address that is located at a distance of $dest - src$ bytes from the original address. This is achieved with the following metadata rule for memory copying functions. Numbered lines indicate original code.

Metadata Rule A.1—Memory copying functions:

```
1: memcpy(v, u, size);
```

$$S = \{(ptr, \langle addr, id \rangle_{*\texttt{ptr}}) \in \mathcal{R}_{\mathcal{P}} : base_{src} \leq ptr < bound_{src}\} \quad \text{(A.1.1)}$$

$$D = \{(ptr + (dest - src), \langle addr, id \rangle_{*\texttt{ptr}}) : (ptr, \langle addr, id \rangle_{*\texttt{ptr}}) \in S\} \quad \text{(A.1.2)}$$

$$\mathcal{R}_{\mathcal{P}} = \mathcal{R}_{\mathcal{P}} \cup D \quad \text{(A.1.3)}$$

In this example, the definition of S selects the pointer metadata associated with the source buffer that must be copied (A.1.1), and the definition of D associates the metadata with a new address within bounds of the destination buffer (A.1.2). Finally, $\mathcal{R}_{\mathcal{P}}$ is updated to contain the copied metadata (A.1.3). To avoid the runtime overhead of performing the metadata copy, MemSafe attempts to infer if the source buffer contains any in-memory pointer values by reasoning about its type and usage. Although this may lead to the pointer metadata facility not being properly updated,

instances of in-memory pointers being coped with the `string.h` functions have been observed to be rare in practice.

A.2 Variadic Function Macros of `stdarg.h`

The variadic function macros defined in `stdarg.h` enable functions to process an unspecified number of optional arguments in addition to their fixed number of mandatory arguments. Variadic functions are convenient because they are able to accept a different number of arguments per invocation. For example, the `printf` function accepts one mandatory argument, the format string, and a variable number of optional arguments, which are the values to print. Declaring a function to be variadic simply involves placing an ellipsis ('...') as the last argument in the function's argument list.

Within the body of a variadic function, optional arguments are processed sequentially in the order in which they were passed to the function. To do so, a program initializes a pointer of type `va_list` to point to the front of the optional argument list using the `va_start` macro. The program can then process each argument with successive calls to the `va_arg` macro. For example, the first call to `va_arg` returns the first optional argument, the second call returns the next optional argument, etc. Finally, a program calls the `va_end` macro to indicate it is finished processing the optional arguments.

Propagating the pointer metadata associated with the optional pointer arguments of a variadic function is straightforward and follows the metadata rules presented in Section 4.3 for function calls. Recall that within the body of a function, MemSafe

retrieves the pointer metadata of mandatory arguments from the function metadata facility based on their position in the function's argument list. Optional arguments are handled in the same way, and MemSafe retrieves their pointer metadata according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule A.2—Variadic function arguments:

```
1: void f(int x, ...)  {
2:   va_list ap;
3:   int     *p, i;
4:   ...
5:   for(i=0; i < x; i++) {
6:     p = va_arg(ap, int*);
```

$$\langle addr, id \rangle_p = fmd(\langle \texttt{\&f}, i+1 \rangle) \tag{A.2.1}$$

```
7:     ...
8:   }
9: }
```

In this example, function f is declared to be variadic. Without loss of generality, the value of argument x is assumed to indicate at runtime the number of optional arguments that are passed to f, and it is also assumed that each optional argument is a pointer to type `int`. The loop beginning in line 5 iterates over the number of arguments using an induction variable i, and pointer p is assigned the location of the next argument using the `va_arg` macro on each iteration of the loop. After p is assigned, MemSafe retrieves from $\mathcal{R}_{\mathcal{F}}$ its associated pointer metadata by performing the lookup operation $fmd(\langle \texttt{\&f}, i+1 \rangle)$, where the argument's offset in the function's argument list is given by $i+1$ since x is the first argument of the function (A.2.1).

