

SAFECODE: A PLATFORM FOR DEVELOPING RELIABLE SOFTWARE IN
UNSAFE LANGUAGES

BY

DINAKAR DHURJATI

B.Tech., Indian Institute of Technology - Delhi, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Many computing systems today are written in weakly typed languages such as C and C++. These languages are known to be “unsafe” as they do not prevent or detect common memory errors like array bounds violations, pointer cast errors, etc. The presence of such undetected errors has two major implications. The first problem is that it makes systems written in these languages unreliable and vulnerable to security attacks. The second problem, which has never been solved for ordinary C, is that it prevents sound, sophisticated static analyses from being reliably applied to these programs. Despite these known problems, increasingly complex software continues to get written in these languages because of performance and backwards-compatibility considerations.

This thesis presents a new compiler and a run-time system called SAFECODE (*Static Analysis For safe Execution of Code*) that addresses these two problems. First, SAFECODE guarantees memory safety for programs in unsafe languages with very low overhead. Second, SAFECODE provides a platform for reliable static analyses by ensuring that an aggressive interprocedural pointer analysis, type information, and call graph are never invalidated at run-time due to memory errors. Finally, SAFECODE can detect some of the hard-to detect memory errors like dangling pointer errors with low overhead for some class of applications and can be used not only during development but also during deployment. SAFECODE requires no source code changes, allows memory to be managed explicitly and does not use metadata on pointers or individual tag bits for memory (avoiding any external library compatibility issues).

This thesis describes the main ideas, insights, and the approach that SAFECODE system uses to achieve the goal of providing safety guarantees to software written in unsafe languages. This thesis also evaluates the SAFECODE approach on several benchmarks and server applications and shows that the performance overhead is lower than any of the other existing approaches.

To my parents and Amulya.

Acknowledgments

This thesis would not have been possible without the support of many people.

I owe my deepest gratitude to my adviser, Prof Vikram Adve. His guidance, patience and encouragement have been crucial in successful completion of this thesis. He has always been available to discuss my ideas and has never ceased to amaze me with his great insights and suggestions. His enthusiasm and work ethic are a great source of inspiration for me.

I owe special thanks to Sumant Kowshik with whom I have collaborated during my initial years at UIUC. I have enjoyed the many discussions we have had on our work and on research life in general.

I owe my thanks to all my committee members and my colleagues in LLVM research group for their valuable feedback, ideas and discussions. Also, many thanks to Chris Lattner for developing LLVM, DSA and pool allocation without which many results in this thesis would not have been possible.

I would also like to thank Molly for taking care of many conference registrations, travel arrangements and such.

Special thanks to my friends, Jayram and Ahmed, for keeping me sane and making my stay in Champaign memorable.

Last but not the least, I am grateful to my dearest family – my parents and my wife Amulya, for their love, support and understanding all these years.

Table of Contents

Chapter 1	Introduction	1
1.1	The SAFECode Approach	2
1.2	Contributions of the Thesis	4
1.3	Thesis Organization	5
Chapter 2	Definitions, Overview and Related Work	6
2.1	Memory Safety : Definition	6
2.2	Strategies for Providing Memory Safety to C Programs	7
2.3	SAFECode Strategy	8
2.3.1	SAFECode Usage	9
2.3.2	Security Guarantees	10
2.4	Related Work	11
2.4.1	Language Restrictions and Annotations for Type Safety	11
2.4.2	Systems that use Run-time Check Strategy	13
2.4.3	Error Checking with no Memory Safety Guarantee	14
Chapter 3	Background	17
3.1	Memory errors in C : Terminology	17
3.2	Program Representation	17
3.3	Pointer Analysis	18
3.4	Background on Automatic Pool Allocation	20
Chapter 4	SAFECode: Sound Analysis and Memory Safety	24
4.1	Insights	25
4.2	SAFECode Type System	27
4.2.1	Syntax	27
4.2.2	Typing Rules	29
4.2.3	Operational Semantics	32
4.2.4	Soundness Proof	34
4.3	SAFECode Guarantees	38
4.3.1	Sound Analysis Guarantee	38
4.3.2	Memory Safety Guarantee	38
4.3.3	Security Guarantee	39
4.4	Extensions for Full C	39
4.4.1	Structure Types	40
4.4.2	Region-polymorphism for Functions	40
4.4.3	Cycles in Points-to Graphs	41

4.4.4	Function Pointers	41
4.4.5	Control Flow	42
4.4.6	Global and Stack Allocations	42
4.5	Limitations	42
4.5.1	Incomplete Error Checking	42
4.5.2	Potential Increase in Memory Usage	43
4.5.3	Restrictions on Pointer Analysis	43
4.5.4	Manufactured Pointers	44
4.5.5	Custom Memory Allocators	44
4.5.6	Compatibility with External Libraries	44
4.6	Implementation	46
4.6.1	Type Inference and Type Checking	46
4.6.2	The SAFECode Runtime System	46
4.7	Sound Static Analyses Enabled By SAFECode	48
4.7.1	Static Array Bounds Checking in SAFECode	48
4.7.2	Static Analyses in ESP	48
4.8	Experiments	50
4.8.1	Run-time Overheads	51
4.8.2	CCured Comparison	53
4.8.3	Effectiveness of Static Analysis	53
4.9	Related Work	53
4.10	Sound Analysis and Memory Safety: Concluding Discussion	55
Chapter 5 Formal Proof of Soundness		57
5.1	Invariants for Well Formed Environments	57
5.2	Proof	59
Chapter 6 Backwards-Compatible Bounds Checking for C/C++		74
6.1	Runtime Checking with Efficient Referent Lookup	76
6.1.1	The Jones-Kelly Algorithm and Ruwase-Lam Extension	76
6.1.2	Our Approach	79
6.1.3	Handling Out-Of-Bounds Pointers	82
6.1.4	Compatibility and Error Detection with External Libraries	83
6.1.5	Errors in Calling Standard Library Functions and System Calls	84
6.1.6	Optimizations	85
6.2	Compiler Implementation	87
6.3	Experiments	88
6.3.1	Overheads	88
6.3.2	Effectiveness in Detecting Known Attacks	90
6.3.3	Performance Comparison with Previous Approaches	91
6.4	Related Work	92
6.5	Run-time Bounds Checking: Concluding Discussion	94
Chapter 7 Efficient Detection of All Dangling Pointer Uses		95
7.1	Detecting Dangling Pointers : Our Approach	98
7.1.1	Overview of the Approach	98
7.1.2	Page Mapping for Detecting Dangling Pointer Errors	99
7.1.3	Reusing Virtual Pages via Automatic Pool Allocation	101

7.1.4	Avoiding Costs of Long-lived Pools	102
7.1.5	Implementation	104
7.2	Experimental Evaluation	104
7.2.1	Run-time Overheads for System Software	104
7.2.2	Comparison with Valgrind	106
7.2.3	Address Space Wastage due to Long-lived Pools	107
7.2.4	Overheads for Allocation Intensive Applications	108
7.3	Related Work	108
7.3.1	Techniques that Rely on Heuristics to Detect Dangling Pointer References	109
7.3.2	Techniques that Guarantee Absence of Dangling Pointer References	109
7.3.3	Techniques that Check using MMU	110
7.4	Detecting Dangling Pointer References: Concluding Discussion	110
Chapter 8	SAFECode for Embedded Systems	112
8.1	Language Restrictions	112
8.2	Uninitialized Pointers	113
8.3	Stack Safety	113
8.4	Dangling Pointers to Freed Memory	113
8.5	Array Safety and String and I/O Libraries	115
8.6	Summary of Results	115
8.7	SAFECode for Embedded Systems: Concluding Discussion	116
Chapter 9	Static Array Bounds Check Analysis	117
9.1	Generating the Constraints	119
9.2	Interprocedural Propagation	121
9.3	Proving Array Safety	122
9.4	Checking Safety Preconditions	122
Chapter 10	Conclusion	123
10.1	Future Work	124
10.1.1	Applying SAFECode to Operating System Components	124
10.1.2	More Precise Type Inference	125
10.1.3	Non-unification based Pointer Analysis	125
References	126
Author's Biography	134

Chapter 1

Introduction

Many computing systems today are written in weakly typed (or unsafe) languages like C and C++ that provide weak semantic guarantees due to the presence of undetected memory errors like bounds violations, pointer cast errors etc. The presence of these undetected memory errors is the primary reason for poor reliability of software today. Despite this, increasingly complex software continues to be written in these languages because of performance and backwards-compatibility considerations.

Safe (or strongly typed) languages such as Java and C#, on the other hand, provide strong semantic guarantees. Consequently, there are three main benefits of using safe languages:

- They improve system security by preventing common sources of vulnerabilities such as buffer overflow attacks
- They improve software reliability by providing better error-detection at development time
- They enable sound static analysis techniques to be used in compile-time tools.

Unfortunately, current techniques that achieve some of these benefits for C/C++ programs generally require expensive run-time checks (see Chapter 2.4), limiting such tools to be used primarily for offline debugging rather than for production code [4, 37, 55, 50].

The third benefit mentioned above is essential for checking program properties in compile-time tools. Compile-time array bounds checking algorithms (e.g. ABCD [7]), static checking tools (e.g. ESP [18]), and many security checks performed by virtual machines [49] rely on program analysis that would not be sound if applied to C or C++ programs. In particular, memory errors

in C programs like dangling pointer references, array bounds violations, and undetectable type conversions, can violate the semantics assumed by most non-trivial static analysis algorithms. Optimizing C/C++ compilers simply give undefined results for erroneous programs, but this is not acceptable for program checking tools that aim to provide guarantees about error detection or prevention using program analysis.

To appreciate the difficulty of enforcing the third property, consider previous error detection tools for C [4, 66, 37, 72, 54, 71, 50]. Most of these tools simply do not provide any guarantees because they use heuristic techniques to detect certain errors, especially dangling pointer errors, and these heuristics may not detect some memory errors. The tools by Xu et. al [71] and Patil and Fisher [54] can reliably detect memory reference errors, including dangling pointer errors. But this comes at the cost of high overheads (2x-6x) in many programs. These tools also require use of extensive metadata that complicate interoperability with external libraries. Furthermore, they do not prevent type violations on certain legal references. To our knowledge, none of these tools provide a sound semantics despite their high-overhead run-time checks.

An alternative solution is provided by systems like Cyclone [29] and CCured [13], both of which aim to guarantee safe execution and a sound semantics for C programs. Both systems, however, disallow explicit memory deallocation and instead, use automatic memory management, require porting effort (syntactic or library compatibility), and incur potentially high run-time overheads in memory and time (further discussed in Chapter 2.4). When programmers are willing to accept these requirements both systems provide an attractive method for achieving the three benefits discussed earlier. We believe, however, that there are many existing applications in weakly-typed languages like C and C++ that will not be modified to meet the requirements of these systems. To our knowledge, there is no existing system today that can provide these three benefits for the vast base of existing software.

1.1 The SAFECODE Approach

This thesis describes a system called **SAFECODE** (Static Analysis For safe Execution of Code) provides a subset of each of the three benefits of safe languages, including useful sound semantics for nearly arbitrary C programs. First SAFECODE enforces memory safety (defined in Chapter 2.1).

Second, SAFECODE detects many programming errors normally prevented by a safe programming language such as uninitialized pointers, array bounds violations, pointer cast errors, etc. SAFECODE attempts to detect these errors at compile-time where possible and uses run-time checks for the rest. It limits the run-time error detection in such a way that *no* metadata is needed in individual pointers or objects.

Third, and perhaps the most important technical contribution of this thesis, SAFECODE provides a foundation for sound static analysis techniques to be used for nearly arbitrary C programs. In particular, SAFECODE ensures the correctness of a call graph, fairly aggressive interprocedural pointer analysis information, and type information for a subset of the data objects in a program. Compiler analyses can build upon this foundation to perform reliable static analyses and transformations of programs. In fact, we exploit this capability to perform aggressive static checking, which greatly reduces the run-time overhead of SAFECODE by eliminating many run-time checks. We also give examples of other static checking techniques and external tools that could be applied on top of SAFECODE and guaranteed to be sound. To our knowledge, SAFECODE is the first tool to enable sound and sophisticated static analyses for ordinary C programs, despite the possibility that such programs may cause dangling pointer references, array bounds violations, and undetectable type conversions.