While the above rule ensures that optional pointer arguments inherit the correct pointer metadata, MemSafe must also define metadata associated with the argument

list itself. Since the argument list is allocated contiguously in memory, it is essentially an array, and MemSafe must insert pointer bounds checks to ensure each `va_arg` operation falls within bounds of the list. Typically, a program ensures each access of the argument list is within bounds by counting the number of arguments at runtime. For example, the `printf` function counts the number of format specifiers in its format string argument to determine the number of optional arguments to process. However, since the optional arguments can be of different types at runtime, MemSafe cannot rely on their number, and must utilize the base and bound addresses of the argument list to perform the required check. MemSafe creates the pointer metadata of the argument list according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule A.3—Argument list bounds:

1: `int x;`
2: `...`

$$\langle addr, id\rangle = \langle addr \in A, id_f\rangle \tag{A.3.1}$$

$$M[addr] \leftarrow \langle 0, \textstyle\sum_{i=1}^{n} \texttt{sizeof}(a_i), id_f\rangle \tag{A.3.2}$$

$$\begin{aligned}\mathcal{R}_{\mathcal{F}} = \mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, -1\rangle, fmd(\langle \texttt{\&f}, -1\rangle))\} \\ \cup \{(\langle \texttt{\&f}, -1\rangle, \langle addr, id\rangle)\}\end{aligned} \tag{A.3.3}$$

3: `f(x,` $a_1 \ldots a_n$`);`

In this example, the variadic function f is called in line 3. However, before the call MemSafe must define the pointer metadata for the function's argument list. First, MemSafe obtains a new address for storing the bounds information of the list (A.3.1). Then, MemSafe stores the list's base and bound addresses at the specified location, temporally setting the base address to zero and the bound address equal to the total size of the arguments (A.3.2). Within the body of f, this range will be adjust to reflect

the correct starting and ending locations in memory, but here MemSafe simply defines the size of the list. Finally, MemSafe updates $\mathcal{R}_{\mathcal{F}}$ with the list's pointer metadata using a key of $\langle \texttt{\&f}, -1 \rangle$, where the index -1 is reserved for a function's argument list pointer (A.3). Refer to Section 4.3 for additional information regarding the keys used with the function metadata facility.

Within the body of a variadic function, in addition to retrieving the pointer metadata of any optional pointer arguments, a program must also retrieve the metadata of the argument list itself. MemSafe retrieves metadata for the argument list according to the following metadata propagation rule. Numbered lines indicate original code.

Metadata Rule A.4—Argument list pointer:

```
1:  void f(int x, ...)  {
2:    va_list ap;
```

$$\langle addr, id \rangle_{ap} = fmd(\langle \texttt{\&f}, -1 \rangle) \tag{A.4.1}$$

$$M[addr_{ap}] \leftarrow M[addr_{ap}] + \langle \texttt{\&x+}\textbf{sizeof}\texttt{(x)}, \texttt{\&x+}\textbf{sizeof}\texttt{(x)}, 0 \rangle \tag{A.4.2}$$

```
7:    ...
8:  }
```

In this example, after the argument list pointer *ap* is declared, MemSafe retrieves its pointer metadata from the function metadata facility with the $fmd(\langle \texttt{\&f}, -1 \rangle)$ lookup operation (A.4.1). MemSafe then offsets the base and bound addresses of the list by an amount that will locate the list adjacent to and directly following the last mandatory argument (i.e., x) in memory (A.4.2).

Appendix B

Spatial Safety and Segmentation

Since the nontrivial performance overheads of MemSafe could limit its usefulness with some performance-critical applications, MemSafe can be modified to enforce lesser forms of memory protection. In doing so, an acceptable trade-off between the level of memory protection and the resulting increase in runtime may be achieved that is suitable for these programs. In this appendix, Section B.1 considers a modified set of runtime checks and metadata propagation rules that would be required to ensure only spatial safety, and Section B.2 further modifies these rules in order to achieve coarse-grained segment-level protection [76].