The basis of our approach is to use a pointer analysis to partition heap memory into fine-grain pools (or regions) [42]. We use type-homogeneous pools containing objects of “known type” and a combination of static analysis and run-time checks to enforce isolation between the pools, prevent memory access violations, and guarantee the correctness of the points-to graph and the call graph.

SAFECODE does not require source changes and does not impose any changes to the explicit memory management model of C. First, it requires no annotations whatsoever with all its transformations and analyses being automatic. Second, SAFECODE programs allocate and free memory objects at exactly the same points as that of the original program, minimizing the need to tune memory consumption. Third, SAFECODE uses novel techniques to avoid the need to associate any metadata with individual pointer variables, memory objects, or tag bits with pointer values. Such metadata or tag bits are undesirable because they complicate the passing of values between a program and external libraries.

1.2 Contributions of the Thesis

The contributions of this thesis can be summarized as follows:

- We present a new system called SAFECODE for guaranteeing memory safety with very low overhead for weakly typed languages.
- SAFECODE guarantees that memory errors do not invalidate the points-to graphs, call graph and type information for a subset of memory objects at run-time, thus providing a platform for doing other high level static analyses. Our approach shows how to refine existing pointer analysis and heap partitioning transformation and combine them with new techniques to provide the above guarantees with lower run-time overhead than previous approaches.
- We show how the SAFECODE approach can be formalized as an extension to the type system of the weakly typed languages to include a new kind of region types. We then formulate the operational semantics for SAFECODE in such a way that, in the absence of memory errors, the semantics are equivalent to the standard semantics of weakly typed languages. More importantly, for any well typed program in this system (including the ones which contain memory errors), the semantics guarantee memory safety, sound points-to graph, sound call graph, and sound type information for a subset of memory.
- We show how SAFECODE semantics enable many static analysis algorithms, which are otherwise not sound in the presence of memory errors. We propose a new interprocedural array bounds check algorithm, which is sound in the SAFECODE framework and show how it can be used to reduce the run-time overhead in SAFECODE.
- We present a backwards-compatible bounds checking solution that can be used as a part of SAFECODE and has very low overhead.
- We present a new technique to detect all dangling pointer references (uses of pointers to freed memory) that incurs low overhead when used for server applications.

1.3 Thesis Organization

The remainder of the thesis is organized as follows.

Chapter 2 gives an overview of the SAFECODE approach and positions the SAFECODE work in relation to other existing approaches. Chapter 3 gives the necessary background to understand the rest of the thesis. Chapter 4 describes the SAFECODE approach for enforcing sound analysis information. Chapter 5 formalizes the SAFECODE approach as a region-based type system and gives a formal proof of soundness. Chapter 6 presents a new run-time bounds checker developed as a part of the SAFECODE approach. Chapter 7 describes a novel approach to detect dangling pointers. Chapter 8 presents the SAFECODE version when restricted to embedded systems. Chapter 9 discusses a novel interprocedural static bounds checking algorithm. Chapter 10 concludes with directions for future work.

Chapter 2

Definitions, Overview and Related Work

2.1 Memory Safety : Definition

We define the following collection of guarantees as a “*memory safety*” guarantee.

- (S1) *Control flow integrity*: An executing program only follows the precise control flow paths predicted by the compiler. Note that for indirect function calls, the set of predicted callees (predicted by the compiler’s “call graph”) may be a superset of those intended by the programmer. Nevertheless, safety is ensured because no unexpected transfers of control can occur.
- (S2) *Data access integrity*: For a subset of data objects, the executing program respects the object type and does not execute any illegal operations on those objects. For other data objects, the object type may be violated but in all cases the program only accesses data memory explicitly allocated by the program.
- (S3) *Rendering memory errors harmless*: Any memory referencing errors (uninitialized pointers, array bounds violations, string buffer overruns, or dangling pointer references to heap or stack objects) are either explicitly prevented or rendered harmless. Harmless here means that they cannot cause violations of (S1) and (S2).

The root cause of violations of these guarantees are the memory referencing errors listed above in S3. Note that other errors commonly associated with security violations, e.g., integer overflow,

are not actually exploitable in themselves; they become exploitable when they produce memory errors such as an array bounds violation.

2.2 Strategies for Providing Memory Safety to C Programs

In general, providing memory safety guarantees to a C/C++ program using only static analysis is undecidable. Many previous tools or techniques have adopted different strategies to make this problem tractable. We classify these strategies for providing memory safety guarantees into three main categories:

Restrict	Place restrictions on the usage of language features (e.g., restrict pointer casts) to enable static analysis tools to check for memory or type safety violations.
Run-time Checking	Perform exhaustive run-time checks augmented with static analysis to eliminate the run-time overhead
Annotate	Add annotations to programs that will enable powerful verification tools to check for memory or type safety violations

All of the previous tools/technologies (research or industrial) have used a combination of these strategies to provide the necessary memory safety guarantees. The advantages and disadvantages of using each of these strategies is given in Table 2.1.

Strategy	Advantages	Disadvantages
Restrict	Complete static checking	Supports only few applications
Run-time Checking	Less rewriting of existing applications	May incur prohibitively high overhead
Annotate	Support large class of applications	May require adding large number of annotations

Table 2.1: Comparing various strategies for memory safety

The first strategy (**Restrict**) is useful in specific application domains where the restrictions on language usage are acceptable (e.g., embedded or control software). The second strategy (**Run-time checking**), as the name suggests, relies on exhaustive run-time checking to check for memory

safety violations. Most of the previous tools that have used this strategy have been either prohibitively expensive or required some changes to the existing software. The third strategy allows more expressive language features than the first but requires (perhaps extensive) rewriting of applications in terms of adding annotations to existing code to enable complete static checking. We consider any wrappers required for library compatibility reasons as annotations.

The strategies adopted by some well known approaches are given in Table 2.2. Vault [19] is a new *restricted* programming language derived from C that also uses some *annotations* to provide type safety guarantees to programs. Cyclone [29] mainly uses a region based annotation scheme to provide type safety guarantees to a C like language. CCured [52] is an extension of C type system that mainly relies on *Run-time checking* strategy and to a limited extent on wrappers (*Annotations*), to provide memory and type safety guarantees. SAL/EspX [31], on the other hand, uses a pure *Annotation* strategy. A detailed description of the strategies used by different tools is given later in this chapter.

Strategy	Examples
Restrict	Vault, Cyclone, Control-C
Run-time Checking	CCured , Xu et al
Annotate	Vault, CCured, Cyclone, SAL/EspX

Table 2.2: Examples that use various strategies for memory safety

2.3 SAFECode Strategy

In the context of providing safety to legacy code, we believe that the *Annotation* strategy though attractive, is perhaps the hardest to adopt for programmers. There have been some successful attempts at adding annotations to a general purpose language (e.g, SAL/EspX [31]), but in general, it has been difficult to popularize the annotation schemes among programmers.

In this thesis, we focus on the other two strategies and extend the state of the art in both. We propose three solutions to the problem:

1. SAFECode with minimal language restrictions to enable static checking (we call this as SAFE-Code for embedded systems)

2. SAFECode with memory safety guarantees but without complete error detection
3. SAFECode with memory safety guarantees and complete error detection

Each of these solutions and their properties, advantages, and limitations are discussed below.

SAFECode for embedded systems: This solution extends the state of the art in minimizing the restrictions that are necessary to support large number of programs. The main advantage of this solution is that on programs that satisfy the restrictions, it does not impose any performance overhead. The restrictions in this work have been carefully chosen so that most existing applications in the embedded world already follow them or can be easily modified to do so. This solution is discussed in detail in Chapter 8.

SAFECode with memory safety and sound analysis guarantees but without complete error detection: This solution explores the possibility where SAFECode provides useful but lesser guarantees than a completely safe language and with very low overhead. The only errors that this solution does not detect are the dangling pointer errors. However, this solution handles these errors in such a way that the call graph information, alias analysis information, and the type information are never invalidated at run-time. In practice, these guarantees are extremely difficult to achieve for arbitrary C programs. We impose the restriction that referencing pointers to manufactured addresses is disallowed. This solution, its advantages, and limitations are discussed in Chapters 4 and 5.

SAFECode with memory safety and sound analysis guarantees with complete error detection: This solution provides a complete safety guarantee with complete error detection. The only difference with the solution above is that it also detects dangling pointer errors, which is discussed in Chapter 7. However, as shown in Chapter 7, this strategy might incur prohibitively high overhead in some cases.

2.3.1 SAFECode Usage

Each of the solutions above can be thought of as a different mode in which SAFECode can be used. All the three modes provide the necessary memory safety guarantees without compromising on the backwards-compatibility (no annotations or wrappers are needed). The first mode (SAFECode for embedded systems) is useful in cases where any performance overhead is intolerable and the

restrictions are not too onerous as in the case of embedded systems. The other two modes are used for general purpose software where language restrictions are not acceptable. We envisage the use of the third mode during development, debugging, testing, and even during deployment if the overheads are low enough for production use. If the overheads are high, we can always fall back on the second mode that does not detect dangling pointer errors but still provides useful guarantees.

2.3.2 Security Guarantees

SAFECode (in all the three modes) provides a memory safety guarantee. This means that SAFECode provides “control-flow” integrity and thus has the following important security properties:

- No security attack can subvert the control flow of a program and execute arbitrary injected code. This prevents many attacks that rely on executing injected code including common buffer-overflow attacks and heap corruption attacks.
- Attacks that rely on library being loaded at certain program points (“return-to-libc” attacks [68]) are not possible.
- The security provided by SAFECode is stronger than that provided by no execute bit (NX) in modern architectures like AMD[28] since the NX bit does not capture return-to-libc attacks.
- The security provided by SAFECode is stronger than that provided by techniques that focus on non-corruption of return address [14, 15].
- Finally, since SAFECode ensures that programs do not deviate from their call-graph, it guards against many attacks that focus on function pointer corruption.

However, in the first two modes above, SAFECode does not detect dangling pointer errors. In these two modes, SAFECode may not be able to detect attacks that are based on corrupting data but not control-flow (called non-control data attacks [11]). As explained later in Chapter 4.3.3, we believe that with SAFECode these errors become much harder to exploit. Nevertheless, there may be some situations, especially server applications, where this guarantee is not enough. we expect SAFECode in the third mode to be used for server applications and guard against such attacks.

2.4 Related Work

In this section, we contrast several related systems that aim to provide total or partial memory safety with SAFECode approach. The focus here is more on evaluating the effectiveness of these systems in realizing our objectives of providing both a memory safety guarantee and a platform for sound high level static analysis. A detailed description of some of the technical differences between SAFECode and these systems is given in the later chapters.

A number of approaches have been proposed in literature to provide memory or type safety to programs written in weakly typed languages. We discuss each of them below including the strategy that they have used to provide the guarantee (*Restrict*, *Run-time Check*, or *Annotate*). We group them into two main categories: techniques that use language *restrictions or annotations* and techniques that use *Run-time checking*. We then also contrast the SAFECode work with other approaches that focus on detecting certain class of memory errors but do not provide any safety guarantees.

We use the following criteria for comparing these approaches with SAFECode:

- How much effort is required for porting existing programs to these approaches?
- How do the run-time overheads of these approaches compare with SAFECode?
- Can they detect all (or most) memory errors?
- Do they guarantee memory safety?
- Can they be used to support sound high level static analysis?
- Do they retain C memory model of explicit memory deallocation?

In Table 2.3, we compare SAFECode in some detail with three other systems representing three different points in the spectrum of safety guarantees using the above criteria. These comparisons will be explained in detail in the following sections.

2.4.1 Language Restrictions and Annotations for Type Safety

One way to achieve type safety (and in turn memory safety) for weakly typed languages is by imposing language restrictions or adding new language mechanisms (or typically both), which will

then enable simple type checkers to statically (or almost statically) check for safety. While this approach provides a better error detection capability than SAFECODE, a major disadvantage of this approach is that it can require a significant porting effort to make the existing systems, written in weakly typed languages, satisfy the much stricter language rules. Because of this, these approaches have enjoyed limited success and only in restricted domains. However, with the stronger type safety guarantee, these systems can easily realize our goal of providing a platform for other sound static analysis to be reliably applied to programs written in these languages. We compare SAFECODE with few such systems in detail this section.

Cyclone: Cyclone [29] uses a region-based type system to enforce strict type safety, and consequently memory safety, for a subset of C programs. We compare Cyclone with SAFECODE in Table 2.3. Unlike SAFECODE, Cyclone disallows non-type-safe memory accesses. Cyclone and other region-based languages [8, 27, 9, 12]) have two disadvantages relative to our work: (a) they can require significant programmer annotations to identify regions; and (b) they provide no mechanism to free or reuse memory within a region. This means that data structures that shrink and grow (with non-nested object life times) must be put into a separate garbage-collected heap or may incur a potentially large increase in memory consumption (e.g., Cyclone and RT Java both include a separate garbage collected heap). Automatic region inference [61, 29] can eliminate or mitigate the first but not the second, and has only been successful for type-safe languages without explicit deallocation. In contrast, we permit explicit deallocation of individual data items and thus retain the C memory model for deallocation. Also, Cyclone requires bounds checks for most array accesses and has reported overheads up to 3x.

Linear and alias type based systems [16, 67, 19]: Linear and alias types have been used to statically prove memory safety in the presence of explicit deallocation of objects. They achieve this primarily with severe restrictions on aliases in a program, which so far have not proved practical for realistic programs. The approach also requires significant type annotations to be inserted. In contrast, we require no annotations, permit nearly arbitrary pointer usage, and yet achieve memory safety for a broad range of applications. One of these languages, Vault [19], also uses such a type system (much more successfully) to encode many important correctness requirements for other dynamic resources within an application (e.g., file handles and sockets). It would be very attractive

to use Vault’s mechanisms within our system to statically check key correctness requirements of other resources besides memory.

Control-C: In our earlier work [40], we adopted a drastic approach of restricting all casts between incompatible types, allowing only those array accesses which can be statically ascertained to be safe, and using a simplified region based annotation system (where only one region is live at any time) for dynamic memory usage to guarantee type safety. While this restricted language proved sufficient in the domain of real-time control systems, it is not suitable for more general programs. The SAFECODE for embedded systems approach (discussed in Chapter 8) is an extension of control-C that allows more general heap allocations and deallocations and is suitable for more general embedded applications..

Annotations for buffer overflow checking: SAL/EspX [31] uses programmer written annotations coupled with powerful automatic inference to statically check for buffer overflows in large code bases. This approach incurs no run-time performance overhead and eliminates large number of buffer overflow errors. If the programmers are willing to write these annotations, this approach is a very attractive option. However, we believe that many existing software may never be rewritten to take advantage of this strategy.

2.4.2 Systems that use Run-time Check Strategy

Software Fault Isolation: The weakest safety guarantees are provided by runtime sandboxing techniques such as Software Fault Isolation (SFI) [66], which only aim to prevent a software module from writing to memory outside its data space (and optionally prevent it from reading such memory). SFI works directly on object code, making it simpler to implement than compiler-based systems, language-independent, and relatively easy to use (it does not require program changes). SFI does not achieve two of the three goals of SAFECODE described in Chapter 1: it does not attempt to detect programming errors directly, and it provides very weak guarantees that can not be used by other static analysis tools. SFI also introduces substantially higher runtime overhead than SAFECODE, 25% to 59% when checking writes only and typically over 100% when checking reads and writes, despite providing weaker guarantees.

Other systems including Patil et al [55], CCured [13], Xu et al [71] have the goal of strong type

safety and thus can satisfy our serve as a sound platform for other high level analysis. The reported overheads are 2x-6x for Patil et al [55], and their work is not usable for production level systems.

Xu et al: Xu et al [71] describe an extension of Patil Fischer, which with other optimizations, can greatly reduce the run-time overhead. But the average reported overheads (**up to 133%**) are much higher than SAFECode. Moreover, they currently reject programs that have casts between pointers of different types. Adding support for such casts could worsen their run-time overheads. In contrast, SAFECode supports casts between arbitrary types. However, the main advantage with their system is that they can detect dangling pointer errors without the use of garbage collection. SAFECode cannot detect dangling pointer errors in the normal mode of operation.

CCured: The CCured [13, 53] system comes closest to our goals of providing memory safety and partial type safety for C programs, and of using static analysis to reduce the overheads of runtime checking. The major advantage of CCured is that it provides nearly complete type safety, the main exception being loads and stores via pointers to memory whose types cannot be statically inferred (WILD pointers). This guarantee is stronger than what SAFECode provides. This benefit comes at some cost, and CCured has three disadvantages relative to SAFECode. First, CCured requires a conservative garbage collector, which may not always be preferred by users. Second, CCured introduces significant metadata for runtime checks, including tag bits and bounds information associated with WILD pointers, bounds information for SEQ and FSEQ pointers, and argument counts for function pointers. This metadata is the primary cause of the porting effort required for using CCured on C programs, in particular, requiring wrappers around some library functions. In contrast, SAFECode uses no metadata on individual pointer values as explained in Chapter 4, and essentially works “out-of-the-box” on existing C programs. Third, CCured has higher run-time overhead than SAFECode in some cases as reported in Chapter 4.8.

2.4.3 Error Checking with no Memory Safety Guarantee

A number of approaches have been proposed for error checking in weakly typed languages. Some rely entirely on static analysis for error detection, while several others use a combination of run-time checks and static analysis. In either case, the main goal of these approaches is to help detect errors. They do not provide any safety guarantees. Without the guarantees, these systems can not

Criterion	SAFECode	CCured	SFI	Cyclone
Goals				
Safety Properties:	Memory safety and <i>limited error detection</i>	Memory and type safety	<i>Memory safety only</i>	Memory and type safety
Key Strength:	Safety without GC	Type safety for most C programs	Simplicity; ease of use	Type safety without GC
Key Weakness:	<i>Limited type safety</i>	<i>Metadata and conservative GC</i>	<i>Weak safety guarantees</i>	<i>Differences from C; runtime overheads</i>
Support for sound static analysis	Yes	Yes	<i>No</i>	Yes
Language restrictions				
Porting effort?	Negligible	Minor	None	<i>Significant</i>
Require type-safe code as input?	No	No	No	<i>Yes</i>
Allow stack addresses stored into heap?	Allowed	<i>Disallowed</i>	Yes	<i>Disallowed</i>
Allow use of pointer cast from integer?	Allowed, except arbitrary constant integer	<i>Disallowed</i>	Yes	<i>Disallowed</i>
Library compatibility?	Easier	<i>More difficult</i>	<i>More difficult</i>	Easier
Platform independent?	Yes	Yes	<i>No</i>	Yes
Error checking				
Prevent array-bound violations?	Yes	Yes	<i>No</i>	Yes
Prevent uses of uninitialized variables?	Yes	Yes	<i>No</i>	Yes
Prevent dangling pointer references?	<i>No</i> if low overhead is desired	Avoid via <i>conservative GC</i>	<i>No</i>	Avoid via region type system and GC
Static Checking and Run-time overheads				
Meta-data for pointers?	No	<i>Yes</i>	No	No
Meta-data for arrays?	No	<i>Yes</i>	No	<i>Yes</i>
Eliminate GC?	Yes	<i>No</i>	Yes	Yes
Static array bounds checks?	Many	<i>No</i>	No	<i>Few</i>
Eliminate null pointer checks?	Yes	<i>No</i>	No	<i>No</i>

Table 2.3: Comparison of SAFECode against CCured, SFI, and Cyclone. *Less desirable features are highlighted in italics.*

(and do not) serve as a sound basis for other high level static analysis techniques.

Error Detection via Static Analysis

Here we discuss systems focusing on error detection via static checking alone. Since complete static error detection (without a high false positive rate) is not possible for general programs, some of these systems [69, 65, 25] focus on one particular aspect of memory errors, namely, bounds checking for arrays. Because of this, they do not guarantee memory safety. In contrast, `SAFECODE` guarantees memory safety not only for array accesses but also for all memory accesses. Other systems like `Metal` [32], `ESP` [18], etc aim to detect protocol errors in programs. Some of the safety problems can be encoded as simple protocols like uninitialized pointers problem, dangling pointers to freed memory problem, etc, and hence can be checked for correctness using these tools. However, both these tools can give a lot of false positives (and in the case of `Metal` can also miss a few errors). `SAFECODE`, in contrast, provides a memory safety guarantee and a sound framework for performing other high level static analysis. Also, when analyzing code for other high level protocol errors, these systems assume memory safe semantics for C.

Error detection via static analysis and run-time checks

There have been large number of systems for detecting memory access errors by adding run time checks and meta-data [33, 58, 4, 37, 50, 72] (the work by Loginov et al. also detects type errors [50]). Again, the main focus of this work has been to check errors in systems or aid in debugging (or both). These systems can generate some false negatives (i.e., undetected memory referencing errors) and do not provide any safety guarantees. Furthermore, the reported run-time overheads in these systems are too high for production use. For example, reported slowdowns are up to 5x for `SafeC` [4] and 5-6x for Jones and Kelly [37]. Widely used tools like `purify` and `valgrind` have similarly large overheads. While Yong et al [72] report overheads (**43%**) that are an order of magnitude better than others, their overheads are less because they check only writes to memory but not reads. If memory reads, which are more common in programs, are checked then the overheads could be much worse.

Chapter 3

Background

3.1 Memory errors in C : Terminology

Using terminology from SafeC [4], memory errors in C programs can be classified into two different types: (1) Spatial memory errors and (2) Temporal memory errors

Spatial memory errors in C programs include array bounds violations (i.e., buffer overrun) errors, uninitialized pointer dereferences (causing an access to an invalid address), invalid type conversion errors, format string errors, etc. Temporal memory errors include uses of pointers to freed heap memory and uses of pointers to an activation record after the function invocation completes.

The basic strategy of the SAFECODE system is to use compile-time pointer analysis as a foundation for static safety checks (for both temporal and spatial errors) and if necessary, for supplementing static checking with runtime checks. In the rest of this chapter, we give the necessary background for understanding the input program representation, pointer analysis and its representation, and the Automatic Pool Allocation transformation [43, 42] used in the thesis.

3.2 Program Representation

Although our implementation of SAFECODE supports all of C, we use a subset of C as the source language in this thesis to simplify the presentation. This language, shown in Figure 3.1, includes most sources of potential memory errors in weakly typed languages including:

(P1) dangling pointers to freed heap memory,

(P2) array bound violations,

vars	$x \ y$
Function names	f
Field names	Fld
Pointer Type pt	$:= \tau^*$
Function Type ft	$:= \tau \longrightarrow \tau$
Structure Type st	$:= \text{struct } \{ \text{Fld}_1 : \tau_1, \dots, \text{Fld}_n : \tau_n \}$
Types τ	$:= \text{int} \mid \text{char} \mid \text{pt} \mid \text{st} \mid \text{ft}$
declarations $decl$	$:= \epsilon \mid \tau \ x; \text{decl}$
Statements S	$:= \epsilon \mid S \ S \mid x = E; \mid \text{store } E, E; \mid \text{storec } E, E; \mid \text{free}(E); \mid \text{if } (E) \text{ then } \{ S \} \text{ else } \{ S \} \mid \text{while } (E) \{ S \}$
Functions F	$:= \tau' f(x : \tau) \{ \text{decl} ; S \}$
Expressions E	$:= x \mid E \text{ op } E \mid \text{cast } E \text{ to } \tau \mid \text{load } E \mid \text{loadc } E \mid \text{malloc}(E) \mid \&E[E] \mid \&(x \rightarrow \text{Fld}_i) \mid \&f \mid \text{alloca}(E)$ $op \in \{+, -, *, /, \%, \&\&, , \hat{,} <<, >>\}$
Definitions d	$:= F \mid \text{struct } x \{ \text{Fld}_i : \tau_1, \dots, \text{Fld}_n : \tau_n \}$
GlobalDecl gd	$:= \tau * \rho \ x = \text{galloc}(E, E)$
Programs p	$:= d \ p \mid gd \ p \mid \epsilon$

Figure 3.1: C like Language

(P3) accesses via uninitialized pointers, and

(P4) arbitrary cast from an int type to another pointer type and subsequent use.