B.1 Spatial Safety

Before MemSafe's checks and metadata propagation rules are modified to eliminate the enforcement of temporal safety, recall MemSafe's basic method of detecting temporal memory violations. In order to ensure temporal safety, MemSafe assigns the *invalid* pointer to pointers that no longer refer to a temporally valid object. Since the *invalid* pointer refers to an impossible address range, spatial safety checks for the dereference

of pointers equal to *invalid* are guaranteed to report a safety violation. Thus, MemSafe achieves temporal safety by reusing the techniques of spatial safety. To reduce the runtime cost of ensuring both spatial and temporal safety, MemSafe maintains a combination of pointer and object metadata. The pointer bounds check (`PBC`) uses the pointer metadata to enforce complete spatial and partial temporal safety whereas the object bounds check (`OBC`) uses the object metadata to enforce complete temporal and partial spatial safety, as depicted in Figure 4.2.

For MemSafe to achieve only spatial safety, much of MemSafe's basic method is no longer required. Specifically, MemSafe does not require assignments of the *invalid* pointer to be inserted for modeling memory deallocation, and MemSafe does not require object metadata or the `OBC`, since these are primarily used for enforcing the temporal safety of sub-object references. The remainder of this section describes changes to MemSafe's checks and metadata propagation rules (presented in Sections 4.2–4.3) that are needed in order to enforce only spatial safety.

B.1.1 The required checks and metadata

MemSafe only requires a limited amount of metadata for ensuring spatial safety. As mentioned above, object metadata is not required for ensuring spatial safety so it can be eliminated along with the object metadata facility $\mathcal{R}_{\mathcal{O}}$. In addition, the *id* field of MemSafe's pointer metadata representation is no longer required since it is used to link a pointer to the object metadata of its referent. Thus, to ensure spatial safety, MemSafe requires only pointer metadata in the form of a tuple $\langle base, bound \rangle_p$ that indicates the

range $[base, bound)$ of memory pointer p is permitted to access. MemSafe maintains this pointer metadata in memory and allocates at runtime an address $addr_p$ from a set of unused address A for storing $\langle base, bound \rangle_p$. These values are stored to memory with an explicit dereference operation, represented by $M[addr_p] \leftarrow \langle base, bound \rangle_p$, where $M[addr_p]$ holds the value at address $addr_p$ in memory.

The pointer bounds check, modified to enforce only spatial safety, utilizes the above pointer metadata. Before every pointer dereference, MemSafe inserts a call to the forcibly inlined procedure defined below. Refer to Section 4.2 for additional details regarding pointer metadata and the original PBC.

Runtime Check B.1—Pointer bounds check (spatial safety):

```
1: inline void pbc(ptr, size, addr) {
2:   ⟨base, bound⟩_ptr ← M[addr]
3:   if ((ptr < base_ptr) || (ptr + size > bound_ptr)) {
4:     signal_safety_violation();
5:   }
6: }
```

B.1.2 Propagation of the required metadata

Having presented the modified pointer bounds check and the metadata MemSafe requires for enforcing spatial safety, this section describes MemSafe's corresponding translations for creating and propagating the required pointer metadata. Similar assumptions are made in the following discussion as were made in Section 4.3. Namely, it is assumed that the program has been transformed into a low-level, typed SSA form that includes MemSafe's ϱ-functions for modeling pointer stores as explicit pointer assignments. However, since temporal safety is not enforced, it is assumed that

assignments of the *invalid* pointer have not been inserted.

B.1.2.1 Memory allocation

For static and automatic memory allocation, metadata is neither created or propagated since such allocation does not define a new pointer value. Whereas to achieve full memory safety, the bounds of the allocated object must be mapped in the object metadata facility, this mapping is no longer required for achieving only spatial safety. However, for dynamic memory allocation, MemSafe creates pointer metadata according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule B.1—Dynamic memory allocation (spatial safety):

```
1:  int *p;
2:  ...
3:  p = (int*) malloc(size);
```

$$\langle base, bound \rangle_p = \begin{cases} \langle \texttt{NULL}, \texttt{NULL} \rangle & \text{if } p = null, \\ \langle \texttt{p}, \texttt{p+}size \rangle & \text{otherwise} \end{cases} \tag{B.1.1}$$

$$M[addr_p \in A] \leftarrow \langle base, bound \rangle_p \tag{B.1.2}$$

In this example, an object of *size* bytes is allocated dynamically by calling `malloc`, and the address returned by `malloc` is assigned to the pointer p. If the value returned by `malloc` is equal to `NULL`, MemSafe sets the base and bound addresses associated with p equal to `NULL`.[1] Otherwise, the pointer metadata is defined such that it refers to the space occupied by the allocated region of memory (B.1.1). Finally, MemSafe obtains

[1]Setting the base and bound addresses associated with the returned pointer to `NULL` is equivalent to defining the base and bound of the region to be that of the *invalid* pointer, as is done in Section 4.3. This ensures that any `PBC` inserted before the dereference of the returned pointer will report a safety violation.

an address $addr_p$ for holding the pointer metadata of p and stores the metadata at this location (B.1.2).

B.1.2.2 Address-of operator

Like dynamic memory allocation, the address-of operator (`&`) creates a pointer to a new location. Therefore, MemSafe must define the metadata of the newly created pointer. MemSafe sets the pointer metadata of a pointer defined in terms of the address-of operator according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule B.2—Address-of operator (spatial safety):

```
1: struct { ...  int array[100]; ...  } s;
2: int *p;
3: ...
4: p = &(s.array[42]);
```

$$M[addr_p \in A] \leftarrow \langle \texttt{\&s.array[0]}, \textbf{sizeof}\texttt{(s.array)} \rangle \tag{B.2.1}$$

In this example, pointer p is assigned the address of an element of the *array* field of structure s. Because the program creates a new pointer, MemSafe obtains a new address for locating the pointer metadata of p and creates and stores the metadata at the specified address (B.2.1).

B.1.2.3 Pointer copies and arithmetic

Pointers defined as simple pointer copies or in terms of pointer arithmetic inherit the pointer metadata of the original pointer. MemSafe sets the pointer metadata of pointers defined by simple assignments according to the following metadata rule.

Numbered lines indicate original code.

Metadata Rule B.3—Pointer copies and arithmetic (spatial safety):

```
1:  int x, *p0, *p1;
2:  ...
3:  p1 = p0 + x;
```

$$addr_{p_1} = addr_{p_0} \tag{B.3.1}$$

In this example, since pointer p_1 is defined in terms of pointer arithmetic, it simply inherits the pointer metadata associated with pointer p_0 (B.3.1).

B.1.2.4 ϱ-functions

MemSafe requires the use of the pointer metadata facility $\mathcal{R}_{\mathcal{P}}$ for disambiguating the value produced by a ϱ-function in order for the returned pointer to inherit the correct metadata. Since MemSafe's pointer metadata representation was able to be simplified by not enforcing temporal safety, the pointer metadata facility must be updated accordingly. The pointer metadata facility, modified to support spatial safety only, maps the address of an in-memory pointer to the address of its pointer metadata and is defined by the partial function:

$$\begin{aligned} pmd : A &\rightarrow A \\ ptr &\mapsto addr_{*\mathtt{ptr}} \end{aligned}$$

where A are memory addresses. For convenience, pmd is also represented more generally as the relation $\mathcal{R}_{\mathcal{P}}$, where $(ptr, addr_{*\mathtt{ptr}}) \in \mathcal{R}_{\mathcal{P}}$.

For pointer loads, MemSafe creates a new definition for the loaded value and assigns it the result of a ϱ-function, which indicates the set of values to which the loaded

value may potentially be equal. For a pointer *ptr* whose pointed-to location is loaded in defining another pointer p, MemSafe retrieves from the pointer metadata facility the required pointer metadata for p with the lookup operation $pmd(ptr)$. MemSafe performs this operation according to the following metadata rule for pointer loads. Numbered lines indicate original code.