We only include 4-byte and 1-byte integer types (`int` and `char`) as primitive data types and use distinct load and store operations for these types (load `E` for loading `ints` and `loadc E` for loading `chars`). The `cast`, `malloc`, and `free` operations are similar to those in C. We use a new operation called `alloca` (with arguments similar to `malloc`) to allocate memory on the stack. Also, global variables can only be pointers pointing to global memory that is allocated and initialized using a new operation called `galloc`, which takes a size parameter and an initializer; this essentially is how globals in a C program operate. These two features make it unnecessary to apply the `&` operator to get the address of a stack variable or global object; `&` is only used for indexing into structures, arrays and for function pointers.

We will use a running example, shown in Figure 3.3 (a), to illustrate the steps of our approach.

3.3 Pointer Analysis

We now describe the properties of pointer analysis that we use in our approach and its representation.

Node var	ρ
new PointerType pt	$:= \tau * \rho \tau * (\rho, n)$
Function Sets fs	$:= f, fs \epsilon$
FuncPtrType fpt	$:= ft * fs$
new StructType st	$:= st_{prev} \forall \rho.st \tau < \rho >$
new Type τ	$:= int pt st ft fpt Unknown$
new Expressions E	$:= E_{prev} \&(x \rightarrow Fld_i) \&f$
new Statements S	$:= S_{prev} associate(\rho, \tau)$
new Definitions d	$:= d_{prev} FSET fs = f, fs$

Figure 3.2: Syntactic extensions for representing pointer analysis results. st_{prev} , E_{prev} , S_{prev} and d_{prev} are same as st , E , S and d in Figure 3.1.

Intuitively, pointer analysis representation can be thought of as a storage-shape graph [59, 36] (also referred to as a points-to graph), where each node represents a set of dynamic memory objects (a “points-to set”) and distinct nodes represent disjoint sets of objects. Pointers pointing to two different nodes in the graph are not aliased. We assume there is one points-to graph per function, since this allows either context-sensitive or insensitive analyses. Figure 3.3 (b) shows the points-to graph for the running example. We do not concern ourselves with how these analysis results are actually computed; we only assume that they are given in the format described here. In our implementation, we use the Data Structure Analysis (DSA) [44] to compute the pointer analysis results.

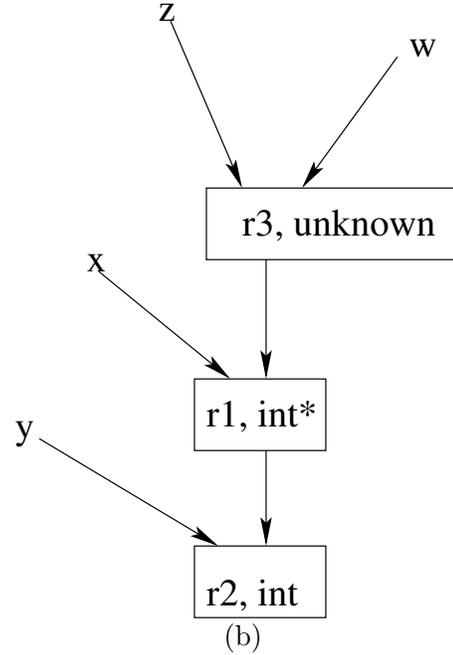
We assume that the pointer analysis is encoded as type attributes within the program in a type system analogous to Steensgaard’s [59]. Each points-to graph node is encoded as a distinct type (although we continue to refer to nodes below). The input to our approach is a program in this language shown in Figure 3.2. Figure 3.4 shows the running example in our input language. Each pointer in this language has an extra attribute, ρ , which intuitively corresponds to the node it points to in the points-to graph. For example, in Figure 3.4, the type of y is $int * r2$, denoting that it points to objects of node $r2$ in the points-to graph. The statement $associate(\rho, \tau)$ associates node ρ of the graph with type τ , denoting that the node ρ contains objects of type τ . If τ is a pointer type, say, $\tau' * \rho'$, then $associate(\rho, \tau' * \rho')$ directly encodes a “points-to” edge from node ρ to node ρ' . These $associate$ statements are typically listed at the beginning of each function. Note that there can be only a single target node for each variable (or field of pointer type), which restricts the input to a unification-based pointer analysis. We discuss the function pointer and struct representation, later in chapter 4.4.

```

int **x, *y, *z, ***w, u;
x = (int **) malloc(4);
y = (int *)malloc(4);
z = (int *) malloc(4);
...
store y, x // equivalent of *x = y
store 5 ,y
free(z) ; // creates a dangling pointer
store 10, z;
...
u = load z; // equivalent of u = *z;
...
w = cast z to (int ***);
store x, w;

```

(a)



(b)

Figure 3.3: (a) Running example (b) its points-to graph

Memory that is used in a type-inconsistent manner, e.g., due to unions or casts in C, must be assigned type *Unknown* (this is verified by our type checker). *Unknown* is interpreted as an array of chars. In the running example, the target of *z* (node *r3*) has type *Unknown* because this memory is accessed both as an *int* and an *int***. Distinct array elements due to *Unknown* or an actual array are not tracked separately.

In the absence of `free`s and other memory errors, we can check that this program encodes the correct aliasing information by using typing rules similar to Steensgaard’s. We do not give those rules here as our approach described in chapter 4.2 is stronger and subsumes this checking; we not only check that the static aliasing is correct but we also enforce it in the presence of memory errors.

3.4 Background on Automatic Pool Allocation

Given a program containing explicit `malloc` and `free` operations and a points-to graph for the program in the representation above, Automatic Pool Allocation transforms the program to segregate data into distinct pools on the heap [46]. By default, pool allocation creates a distinct pool for each points-to graph node representing heap objects in the program; this choice is necessary for the current work as explained later. Pools are represented in the code by pool descriptor variables.

```

associate(r1, int * r2);
associate(r2, int);
associate(r3, Unknown);
int *r2 *r1 x;
int *r2 y;
int *r3 z; *r2*r1*r3 w; int u;
x = malloc(4);
y = malloc(4);
z = malloc(4);
...
store y, x; store 5, y;
free(z); // dangling pointer still exists
store 10, z ;
...
u = load z ;
...
w = cast z to (int *r2*r1*r3);
store x, w ;

```

Figure 3.4: Running example with pointer analysis results encoded within the program

For a points-to graph node with $\tau \neq \textit{Unknown}$, the pool created will only hold objects of type τ (or arrays thereof), i.e., the pools will be *type homogeneous* with a known type. We refer to these as *TK* (stands for type known) pools and all others as *TU* (stands for type unknown) pools. Calls to `malloc` and `free` are rewritten to call new functions `poolalloc` and `poolfree`, passing in the appropriate pool descriptor.

In order to minimize the lifetime of pool instances at run-time, pool allocation examines each function and identifies points-to graph nodes whose lifetime is contained within the function, i.e., the objects are not reachable via pointers after the function returns. This is a simple escape analysis on the points-to graph. The pool descriptor for such a node is created on function entry and destroyed on function exit so that a new pool instance is created every time the function is called. For other nodes, the pool descriptor must outlive the current function so pool allocation adds new arguments to the function to pass in the pool descriptor from the caller.

We explain the pool allocation transformation with the help of a simple example shown in Figure 3.5. In this example, function `f` calls `g`, which first creates a linked list of 10 nodes, initializes them, and then calls `h` to do some computation. `g` then frees all of the nodes except the head and then returns. Note that the program has a dangling pointer error: the reference to `p->next->val`

```

f() {
    struct s *p = 0;
    // p is local
    g(p);
    p->next->val = ...; // p->next is dangling
}

g(struct s *p) {
    p->next = malloc(sizeof(struct s));
    create_10_Node_List(p);
    initialize(p);
    h(p);
    free_all_but_head(p);
}

```

Figure 3.5: Example for illustrating Automatic Pool Allocation transformation

```

f() {
    Pool PP;
    poolinit(&PP, sizeof(struct s));
    g(p, PP);
    p->next->val = ... ; //p->next is dangling
    pooldestroy(PP);
}

g(struct s *p, Pool *PP) {
    p->next = poolalloc(PP, sizeof(struct s));
    create_10_Node_List(p, PP);
    initialize(p);
    h(p);
    free_all_but_head(p, PP);
}

```

Figure 3.6: Example after Automatic Pool Allocation transformation

tries to access the second node in the list, which has been freed. Figure 3.6 shows the example after the Pool Allocation transformation. The transformation first identifies points-to graph nodes that do not “escape” a function using a traditional escape analysis (reachability analysis from function arguments, globals and return values) and creates pools for those nodes at the function entry and destroys them at the function exit. In the example, the data structure pointed to by `p` never escapes the function `f()`, so the transformation inserts code to create a pool `PP` within `f` using `poolinit` and destroys at the function exit using `pooldestroy`. For a function where the pool “escapes” (e.g, function `g` in the example) the transformation automatically modifies the function to take in extra pool descriptor arguments (see function `g` in Figure 3.6). Pool allocation then ensures that all allocations and deallocations for the data structure happen out of the corresponding pool – it converts `malloc` and `free` calls to use `poolalloc` and `poolfree` with the appropriate pool descriptors in the program. This is illustrated by the change of `malloc` call in function `g` to `poolalloc`. Similarly `malloc` calls in `create_10_Node_list()` and `free` calls in `free_all_but_head` (not shown here) are also changed appropriately to use the corresponding pool calls. Finally all function invocations are modified to pass in the extra pool descriptor arguments (see invocations of `g()`, `create_10_Node_list()`, `free_all.but_head()` in Figure 3.6).

The Automatic Pool Allocation transformation by itself *does not ensure program safety*. Explicit

deallocation via `poolfree` can return freed memory to its pool and then back to the system, which can then allocate it to a different pool. Use of dangling pointers to the freed memory could allow data of arbitrary types to be accessed and violate memory safety.

Chapter 4

SAFECode: Sound Analysis and Memory Safety

Alias information, type information, and call graphs are the fundamental building blocks for many kinds of static analysis tools, including model checkers and error checking tools. For programs written in weakly typed languages, however, these fundamental building blocks may not be valid if the program performs any illegal memory operations such as array bound violations, dangling pointer dereferences, and references using uninitialized pointers, because these unsafe operations can overwrite memory locations in ways not predicted by the compiler. This means that even tools that aim to provide sound results with no false negatives [34, 18] cannot guarantee that they do so. In fact, software validation tools usually assume that such memory corruption cannot occur, e.g. they assume malloc always returns fresh memory (so dangling pointer references cannot occur) and that memory allocations are logically infinitely apart (so a buffer overflow cannot trample any other allocation). This problem is potentially important because many software validation tools today are used to detect security vulnerabilities or identify logical errors in important system software.

In this chapter, we describe a novel, automatic approach an ordinary compiler can use to ensure three key analysis results — namely, a points-to graph, a call graph, and available type information — are sound, i.e., will not be invalidated by any possible memory errors, even undetected errors such as dangling pointer dereferences. We then later discuss how this also enforces memory safety.

The inputs to our approach are:

1. A program written in C;
2. The results of a flow-insensitive, field-sensitive, and unification-based pointer analysis on that

program. This includes both points-to information and type information for a subset of memory objects. The analysis may use various forms of context-sensitivity.

3. A call graph computed for the program.

Our goal is to enforce the correctness of these analyses for all executions of the program. We do not concern ourselves with how these analysis results are actually computed; we only assume that they are given in the format described earlier in Chapter 3.3.

4.1 Insights

We first give an informal overview of the our approach, focusing on four key insights we exploit in this work.

The goal of our work is to ensure that memory errors (e.g., dangling pointer references after a free, array bounds violations, etc.) do not invalidate the points-to information, call graph, or type information computed by the compiler. The major challenge is enforcing points-to information; type information follows directly from this. The call graph is simply checked explicitly at each indirect call site (see section 4.4 for a discussion on eliminating some of the run-time checks at indirect call sites).