Metadata Rule B.4—Pointer loads (spatial safety):

```
1:  int **ptr1, *p0, *p1, ...;
2:  ...
3:  p0 = *ptr1;                  ▷ MemSafe models in-memory data-flow with
4:  p1 = ϱ(a0, b0, ...);           the ϱ-function
```

$$addr_{p_1} = pmd(ptr_1) \tag{B.4.1}$$

In this example, an in-memory pointer is loaded and assigned to pointer p_0. MemSafe then creates a new pointer p_1 and assigns it the result of a ϱ-function indicating the values the in-memory pointer may potentially equal. The address of the pointer metadata for p_1 is retrieved from the pointer metadata facility with the $pmd(ptr_1)$ lookup operation (B.4.1), and all uses of p_0 are replaced with uses of p_1.

For each argument of the ϱ-function, MemSafe saves the location of the pointer's metadata in $\mathcal{R}_\mathcal{P}$ at the point where it is is stored to memory. MemSafe updates the pointer metadata facility for pointer stores according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule B.5—Pointer stores (spatial safety):

```
1:  int **ptr2, *a0;
2:  ...
3:  *ptr2 = a0;      ▷ ptr2 may alias ptr1 from above
```

$$\mathcal{R}_\mathcal{P} = (\mathcal{R}_\mathcal{P} \setminus \{(ptr_2, pmd(ptr_2))\}) \cup \{(ptr_2, addr_{a_0})\} \tag{B.5.1}$$

In this example, pointer ptr_2 is assumed to potentially alias with pointer ptr_1 from the previous example. Thus, pointer a_0 appears in the ϱ-function defined above for pointer p_1 because of the pointer store in line 3. Here, MemSafe maps pointer ptr_2 to the address of the pointer metadata of a_0 in $\mathcal{R}_{\mathcal{P}}$ (B.5.1).

B.1.2.5 `NULL` and manufactured pointers

Pointers defined as `NULL` or as a cast from a non-pointer type must have their base and bound addresses set to `NULL`. An exception to this rule is made for memory-mapped I/O locations, and MemSafe requires a target's backend to specify the base and bound address of all memory-mapped address ranges. MemSafe defines the pointer metadata for `NULL` and manufactured pointers according to the following rule. Numbered lines indicate original code.

Metadata Rule B.6—`NULL` and manufactured pointers (spatial safety):

```
1: int *p;
2: ...
3: p = (int*) 42;
```

$$M[addr_p \in A] \leftarrow \langle \texttt{NULL}, \texttt{NULL} \rangle \tag{B.6.1}$$

In this example, pointer p is defined as a type-cast from the integer 42. Thus, MemSafe obtains an address to store the pointer metadata of p and defines its base and bound addresses to be equal to `NULL` (B.6.1).

B.1.2.6 Function arguments and return values

Just as the pointer metadata facility had to be modified to account for the simplified pointer metadata representation required to achieve spatial safety, the function

metadata facility $\mathcal{R}_{\mathcal{F}}$ must be modified as well. Recall that MemSafe uses $\mathcal{R}_{\mathcal{F}}$ for propagating pointer metadata for pointers passed as arguments to functions or returned from functions. Let *callee* values refer to formal pointer arguments and pointer values that are returned from functions. Similarly, let *caller* values refer to actual pointer arguments and local pointer values to be returned from functions. The function metadata facility maps a *callee* value to the location of the pointer metadata of its corresponding *caller* value and is defined by the partial function:

$$\begin{aligned} fmd &: C \rightarrow A \\ callee &\mapsto addr_{caller} \end{aligned}$$

where C is the set of caller values, A are memory addresses, and *callee* is a tuple $\langle \texttt{\&f}, i \rangle$ indicating the i^{th} pointer associated with function *f*. The function *fmd* is also represented as the relation $\mathcal{R}_{\mathcal{F}}$, where $(callee, addr_{caller}) \in \mathcal{R}_{\mathcal{F}}$. Refer to Section 4.3 for additional details regarding the function metadata facility.

For function calls, MemSafe creates an entry in the function metadata facility for pointer arguments passed to the function. Similarly, MemSafe defines the pointer metadata of a pointer returned from the function call by performing a lookup operation of $\mathcal{R}_{\mathcal{F}}$. MemSafe updates and defines pointer metadata for function calls according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule B.7—Function calls (spatial safety):

```
1: int *p0, *p1;
2: ...
```

$$\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 1 \rangle, fmd(\langle \texttt{\&f}, 1 \rangle))\}) \cup \{(\langle \texttt{\&f}, 1 \rangle, addr_{p_0})\} \tag{B.7.1}$$

```
3: p1 = f(p0);
```

$$addr_{p_1} = fmd(\langle \texttt{\&f}, 0 \rangle) \tag{B.7.2}$$

In this example, a pointer p_0 is passed as an argument to function f and pointer p_1 is assigned the returned value. The return value of f is statically associated with the index "0," and its single pointer argument is given an index of "1." Thus, before the function call, the address of the pointer metadata of p_0 is associated with the tuple $\langle \texttt{\&f}, 1 \rangle$ in $\mathcal{R}_{\mathcal{F}}$ (B.7.1). Similarly, after the call returns, the address of the pointer metadata for p_1 is retrieved from $\mathcal{R}_{\mathcal{F}}$ with the tuple $\langle \texttt{\&f}, 0 \rangle$ (B.7.2).

For the declaration of a function with pointer arguments, MemSafe retrieves from $\mathcal{R}_{\mathcal{F}}$ the address of each incoming pointer's metadata. Similarly, if a function returns a pointer value, MemSafe creates an entry in the function metadata facility for the location of its pointer metadata just before the function returns. MemSafe updates and defines pointer metadata for function declarations according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule B.8—Function declarations (spatial safety):

```
1:  int* f(int *q) {
2:    int *r;
3:    ...
```

$$addr_q = fmd(\langle \texttt{\&f}, 1 \rangle) \tag{B.8.1}$$

```
2:    ...
```

$$\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 0 \rangle, fmd(\langle \texttt{\&f}, 0 \rangle))\}) \cup \{(\langle \texttt{\&f}, 0 \rangle, addr_r)\} \tag{B.8.2}$$

```
3:    return r;
4:  }
```

In this example, pointer q is a formal argument of function f, and pointer r is returned at the end of the procedure. Since q is declared to be the first pointer in the function's argument list, MemSafe retrieves the address of the pointer metadata for q from

$\mathcal{R}_{\mathcal{F}}$ with the tuple $\langle \texttt{\&f}, 1 \rangle$ at the beginning of the procedure (B.8.1). Similarly, since MemSafe statically assigns pointer return values the index "0," the address of the pointer metadata of r is associated with the tuple $\langle \texttt{\&f}, 0 \rangle$ in $\mathcal{R}_{\mathcal{F}}$ before the procedure exits (B.8.2).

B.2 Segmentation

Often times low-end embedded systems do not have any form of hardware memory protection. For these systems, MemSafe can be used to achieve a low-overhead, yet weaker form of spatial memory safety called segment-level protection, or simply memory segmentation [76]. To understand how MemSafe can be used to enforce memory segmentation, it is important to consider the layout of a program's memory.

The address space available to a program is typically organized into four basic segments, which are described below.

1. *Code Segment*: The code segment (sometimes called the "text segment") is the portion of memory allocated to a program for holding its executable instructions. This segment has a statically known size and is typically marked read-only.

2. *Globals Segment*: The globals segment of memory contains the statically allocated objects that are both initialized and uninitialized by the programmer. The globals segment also has a statically known fixed size, but unlike the code segment, it is not read-only. The portion of memory containing uninitialized global data is also known as the "BSS segment," which historically was an acronym for "Block Started by Symbol". The globals segment is usually placed above and adjacent

to the code segment in memory.

3. *Heap Segment*: The heap segment is the portion of memory used for storing a program's dynamically allocated objects. It begins adjacent to the globals segment and grows to larger addresses from there. Objects are allocated to the heap segment by the `malloc` family of standard library functions.

4. *Stack Segment*: The stack segment is reserved for maintaining the program's execution stack. Objects are allocated to the stack segment automatically by a programmer declaring variables as being local to a particular function. The base of the stack is typically located at a high address in memory, and the stack then usually grows towards lower addresses (i.e., towards the heap segment).