Note that a node in a points-to graph (or the storage shape graph) is just a static representation of a set of dynamic memory objects. If these memory objects are scattered about in memory (as is usually the case), it is prohibitively expensive to check that a pointer actually points to a memory object corresponding to its target node (i.e., the pointer has not been corrupted by some memory error). As noted earlier, however, our transformation called *Automatic Pool Allocation* partitions the heap into regions based on a points-to graph [46]. This leads us to the following new insight that is the key to the current work:

[Insight1]: *If memory objects corresponding to each node in the points-to graph are located in a (compact) region of the heap, we could check efficiently at run-time that the target of a pointer is a valid member of the compile-time points-to set for that pointer, i.e., that alias analysis is not invalidated.*

Note that this insight relies on the property that unaliasable memory objects are not allocated

within the same region, which is not usually guaranteed by previous region-based systems [63, 29].

Non-heap (i.e., global and stack) objects may be in the same or different points-to sets as heap objects. We can simply include such objects in the set of address ranges for the appropriate pool (but many stack objects can be handled more efficiently as described in Section 4.4). Overall, the operation `poolcheck(ph, A, o)` verifies that the address, A , is contained within the set of memory ranges assigned to pool, ph , and has the correct alignment for the pool’s data type (or for the field at offset o if $o \neq 0$).

Even with the above partitioning of memory, checking every pointer dereference (or every pointer definition) would be prohibitively expensive. The second insight allows us to eliminate a large number of the run-time checks:

[Insight2]: *Any initialized pointer read from an object in a TK region or from an allocation site, will hold a valid address for its target region. All other pointers, i.e., pointers derived from indexing operations, and pointers read from TU regions (including function pointers), need run-time checks before being used.*

Intuitively, in the absence of dangling pointer errors and array indexing errors, an initialized pointer obtained from a TK region will always be valid; it cannot have been corrupted in an unpredictable way e.g. via arbitrary casts and subsequent stores (it would then be obtained from a TU region).

Uninitialized pointers and array indexing errors are addressable via run-time checks. Dangling pointer references, however, are difficult to detect in general programs, and we do not attempt to detect or prevent such errors. Instead, we ensure that such errors do not invalidate the results of alias analysis by using the following insight:

[Insight3]: *In a TK (type-homogeneous) region, if a memory block holding one or more objects were freed and then reallocated to another request in the same region with the same alignment, then dereferencing dangling pointers to the previous freed object cannot cause either a type violation or an aliasing violation.*

Essentially, we make sure that, if a dangling pointer to freed memory points into a newly allocated object, the old and new objects have the same static type and that any pointers they contain have identical aliasing properties. Thus loads or stores using the dangling pointers may

give unexpected results but cannot trample memory outside the expected pool.

This principle allows free memory to be reused *within* the same region unlike other region-based languages, which either disallow such reuse [63] or allow it only in restricted cases [35, 62]. For reuse across regions Automatic Pool Allocation already provides us a solution:

[Insight4]: *We can safely release the memory of a region when there are no reachable pointers into that region.*

This gives us a way to release memory to the system. Since Automatic Pool Allocation already binds the life times of regions (using escape analysis), we can arrange for memory to be released at the end of a region’s life time.

Finally, in order to prove the correctness of our approach, we formalize the key properties of our regions by extending the previous type system encoding points-to information (described in Chapter 3.3) in two ways: (1) to encode regions corresponding to points-to sets, with allocation and deallocation out of these regions; and (2) to encode information about region lifetimes. The type system is designed to be mostly statically checkable for the correctness of encoded types (i.e. the points-to relations, lifetimes, and the call graph). We borrow a key idea from Tofte and Talpin’s work on regions for ML [63] to simplify the type system, namely, we restrict region lifetimes to be lexically scoped (others have shown that this is not strictly necessary [3, 29]).

4.2 SAFECODE Type System

We now give a formal description of the SAFECODE type system.

4.2.1 Syntax

Figure 4.1 gives the syntax of the language in our type system. This syntax, which forms the input to our type checker, includes new constructs for encoding region handles, region lifetimes, region allocation and deallocation, and separate versions of `load/store` that require run-time checks. The `associate` statement of Figure 3.1 is now transformed to the `poolinit` statement along with a lexical scope indicating where the association is valid, essentially creating a lifetime for the corresponding region. For example, statement `poolinit(ρ, τ) x_ρ { S }`, creates a region named ρ that can hold objects of type τ , with the handle x_ρ . Our typing rules, described in Section 4.2.2,

RegionVar		ρ
Var		$x \quad y$
Types	$\tau ::=$	<code>int</code> <code>char</code> <code>Unknown</code> $\tau * \rho$ <code>handle</code> (ρ, τ)
Statements	$S ::=$	<code>ε</code> <code>S</code> ; <code>S</code> <code>x = E</code> <code>store</code> E, E <code>storeToU</code> x, E, E <code>storec</code> E, E <code>storecToU</code> E, E <code>poolfree</code> (E, E) <code>poolinit</code> (ρ, τ) $x \{ S \}$ <code>pool</code> { S } <code>pop</code> (ρ)
Expressions	$E ::=$	<code>var</code> V $E \text{ op } E$ <code>load</code> E <code>loadFromU</code> x, E <code>loadc</code> E <code>loadcFromU</code> E <code>cast</code> E to τ <code>poolalloc</code> (x, E) ($x, \&E[E]$) <code>castint2pointer</code> x, E to τ
Value	$V ::=$	<code>Uninit</code> <code>Int</code> <code>region</code> (ρ)

Figure 4.1: Our syntax

make it illegal to store an object of type other than τ in this region. The type of the region handle x_ρ is `handle`(ρ, τ). Notice that regions in our system are nested and a region can only contain objects of one type although this type may have to be *Unknown* for some regions. The lexical scoping, along with region attributes for pointers, allow the type checker to ensure that an object in a region cannot be accessed outside the lifetime of the region. Although this seems to disallow cycles in points-to graph, extensions for handling them are straightforward and discussed later in section 4.4.

Calls to `malloc` and `free` in the input program are replaced by calls to `poolalloc` and `poolfree`. `poolalloc` takes in a handle to the region as an argument and allocates an object (or an array of objects) out of the region. The type of an allocated object (or array element) is the type associated with the region. The `poolfree` statement frees a memory object and releases the memory back to the region. *Uninit* essentially represents the NULL value in C. The `castint2ptr`, `loadFromU`, `storeToU` are versions of `cast`, `load`, `store` that require various run-time checks. Other than uninitialized pointer checks, the only operations that require a run-time check are those that take in a pool handle as an argument.

Everything else in the syntax including `pool`{ S }`pop`(ρ) , `region`(ρ) are not part of the source language but needed for operational semantics and are described in section 4.2.3.

The Figure 4.2 shows the running example (Figure 3.4) in this new syntax. The `storeToU` and `loadFromU` operations are versions of `store` and `load` that need run-time checks. The `associate` is now replaced by `poolinit`, binding the lifetime of the pools.

```

int *r2 *r1 x;
int u, *r2 y
int *r3 z;
int *r2*r1*r3 w;
poolinit(r2, int) r2handle {
  poolinit(r1, int *r2) r1handle {
    poolinit(r3, Unknown) r3handle {
      x = poolalloc(r1handle, 1);
      y = poolalloc(r2handle, 1);
      z = poolalloc(r3handle, 1);
      store y, x;      store 5, y;
      poolfree(r3handle,z); //dangling pointer exists
      storeToU r3handle, 10, z ;
      ...
      u = loadFromU r3handle, z ;
      ...
      w = cast z to (int *r2*r1*r3);
      storeToU r3handle, x , w ; //type checks as region of r3
      ... //is Unknown
    }
  }
}

```

Figure 4.2: Running example in our type system

We use Automatic Pool Allocation transformation to take an input program including the pointer analysis annotations described in chapter 3.3 and produce a program with region types and region allocation.

4.2.2 Typing Rules

The type system is expressed by the following three judgments: $C \vdash e : \tau$ (for expression typing), $C \vdash S$ (for statement typing), and $C \vdash \tau$ (for type typing).

In these judgments C , the typing context, is a pair of typing environments $(\Gamma; \Delta)$ where Γ is a map between variable names and their types (built up using the variable declarations) and Δ is a map between region names and the type of objects stored in the region (built up using `poolinits`). We present the typing rules for our language in Figures 4.3, 4.4, and 4.5.

While many of the type rules are similar to those of C, some type rules are unique to our approach and require further explanation. (SS4) and (SS14) type loads/stores using pointers to

$$\begin{array}{l}
(\text{SS0}) \frac{C \vdash \tau \quad \Gamma(x) = \tau}{C(= \Gamma, \Delta) \vdash x : \tau} \\
(\text{SS1}) \frac{}{C \vdash n : \text{int}} \\
(\text{SS2}) \frac{C \vdash e1 : \text{int} \quad C \vdash e2 : \text{int}}{C \vdash e1 \text{ op } e2 : \text{int}} \\
(\text{SS3}) \frac{C \vdash \tau}{C \vdash \text{Uninit} : \tau} \quad \tau \neq \text{handle}(\rho', \tau') \\
(\text{SS4}) \frac{C \vdash e : \tau * \rho \quad C \vdash \rho : \tau \quad \tau \notin \{\text{Unknown}, \text{char}\}}{C \vdash \text{load } e : \tau} \\
(\text{SS4char}) \frac{C \vdash e : \text{char} * \rho \quad C \vdash \rho : \text{char}}{C \vdash \text{loadc } e : \text{char}} \\
(\text{SS5}) \frac{C \vdash e : \tau * \rho \quad C \vdash \rho : \text{Unknown} \quad C \vdash x : \text{handle}(\rho, \text{Unknown})}{C \vdash \text{loadFromU } x, e : \text{int}} \\
(\text{SS5char}) \frac{C \vdash e2 : \tau * \rho \quad C \vdash \rho : \text{Unknown} \quad C \vdash x : \text{handle}(\rho, \text{Unknown})}{C \vdash \text{loadcFromU } x, e2 : \text{char}} \\
(\text{SS6}) \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e : \text{int}}{C \vdash \text{poolalloc}(x, e) : \tau * \rho} \\
(\text{SS7}) \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e : \text{int}}{C \vdash \text{castint2ptr } x, e \text{ to } \tau * \rho : \tau * \rho} \\
(\text{SS8}) \frac{C \vdash \tau' \quad C \vdash e : \tau * \rho}{C \vdash \text{cast } e \text{ to } \tau' * \rho : \tau' * \rho} \\
(\text{SS9}) \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : \tau * \rho \quad C \vdash e3 : \text{int}}{C \vdash x, \&e2[e3] : \tau * \rho} \\
(\text{SS10}) \frac{C \vdash e : \tau}{C \vdash \text{cast } e \text{ to } \text{int} : \text{int}} \quad \tau \neq \text{handle}(\rho, \tau')
\end{array}$$

Figure 4.3: Expression typing judgments

$$\begin{array}{c}
\text{(SS11)} \frac{}{C \vdash \epsilon} \\
\text{(SS12)} \frac{C \vdash s1 \quad C \vdash s2}{C \vdash s1; s2} \\
\text{(SS13)} \frac{C \vdash x : \tau \quad C \vdash e : \tau}{C \vdash x = e} \\
\text{(SS14)} \frac{C \vdash \rho : \tau \quad C \vdash e1 : \tau * \rho \quad C \vdash e2 : \tau \quad \tau \notin \{Unknown, \text{char}\}}{C \vdash_{\text{store}} e2, e1} \\
\text{(SS14char)} \frac{C \vdash \rho : \text{char} \quad C \vdash e1 : \rho * \text{char} \quad C \vdash e2 : \text{char}}{C \vdash_{\text{storec}} e2, e1} \\
\text{(SS15)} \frac{C \vdash \rho : Unknown \quad C \vdash e1 : \tau * \rho \quad C \vdash e2 : \tau \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash_{\text{storeToU}} x, e2, e1} \\
\text{(SS15char)} \frac{C \vdash \rho : Unknown \quad C \vdash e1 : \tau * \rho \quad C \vdash e2 : \text{char} \quad C \vdash x : \text{handle}(\rho, Unknown)}{C \vdash_{\text{storecToU}} x, e2, e1} \\
\text{(SS16)} \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : \tau * \rho}{C \vdash_{\text{poolfree}}(x, e2)} \\
\text{(SS17)} \frac{C \vdash \tau \quad \Gamma[x \mapsto \text{handle}(\rho, \tau)], \Delta[\rho \mapsto \tau] \vdash s \quad x \notin \Gamma \text{ and } \rho \notin \Delta}{C(= \Gamma, \Delta) \vdash_{\text{poolinit}}(\rho, \tau)x\{s\}}
\end{array}$$

Figure 4.4: Statement typing judgments

type consistent memory (TK pools). They check that the type of the objects in the pool matches the type of the pointer operand. (SS5) is for loads using pointers to untyped *Unknown* memory (TU pools); note that we get back an int. (SS7) allows a cast from int to pointer type. As discussed later in the operational semantics, such a cast requires a run-time check to make sure that the pointer is of the right type in the right pool. This coupled with (SS5) above enables loading pointers from TU pools safely. (SS15) types stores to *Unknown* memory. (SS8) types a cast from a pointer to a region to another pointer pointing to the same region. This helps in supporting arbitrary casts of pointer types as long as they have the same region attribute, *without* requiring run-time check; note that (SS4) and (SS14) require that a pointer be cast back to the type of objects in the region before use. (SS17) is for creating a region using `poolinit`; we add the region variable and the handle to the typing context before checking the body of the `poolinit`. (SS6) gives a type for the memory objects allocated in a pool. (SS16) frees objects only when they belong to the appropriate pool.

$$\begin{array}{l}
(\text{SS18}) \frac{}{C \vdash \text{int, char}} \\
(\text{SS20}) \frac{\Delta(\rho) = \tau}{C \vdash \rho : \tau} \\
(\text{SS22}) \frac{\Delta(\rho) = \text{Unknown}}{C(= \Gamma, \Delta) \vdash \tau * \rho}
\end{array}
\qquad
\begin{array}{l}
(\text{SS19}) \frac{}{C \vdash \text{Unknown}} \\
(\text{SS21}) \frac{\vdash \rho : \tau}{C \vdash \tau * \rho} \\
(\text{SS23}) \frac{C \vdash \rho : \tau}{C \vdash \text{handle}(\rho, \tau)}
\end{array}$$

Figure 4.5: Well formed types

4.2.3 Operational Semantics

The operational semantics rules for our language provide a formal basis for reasoning about program behavior even in the presence of problems P1-P4. They essentially describe the run-time checks needed to enforce the correctness of alias analysis. The rules are listed in Figures 4.7, 4.8. and 4.9.

Figure 4.6 lists the environments necessary to describe the operational semantic rules. The rules are described as a small-step operational semantics, \longrightarrow_{expr} for expressions and \longrightarrow_{stmt} for statements. Each program state is represented by $(VEnv, L, es)$ where $VEnv$ is the variable environment (partial map holding the values of variables), L is the partial map of live regions and the corresponding store, and es is an expression or statement in the program. H , the system heap, contains the memory addresses not in use by the program. H is a part of the program state but not included in the notation for the sake of brevity. A program state $(VEnv, L, es)$ becomes $(VEnv', L', es')$ if any of the semantic rules allow for it. The expression in the box, if any, is a run-time check that is executed before the corresponding rule. If the run-time check fails, then the program state becomes specially designated Error state.

We assume that $region(\rho)$ is the handle for a region named ρ . A region (see Figure 4.6) is defined as a tuple $\{ F ; RS \}$: F is a list of freed memory locations within the region, and RS (the region store) is a partial map between memory addresses and their values.

Briefly, the four memory errors **P1-P4** listed in Chapter 3.2 are solved as follows. **P1** is solved using the type homogeneity principle, as explained previously. This is implemented by rules **R14**, **R34**, and the static typing rules that check operations on pointers to known-type pools. We detect problem **P2** using the run-time check on rule **R40**. To detect **P3**, we initialize all newly created memory and all local variables to *Uninit* and check *Uninit* pointers via rules **R6**, **R14**, and **R23**. Issue **P4** is detected using **R31**.

Below we describe in more detail the rules that are unique to our approach.

1. (**R15,R17**): Evaluating `poolinit` creates a new region, sets the free list to be empty, and evaluates the body inside the syntactic construct `pool{S}pop(ρ)`. This construct identifies when the region needs to be deallocated, i.e., when the body (S) becomes empty. This is performed by rule R17.
2. (**R6**): Performs a store via a type-consistent pointers, after checking that v_1 is not *Uninit*. `update(L, v_1, v_2)` just updates the memory location v_1 with value v_2 . Loads via type consistent memory have a similar check for uninitialized pointers.
3. (**R10**): Performs a store via a pointer to *Unknown* memory, after checking that the pointer value legally allows storing of a 4-byte value (an int). Note that our proof below guarantees that $v \in \text{Dom}(L[\rho].\text{RS})$, so at run-time it is enough to check for the open interval $(v_1, v_1 + 3]$.
4. (**R14**): Frees an object from region, ρ , and adds it to the free list F of the same region.
5. (**R34**): Returns a previously freed location from the free list. Together with **R14**, this implements the type homogeneity principle to make error **P1** harmless.
6. (**R35**): For a `poolalloc`, when the free list is empty, this requests fresh memory from the system. `poolalloc` aborts if it cannot allocate requisite memory.
7. (**R31**): A cast from int to another pointer type is always checked at run-time using a `poolcheck`, i.e., we check that the value is a properly aligned address in the appropriate pool for the pointer type, and if not, we abort. This detects problem **P4**.
8. (**R40**): For array indexing, we check that the resultant pointer after the arithmetic always points to the same pool as the source pointer at the proper alignment. These checks are not exact array bounds checks but a much coarser check for the pool bounds. This means some array bound violations may go undetected.

The complete list of run-time checks in our system are the checks in the boxes in Figures 4.7, 4.8, and 4.9 along with checks on casts from integer to function pointers. Note that the `poolcheck` operation described earlier in this section is exactly the same as check given in **R31**. None of the

VarEnv	VEnv	:	Var \longrightarrow Value
Region	R	::=	{ F ; RS }
RegionStore	RS	:	Int \longrightarrow Value
FreeList	F	::=	ϕ aF
LiveRegions	L	:=	RegionVar \longrightarrow RS
SystemHeap	H	\subseteq	Int ₃₂

Figure 4.6: Environments for operational semantics

run-time checks require any metadata on individual pointer variables (usually required for precise array bounds checks) or runtime tag bits on any memory locations (usually required for RTTI or to track legal pointer values). The only metadata we require at runtime is available in the pool descriptor (handle), which is known at compile time.

4.2.4 Soundness Proof

The proof of soundness is composed of two “invariant preservation” theorems — one for expressions and the other for statements of the program. Since we have not included control flow in our formalization, all evaluations of expressions and statements terminate. A detailed proof of our technique is included in the next Chapter 5. Here we just summarize the important invariants that our approach maintains at each step of the operational semantics and state the soundness theorem for statements.

First, for an environment $(VEnv, L)$, we define $\|\tau\|_{(VEnv, L)}$ to be as follows:

$$\begin{aligned}
\|\mathbf{int}\|_{(VEnv, L)} &:= \mathbf{Int}_{32} \\
\|\tau * \rho\|_{(VEnv, L)} &:= \{Uninit\} \cup \text{Dom}(L[\rho].\mathbf{RS}) \\
\|\mathbf{handle}(\rho, \tau)\|_{(VEnv, L)} &:= \{\text{region}(\rho)\} \\
\|\mathbf{Unknown}\|_{(VEnv, L)} &:= \mathbf{Int}_8 \\
\|\mathbf{char}\|_{(VEnv, L)} &:= \mathbf{Int}_8
\end{aligned}$$

Intuitively, for a well-formed type τ , $\|\tau\|_{(VEnv, L)}$ represents the set of values that a variable (or object) of that type can hold under that context and environment. For example, for a variable of type $\tau * \rho$, the set of values it can hold is either *Uninit* or addresses of objects in region ρ , which is $\text{Dom}(L[\rho].\mathbf{RS})$.

Let \vdash_{env} denote the judgment for a well formed environment. We defined an environment $(VEnv, L)$ to be well formed under a typing context C (denoted by $C \vdash_{env} (VEnv, L)$) if and only if the following invariants hold.

- R1**
$$\frac{(\text{VEnv}, L, S1) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', S1')}{(\text{VEnv}, L, S1 ; S2) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', S1' ; S2)}$$
- R2**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, x = E) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', x = E')}$$
- R3**
$$(\text{VEnv}, L, x = v_1) \longrightarrow_{\text{stmt}} (\text{VEnv}[x \mapsto v_1], L, \epsilon)$$
- R4**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{store}/\text{storec } E, E_2) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{store } E', E_2)}$$
- R5**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{store}/\text{storec } v, E) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{store } v, E')}$$
- R6**
$$(\text{VEnv}, L, \text{store}/\text{storec } v_2, v_1) \longrightarrow_{\text{stmt}} (\text{VEnv}, \text{update}(L, v_1, v_2), \epsilon) \quad \boxed{(v_1)! = \text{Uninit}}$$

 where

$$\text{update}(L, v_1, v_2) := L' \cup \{(\rho, \{ \text{R.F} ; \text{R} . (\text{RS}[v_1 \mapsto v_2]) \})\} \text{ if } \exists \rho \in \text{Dom}(L) \text{ s.t. } L = L' \cup \{(\rho, \text{R})\} \text{ and } v_1 \in \text{Dom}(\text{R} . \text{RS})$$

$$L \text{ else}$$
- R7**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{storeToU}/\text{storecToU } E, E_2, E_3) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{storeToU}/\text{storecToU } E', E_2, E_3)}$$
- R8**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{storeToU}/\text{storecToU } v_1, E, E_3) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{storeToU}/\text{storecToU } v_1, E', E_3)}$$
- R9**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{storeToU}/\text{storecToU } v_1, v_2, E) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{storeToU}/\text{storecToU } v_1, v_2, E')}$$
- R10**
$$(\text{VEnv}, L, \text{storeToU } \text{region}(\rho), v_2, v_1) \longrightarrow_{\text{stmt}} (\text{VEnv}, \text{update}(L, v_1, v_2, 4) \}], \epsilon)$$

$$\boxed{(v_1, v_1 + 3) \in \text{Dom}(L[\rho] . \text{RS})}$$

 where

$$\text{update}(L, v_1, v_2, 4) := L' \cup \{(\rho, \{ \text{R.F} ; \text{R} . (\text{RS}[v_1 \mapsto \text{byte}(v_2, 3)]$$

$$\quad \quad \quad [(v_1+1) \mapsto \text{byte}(v_2, 2)][(v_1+3) \mapsto \text{byte}(v_2, 1)][(v_1+4) \mapsto \text{byte}(v_2, 0)])$$

$$\quad \quad \quad \text{if } \exists \rho \in \text{Dom}(L) \text{ s.t. } L = L' \cup \{(\rho, \text{R})\} \text{ and } [v_1, v_1 + 3] \in \text{Dom}(\text{R} . \text{RS})$$

$$L \text{ else}$$

 and $\text{byte}(n, k) := (n \ll (8 * (3 - k))) \gg 24$.
- R11**
$$(\text{VEnv}, L, \text{storecToU } \text{region}(\rho), v_2, v_1) \longrightarrow_{\text{stmt}} (\text{VEnv}, \text{update}(L, v_1, v_2) \}], \epsilon)$$
- R12**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{poolfree}(E, E_2) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{poolfree}(E', E_2))}$$
- R13**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{poolfree}(v, E) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{poolfree}(v, E'))}$$
- R14**
$$(\text{VEnv}, L \cup \{(\rho, \{F; RS\})\}, \text{poolfree}(\text{region}(\rho), v)) \longrightarrow_{\text{stmt}} (\text{VEnv}, L \cup \{(\rho, \{vF; RS\})\}, \epsilon)$$

$$\boxed{v! = \text{Uninit}}$$
- R15**
$$(\text{VEnv}, L, \text{poolinit}(\rho, \tau)x\{S\}) \longrightarrow_{\text{stmt}} (\text{VEnv} \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, \{\phi; \phi\})\}, \text{pool}\{S\}\text{pop}(\rho))$$

 if $(\rho \notin \text{Dom}(L))$.
- R16**
$$\frac{(\text{VEnv}, L, S) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', S')}{(\text{VEnv}, L, \text{pool}\{S\}\text{pop}(\rho)) \longrightarrow_{\text{stmt}} (\text{VEnv}', L', \text{pool}\{S'\}\text{pop}(\rho))}$$
- R17**
$$(\text{VEnv} \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, \text{R})\}, \text{pool}\{\epsilon\}\text{pop}(\rho)) \longrightarrow_{\text{stmt}} (\text{VEnv}, L, \epsilon)$$