Since the code and globals segments are fixed in size, and since the stack segment has a statically known base address, the base and bound addresses of each of the above segments is known at compile-time. The only exception to this statement is the division between the heap and stack segments, which is managed by the `brk` and `sbrk` standard library functions.[2]

For a program to be spatially safe at the granularity of segments, all memory accesses must fall within bounds of a data segment (i.e., the globals, heap, and stack segments). For example, the dereference of a pointer to a stack-allocated object must access memory within bounds of the stack segment, and the dereference of a pointer to a heap-allocated object must access memory within bounds of the heap segment,

[2]The address of the boundary between the heap and stack segments can be determined by the return value of the statement `sbrk(0)`.

etc. MemSafe can enforce segment-level spatial safety using the approach defined above in Section B.1. However, instead of propagating the base and bound addresses of individual objects, MemSafe must propagate the base and bound addresses of the corresponding data segments.

B.2.1 Propagation of the required metadata

This section describes changes to the metadata propagation rules presented above that are required for MemSafe to enforce segment-level protection. Only the metadata rules that define the base and bound addresses of pointer metadata must be updated; all other rules remain the same.

B.2.1.1 Memory allocation

For automatic memory allocation, metadata is neither created or propagated since such allocation does not define a new pointer value. However, recall that MemSafe identifies statically allocated objects by their location in memory (see Section 4.1). For these global pointer values, MemSafe initializes their pointer metadata at the beginning of the `main` procedure to refer to the base and bound addresses of the globals segment. For dynamic memory allocation, MemSafe creates pointer metadata according to the following metadata rule. Numbered lines indicate original code.

Metadata Rule B.9—Dynamic memory allocation (segmentation):

```
1:  int *p;
2:  ...
3:  p = (int*) malloc(size);
```

$$\langle base, bound \rangle_p = \begin{cases} \langle \texttt{NULL}, \texttt{NULL} \rangle & \text{if } p = null, \\ \langle base, bound \rangle_{heap} & \text{otherwise} \end{cases} \tag{B.9.1}$$

$$M[addr_p \in A] \leftarrow \langle base, bound \rangle_p \tag{B.9.2}$$

In this example, MemSafe sets the base and bound addresses associated with p equal to `NULL` if the value returned by `malloc` is equal to `NULL`. Otherwise, the pointer metadata is defined such that it refers to the base and bound addresses of the entire heap segment (B.9.1). Finally, MemSafe obtains an address $addr_p$ for holding the pointer metadata of p and stores the metadata at this location (B.9.2).

B.2.1.2 Address-of operator

MemSafe sets the pointer metadata of a pointer defined in terms of the address-of operator (`&`) according to the following metadata rule for achieving segment-level protection. Numbered lines indicate original code.

Metadata Rule B.10—Address-of operator (segmentation):

```
1: struct { ...  int array[100]; ...  } s;
2: int *p;
3: ...
4: p = &(s.array[42]);
```

$$M[addr_p \in A] \leftarrow \langle base, bound \rangle_{stack} \tag{B.10.1}$$

In this example, pointer p is assigned the address of a stack-allocated object. Thus, MemSafe obtains a new address for locating the pointer metadata of p and stores the base and bound addresses of the entire stack segment at the specified address (B.10.1).

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[5] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. II, United States Air Force Electronic Systems Division, 1972.

[6] *Apache HTTP Server*. Apache Software Foundation. http://httpd.apache.org/.

[7] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[8] ARM Holdings. *ARMv5 Architecture Reference Manual*, 2007.

[9] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[10] David F. Bacon, Perry Cheng, and David Grove. Garbage collection for embedded systems. In *Proceedings of the 4th ACM International conference on Embedded Software*, pages 125–136, 2004.

[11] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[12] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[13] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[14] Surupa Biswas, Thomas Carley, Matthew Simpson, Bhuvan Middha, and Rajeev Barua. Memory overflow protection for embedded systems using run-time checks, reuse, and compression. *ACM Transactions on Embedded Computing Systems*, 5 (4):719–752, November 2006.