 Note that H the set of addresses in the system heap and not used by the program gets updated by $H \cup \text{Dom}(\text{R} . \text{RS})$

Figure 4.7: Operational semantic rules for statements

- R18** $(\text{VEnv} \cup \{(x, v)\}, L, x) \longrightarrow_{\text{expr}} (\text{VEnv} \cup \{(x, v)\}, L, v)$
- R19**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, E \text{ op } E_2) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E' \text{ op } E_2)}$$
- R20**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, v \text{ op } E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', v \text{ op } E')}$$
- R21** $(\text{VEnv}, L, m \text{ op } n) \longrightarrow_{\text{expr}} (\text{VEnv}, L, m \text{ op}_{Int} n)$
- R22**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{load/loadc } E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{load/loadc } E')}$$
- R23** $(\text{VEnv}, L, \text{load/loadc } v_1) \longrightarrow_{\text{expr}} (\text{VEnv}, L, \text{getvalue}(L, v_1))$ $\boxed{(v_1)! = Uninit}$ where
 $\text{getvalue}(L, v_1) := \begin{array}{l} L[\rho].\text{RS}[v_1] \text{ if } \exists \rho \in \text{Dom}(L) \text{ s.t. } v_1 \in L[\rho].\text{Dom}(\text{RS}) \\ Uninit \text{ else} \end{array}$
- R24**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{loadFromU/loadcFromU } E, E_2) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{loadFromU/loadcFromU } E', E_2)}$$
- R25**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{loadFromU/loadcFromU } v_1, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{loadFromU/loadcFromU } v_1, E')}$$
- R26** $(\text{VEnv}, L, \text{loadFromU } \text{region}(\rho), v_1) \longrightarrow_{\text{expr}} (\text{VEnv}, L, \text{getvalue}(L, v_1, 4))$ $\boxed{(v_1, v_1 + 3) \in \text{Dom}(L[\rho].\text{RS})}$
where
 $\text{getvalue}(L, v_1, 4) := \begin{array}{l} \text{combine}(L[\rho].(\text{RS}[v_1]), L[\rho].(\text{RS}[v_1 + 1]), L[\rho].(\text{RS}[v_1 + 2]), L[\rho].(\text{RS}[v_1 + 3])) \\ \text{if } \exists \rho \in \text{Dom}(L) \text{ s.t. } [v_1, v_1 + 3] \in L[\rho].\text{Dom}(\text{RS}) \\ Uninit \text{ else} \end{array}$
and $\text{combine}(b1, b2, b3, b4) := (b1 \ll 24) \parallel (b2 \ll 16) \parallel (b3 \ll 8) \parallel (b4)$.
- R27** $(\text{VEnv}, L, \text{loadcFromU } \text{region}(\rho), v_1) \longrightarrow_{\text{expr}} (\text{VEnv}, L, \text{getvalue}(L, v_1))$
- R28** $(\text{VEnv}, L, \text{cast } E \text{ to } \tau) \longrightarrow_{\text{expr}} (\text{VEnv}, L, E)$
- R29**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{castint2ptr } E, E_2 \text{ to } \tau) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{castint2ptr } E', E_2 \text{ to } \tau)}$$
- R30**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{castint2ptr } v, E \text{ to } \tau) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{castint2ptr } v, E' \text{ to } \tau)}$$
- R31** $(\text{VEnv}, L, \text{castint2ptr } (\text{region}(\rho), v \text{ to } \tau) \longrightarrow_{\text{expr}} (\text{VEnv}, L, v)$ $\boxed{v \in \text{Dom}(L[\rho].\text{RS})}$
- R32**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{poolalloc}(E, E_2)) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{poolalloc}(E', E_2))}$$
- R33**
$$\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, \text{poolalloc}(v, E)) \longrightarrow_{\text{expr}} (\text{VEnv}', L', \text{poolalloc}(v, E'))}$$
- R34** $(\text{VEnv}, L \cup \{(\rho, \{a \ F; \text{RS}\})\}, \text{poolalloc}(\text{region}(\rho), 1)) \longrightarrow_{\text{expr}} (\text{VEnv}, L \cup \{(\rho, \{F; \text{RS}\})\}, a)$
- R35** $(\text{VEnv}, L \cup \{(\rho, \{\phi; \text{RS}\})\}, \text{poolalloc}(\text{region}(\rho), 1)) \longrightarrow_{\text{expr}} (\text{VEnv}, L[\rho \mapsto \{\phi; \text{RS}[a \mapsto Uninit]\}], a)$
where a is a new address obtained from system allocator, i.e. $a \in H$. H becomes $H - \{a\}$.

Figure 4.8: Operational semantic rules for expressions - I

- R36** $(\text{VEnv}, L \cup \{ (\rho, \{F; RS\}) \}, \text{poolalloc}(\text{region}(\rho), m)) \longrightarrow_{\text{expr}} (\text{VEnv}, \text{Initialize}(L \cup \{ (\rho, \{F; RS\}) \}, \text{Uninit}, a, m), a)$ if $(m \neq 1)$
 where a is a new address for the array obtained from system allocator and `Initialize` initializes each element of the array with `Uninit`. H becomes $H - \{ a, a + 1, \dots, a + m - 1 \}$
- R37** $\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, (E, \&(E_1)[E_2])) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E', \&(E_1)[E_2])}$
- R38** $\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, (v, \&(E)[E_2])) \longrightarrow_{\text{expr}} (\text{VEnv}', L', v, \&(E')[E_2])}$
- R39** $\frac{(\text{VEnv}, L, E) \longrightarrow_{\text{expr}} (\text{VEnv}', L', E')}{(\text{VEnv}, L, (v, \&(v_1)[E])) \longrightarrow_{\text{expr}} (\text{VEnv}', L', (v, \&(v_1)[E']))}$
- R40** $(\text{VEnv}, L, (\text{region}(\rho), \&v1[v2])) \longrightarrow_{\text{expr}} (\text{VEnv}, L, v1 + v2 * \text{sizeof}(\tau))$
 $\boxed{(v1 + v2 * \text{sizeof}(\tau)) \in \text{Dom}(L[\rho].RS)}$
 where τ is the “static” type of the individual element of the array, available from the declaration.
 Note that $\text{sizeof}(\tau)$ is a compile time constant.

Figure 4.9: Operational semantic rules for expressions - II

Inv1 $\text{Dom}(\Gamma) = \text{Dom}(\text{VEnv})$

All variables in the typing environment are present in the variable environments and vice versa.

Inv2 $\text{Dom}(\Delta) = \text{Dom}(L)$

All region names in the region type environment are already present in the domain of region maps and vice versa.

Inv3 $\forall x \in \text{Dom}(\text{VEnv}), \text{if } C \vdash x : \tau \text{ then } \text{VEnv}[x] \in \|\tau\|_{(\text{VEnv}, L)}$

If a variable has type τ , then it must contain only valid values of type τ . In particular, a pointer variable with region attribute ρ , must always point to an object in that region or it has the value `Uninit`.

Inv4 $\forall \rho \in \text{Dom}(L), \text{if } C \vdash \rho : \tau \text{ then } \forall v \in \text{Dom}(L[\rho].RS), L[\rho].RS[v] \in \|\tau\|_{(\text{VEnv}, L)}$.

If region ρ is associated with type τ then each memory location in the region store will only contain values of the correct type.

Inv5 $\forall \rho \in \text{Dom}(L), L[\rho].F \subseteq \text{Dom}(L[\rho].RS)$

This invariant states that the memory addresses in the free list are a subset of the addresses of the region.

Inv6 $\forall \rho_1 \rho_2 \in \text{Dom}(L)$, if $\rho_1 \neq \rho_2$ then $\text{Dom}((L[\rho_1]).\text{RS}) \cap \text{Dom}((L[\rho_2]).\text{RS}) = \phi$ and $\forall \rho \in \text{Dom}(L)$, $\text{Dom}(L[\rho].\text{RS}) \cap H = \phi$.

A memory address cannot be part of two live regions. Also, a memory address cannot be a part of both system heap (i.e., unused by a program) and live region.

Now assume that a run-time check failure leads to the Error state in the operational semantics. We prove the following soundness theorem:

Theorem 1 *If $\Gamma \vdash S$ and $\Gamma \vdash_{env} (VEnv, L)$ then either $(VEnv, L, S) \longrightarrow_{stmt}^* \text{Error}$ or $(VEnv, L, S) \longrightarrow_{stmt}^* (VEnv', L', \epsilon)$ and $C \vdash_{env} (VEnv', L')$.*

Proof: The proof for this theorem is by induction on the structure of typing derivations (Chapter 5).

The soundness result gives us the following invariant – “For a well typed program containing pointer variable p whose declared type is $\tau * \rho$, in every execution state the value of p is guaranteed to be a pointer to an object in the region ρ ”. This holds even in the presence of undetected memory errors like dangling pointer dereferences and array bound violations, and thus it guarantees correctness of the aliasing information induced by our type system.

Similarly the function pointer extensions to the type system described in Section 4.4 along with a soundness theorem for the extended type system guarantee the correctness of the call graph.

4.3 SAFECODE Guarantees

4.3.1 Sound Analysis Guarantee

The soundness theorem above immediately gives the sound analysis guarantee, i.e., SAFECODE ensures that no memory error can invalidate the alias analysis, call graph, or type information for a subset of programs.

4.3.2 Memory Safety Guarantee

The soundness proof also entails that SAFECODE provides memory safety guarantee (defined in chapter 2.1) to well typed programs. This is because SAFECODE renders memory errors harmless,

FunctionType ft	:=	$\tau \rightarrow \tau$ $\tau \rightarrow \text{Unit}$ $\forall \rho. \text{ft}$ $\text{ft} \langle \rho \rangle$
new Types	:=	τ ft Unit
Functions F	:=	$\tau' f(x : \tau) \{ S \}$ $\langle \rho, \tau \rangle F$
Instantiation finst	:=	$f x $ finst $\langle \rho \rangle$
new Statements	:=	S call finst (x)

Figure 4.10: Syntactic extensions for some of the remaining C constructs

ensures control-flow integrity and ensure that memory accesses respect the type of the objects being accessed.

4.3.3 Security Guarantee

As noted in Chapter 2.3.2, the SAFECODE solution described thus far, has the following important security properties:

- No security attack can subvert the control flow of a program and execute arbitrary injected code. This prevents many attacks that rely on executing injected code including common buffer-overflow attacks, heap corruption attacks.
- Attacks that rely on library being loaded at certain program points (“return-to-libc” attacks [68]) are not possible.

However, SAFECODE in this mode does not detect dangling pointer errors. Consequently, SAFECODE may not be able to detect attacks that are based on corrupting data but not control-flow (called non-control data attacks [11]). We believe that with SAFECODE these attacks become much harder to exploit, since any data corruption that can occur in SAFECODE is confined within a pool and on values of the same type. Nevertheless, there may be some situations, especially server applications, where this guarantee is not enough. We developed a dangling pointer detection mechanism (discussed later in chapter 7) that we show has low overhead in server applications and can be used to provide a much stronger security guarantee for such applications.