[15] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[16] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice & Experience*, 18(9):807–820, 1988.

[17] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[18] Satish Chandra and Thomas Reps. Physical type checking for C. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, 1999.

[19] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the International Conference on Compiler Construction*, pages 253–267, 1996.

[20] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 284–292, 2007.

[21] Jeremy Paul Condit. *Dependent Types for Safe Systems Software*. PhD thesis, University of California, Berkeley, 2007.

[22] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *Proceedings of the International Symposium on Static Analysis*, pages 62–77, 2008.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[24] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 351–366, 2007.

[25] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 36–45, 1993.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[27] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, 2002.

[28] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–114, 2008.

[29] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, 2006.

[30] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, pages 162–171, 2006.

[31] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 144–157, 2006.

[32] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 435–445, 2007.

[33] Frank Ch. Eigler. Mudflap: Pointer use cheking for C/C++. In *Proceedings of the GCC Developers Summit 2003*, pages 57–70, 2003.

[34] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[35] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[36] Björn Franke and Michael O'boyle. Array recovery and high-level transformations for DSP applications. *Transactions on Embedded Computing Systems (TECS)*, 2 (2):132–162, 2003.

[37] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-5769, University of Cambridge, 2004.

[38] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[39] *GNU Core Utilities*. GNU Project. http://www.gnu.org/software/coreutils/.

[40] *GNU Compiler Collection*. GNU Project. http://gcc.gnu.org/.

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

[42] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 272–282, 1990.

[43] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the USENIX Winter Technical Conference*, pages 125–138, 1992.

[44] Institute of Electrical and Electronics Engineers, Incorporated. *IEEE Std 1003.1c-1995*, 1996.

[45] Intel Corporation. *Intel XScale Core: Developer's Manual*, 2004.

[46] International Organization for Standardization. *ISO/IEC 9899: Programming Languages—C*, 1999.

[47] Barnaby Jack. Vector rewrite attack: Exploitable NULL pointer vulnerabilities on ARM and XScale architectures. Technical report, Juniper Networks, Incorporated, 2007.

[48] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, 2002.

[49] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, pages 13–26, 1997.

[50] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[51] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[52] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 2 edition, 1998.

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

[54] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[55] Chris Lattner. *Macroscopic Datastructure Analysis and Optimization*. PhD thesis, University of Illinois, Urbana-Champaign, 2005.

[56] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–87, 2004.

[57] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[58] Shin-Ming Liu, Raymond Lo, and Fred Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 228, 1996.

[59] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 317–326, 2003.

[60] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[61] Bhuvan Middha, Matthew Simpson, and Rajeev Barua. MTSS: Multitask stack sharing for embedded systems. *ACM Transactions on Embedded Computing Systems*, 7(4):46:1–46:37, August 2008.

[62] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[63] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[64] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[65] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[66] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2009.

[67] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

[68] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practice & Experience*, 27(1):87–110, 1997.

[69] Bruce Perens. Electric fence malloc debugger. http://perens.com/FreeSoftware/ElectricFence/.

[70] Dennis M. Ritchie. The development of the C language. In *The second ACM SIGPLAN Conference on the History of Programming Languages*, pages 201–208, 1993.

[71] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, 1995.

[72] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[73] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, 27(2):185–235, 2005.

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

[75] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Technical Conference*, pages 17–30, 2005.

[76] Matthew Simpson, Bhuvan Middha, and Rajeev Barua. Segment protection for embedded systems using run-time checks. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 66–77, 2005.

[77] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring the spatial and temporal safety of C at runtime. In *Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 199–208, 2010.

[78] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[79] *SPEC CPU Benchmarks*. Standard Performance Evaluation Corporation. http://www.spec.org/.

[80] *US-CERT Vulnerability Notes Database*. U.S. Computer Emergency Readiness Team. http://www.kb.cert.org/vuls/.

[81] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, 1998.

[82] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, 2004.

[83] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 307–316, 2003.

[84] Benjamin Zorn. The measured cost of conservative garbage collection. *Software: Practice & Experience*, 23(7):733–756, 1993.