4.4 Extensions for Full C

Several constructs of C were omitted in the previous section to explain our core ideas. Our type system and semantics correctly handle all constructs in C and so does our implementation. In

this section, we informally discuss how we handle the remaining constructs including function calls, function pointers and support for region polymorphic functions. Some of the ideas for implementing region polymorphism in functions and structs are directly borrowed from Cyclone [29]. However it is worth noting that our universal types are quantified only over region type variables (and not arbitrary type variables). This is sufficient in our domain of trying to retrofit polymorphic region types to existing non-polymorphic C code.

4.4.1 Structure Types

Structures types are like in C and the syntax for structures is shown in Figure 3.2. A pointer can point into a structure at an offset $n \geq 0$ and we use $\tau * (\rho, n)$ to denote the type of such a pointer (n is a compile time constant). Given this, the only extra safety implications of structure types are that (a) the `poolcheck` must use the offset o in checking alignment, and (b) structure indexing operations for pointers to TU regions need a poolcheck (similar to array indexing). A notational issue is that it is convenient to include polymorphic type constructors, similar to those used in Cyclone [29]. These constructors allows a struct type with a pointer field to be used in different places with the field pointing to distinct sets of objects (e.g., when two distinct linked lists are created with the same list node type). For example, the polymorphic type `struct S<rho> { Field0 : int, Field1 : int * rho }` can be instantiated with a region type variable to get a new type pointing to a particular points-to set.

4.4.2 Region-polymorphism for Functions

Like Cyclone [29], we support region polymorphic functions parameterized via region names. Region polymorphism is necessary because it is impractical to duplicate function definitions for each context in which they are used. Automatic Pool Allocation already infers this region polymorphism automatically for C programs based on points-to analysis [46]. We leverage that work and only have to type-check that the inferred polymorphism and instantiation are correct.

```

    struct Y<rho1, rho2>; //forward declaration
struct X<rho1, rho2> {
    Fld1 : struct Y<rho2, rho1>*rho2
}
struct Y<rho2, rho1> {
    Fld1 : struct X<rho1, rho2>*rho1 ;
}
...
poolinit(2, rho1, rho2,
        struct X<rho1, rho2>, struct Y<rho1, rho2>) (ph1, ph2)
{
    ....
}

```

Figure 4.11: Cycles in Points-to graph

4.4.3 Cycles in Points-to Graphs

The syntax in the core language does not allow mutually recursive types in our languages. This is clearly not acceptable for supporting general C programs. We solve this by requiring that the regions for mutually recursive data structures be created at the same lexical level. For this reason, we add a new construct to our language that enables creating multiple regions at the same lexical level. An example is shown in Figure 4.11. Here `poolinit` creates two regions `rho1`, `rho2`, such that `rho1` contains objects of type `struct X<rho2>` and vice versa. Note that we only require that the region initialization (via `poolinit`) for these data structures to be done at the same lexical level, actual memory allocation within a region is done at the same place as in the original program.

4.4.4 Function Pointers

We represent the call graph in the input type system by adding a function set attribute (called `fs` in Figure 3.2) to each function pointer type, making explicit the set of possible targets for that function pointer. The function set attribute can be initialized using the `FSET` definition. For example, the definition `FSET fs = func1, func2, func3` followed by a use `(int -> int)*fs fptr` denotes a function pointer `fptr` whose targets are the functions `func1`, `func2`, `func3`. Before an indirect call, we check at run-time if the function pointer actually points to one of the functions in its `FSET` attribute. A number of these run-time checks are unnecessary and can be eliminated using simple

static typing rules. Essentially function pointers that are read from a TU pool (via casts from `int` to function pointers), and function pointers whose targets are more precise than the one used by the Automatic Pool Allocation, will continue to require a run-time check.

4.4.5 Control Flow

Ordinary control does not require any additional safety checks. Adding typing rules and semantic rules for control flow to our language is fairly straightforward and we omit the discussion here.

4.4.6 Global and Stack Allocations

We make memory allocation for both global and stack variables explicit using operations `alloca` and `galloc` (the latter takes an optional initializer). These eliminate the need for the `&` operator for taking the address of variables. Note that a global or stack object may not have a valid pool handle if no heap object is aliased to it. To ensure that valid pool handles are created for these objects, we pretend that they are allocated using `malloc` at program entry and function entry respectively, and infer the life times of regions using Automatic Pool Allocation. Globals are still allocated in the global area just like the original program. We simply register the valid range of global addresses with the corresponding pool handle if any run-time checks are ever needed for that global pool. Stack objects whose region is created within the same function (i.e., does not escape to a parent) are allocated on the stack, like the original program. Otherwise, we allocate them using `poolalloc` at function entry and free them at function returns. In practice, we found that most stack allocations in the original program do not escape and can actually be allocated on the stack.

4.5 Limitations

We now discuss the limitations of the SAFECODE work and identify possible solutions where possible.

4.5.1 Incomplete Error Checking

The key weakness of the work we have described so far, is that it permits few array bounds violations and dangling pointer errors to go undetected but confined within a pool. To overcome the first

issue, we have also developed an array bounds checking tool that is backwards-compatible and achieves lower overheads than any of the other existing approaches for run-time bounds checking. We have integrated the bounds checker with the work here and the overheads continue to be low. This is discussed in detail in chapter 6. We have also developed a new technique to detect dangling pointer errors, discussed in chapter 7.

4.5.2 Potential Increase in Memory Usage

A second issue is that in some cases, our system might require more memory than the original C program since we cannot free memory to the system until a region goes out of scope. We have evaluated the increase in the context of programs with no type casts and found that the increase is minimal in practice (see chapter 8). We believe this issue is unlikely to be significant in practice because we allow reuse within regions, which is quite common for data structures that shrink and grow.

4.5.3 Restrictions on Pointer Analysis

SAFECode imposes the requirement that the pointer analysis be flow-insensitive and unification-based. We believe our approach can be extended to enforce non-unification based pointer analysis as well, by adding meta-data to every pointer that may target multiple pools. However, the requirement of flow-insensitivity may be much harder to relax (though as discussed in Section 4.7, sound flow-sensitive techniques can be implemented on top of our approach.) The pointer analysis algorithm (DSA), uses an aggressive form of context-sensitivity (distinguishing heap objects by entire acyclic call paths), which compensates for many of the limitations of unification [41] and is important to distinguish distinct instances of the same kind of data structure [46]. Our approach could be extended to a non-unification-based pointer analysis, mainly by extending Automatic Pool Allocation. The primary change would be to track at run-time which pool a pointer points to at any point in the execution. This requires some metadata for any pointer that may target multiple pools. With this extension, we believe that the semantic checking techniques (the type homogeneity principle, the pool runtime checks, and the optimizations of run-time checks) would apply directly.

Extending the techniques for a flow-sensitive alias analysis is more difficult but there are two

reasons why this may not be a significant limitation in many situations. First, interprocedural pointer analysis algorithms used in practice are generally flow-insensitive because of the high cost of flow-sensitive whole program analysis [36]. Second, as discussed in Section 4.7.2, flow-sensitive techniques can be implemented (on top of flow-insensitive points-to results) in a sound manner using our approach.

4.5.4 Manufactured Pointers

If the pointer analysis cannot infer an allocation site and consequently a region, for a pointer (e.g. if the address is “manufactured” or read off the disk), we simply insert an `abort` before every use of such a pointer. This could reject a legal C program (other systems like CCured share the same weakness). A possible solution for the manufactured address case is to use pragmas or compile-time options or trusted registry mechanism that can recognize such addresses and allow them.

4.5.5 Custom Memory Allocators

Automatic Pool Allocation transformation, thus SAFECode, assume that the source program uses `malloc/free` to manage memory. When applications use custom memory allocators instead, the pool allocation transformation cannot distinguish between various memory allocations and consequently uses one single **TU** pool for the entire heap. This means that every memory access in the program is checked before use, resulting in very high overhead. One solution to this problem is to let the compiler know the different allocation and deallocation routines using compile time options.

4.5.6 Compatibility with External Libraries

So far we have assumed that we have the source for the complete program including all external libraries. In practice, we have to deal with cases where the sources for some external libraries may not be available or it may not be feasible to analyze them. Here, we explain how we handle these external library calls.

Our approach can work correctly but slightly inefficiently for most library calls. Our pointer analysis marks any points-to graph node reachable from an external function as “incomplete”. We can treat pools corresponding to such points-to graph nodes as **TU** pools. Our typing rules then

ensure that we can only load/store an `int` or `char` from/into such pools. All pointers read from such memory have to go through a run-time check, because of **R31**, thus ensuring soundness. However, this means we may conservatively perform more checks than necessary.

One case that deserves a special mention here is that of pointers to memory allocated within external library and returned to the program. The pool corresponding to such pointer may not exist (will be null in our implementation) or even if it does exist because of node merging during the pointer analysis, such a pool does not actually contain the target object of the pointer. This will lead to a run-time failure if the program ever executes a run-time check on pointers to such object. To partially solve this problem, we intercept all calls to `malloc/free` from the libraries. We store all the allocations in a global hash table and do a run-time check in the global hash table if a pool is null or if a normal run-time poolcheck fails. However, we can do this only if the memory returned by an external library call is dynamically allocated heap memory. If the library call returns stack allocated memory or memory in the static region, which we didn't encounter so far in our experiments, we abort the program.

Another problem we encounter is that of call back functions. If an internal function may be called from external code, we must ensure that the external code calls the original function, not the pool-allocated version. This ensures backwards-compatibility but at the cost of soundness. In most cases, we can directly transform the program to pass in the original function and not the pool-allocated version: this change can be made at compile-time if it passes the function name but may have to be done at run-time if it passes the function pointer in a scalar variable. In the general case (which we have not encountered so far), the function pointer may be embedded inside another data structure. Even for most such functions, the compiler can automatically generate a “`varargs`” wrapper designed to distinguish transformed internal calls from external calls. When this is not possible, we must leave the callback function and all internal calls to it, completely unmodified and use the unsafe versions.

The third problem is that of incorrect usage of library calls leading to undetected/unmasked memory errors in the unchecked external code. Though we automatically check preconditions for some of the standard C library calls before their invocation, the general solution again involves analyzing the source of the libraries.

4.6 Implementation

Our compiler system, SAFECODE (*Static Analysis For safe Execution of Code*), is implemented using the LLVM compiler infrastructure [45]. In principle, SAFECODE supports any source language translated into the LLVM IR, but our experience has been with C.

4.6.1 Type Inference and Type Checking

Conceptually, analysis validation in SAFECODE consists of a non-standard type inference step using Automatic Pool Allocation, followed by a standard type checking step using our pool-based type system defined earlier, and insertion of the necessary run-time checks described in Section 4.2.3.

The “type inference” phase of SAFECODE takes the input program and the points-to graph as defined in section 3.3 and transforms the program to add the region type attributes and region parameters of our extended type system. Because our type rules include the region types, region lifetimes, and lexical scoping of region parameters, our type checker effectively ensures the correctness of the region inference.

Our current implementation does more run-time checks than those outlined in the operational semantics; we do poolchecks before all uses of a pointer pointing to TU pool. These checks subsume checks of casts from int to pointer to TU pool (**R31**) and checks on indexing of pointers to TU pools (**R40**) but add unnecessary checks before uses of pointers to TU pools read from TK pools. We are refining our implementation to eliminate unnecessary checks.

4.6.2 The SAFECODE Runtime System

The key new aspects of the run-time (and some relevant implementation details needed to understand them) are as follows.

A pool in our implementation is organized as a linked list of large blocks. The pool handle stores the header to this list. If there is insufficient space for a new allocation, the pool requests more blocks from the underlying system heap using malloc. An allocation request is satisfied by returning a free chunk within one block or spanning multiple blocks if needed.

One key change in the pool implementation is that heap metadata such as the object header describing the size of an allocated block and the free list cannot be interleaved with live objects

in a pool since our approach allows some memory errors to overwrite arbitrary data within a pool. Allowing the metadata to be corrupted would potentially lead to arbitrary safety violations. We maintain metadata for the free list at the start of each free block and ensure (as part of the poolchecks below) that this data cannot be corrupted. To record the size of an allocated block so that it can be found efficiently, we take advantage of type homogeneity (which we have empirically found is available for most pools even in C programs) We use a bit vector (with one bit per data element of the pool type) to track the start of each allocated object (or the start of a free chunk immediately after an allocated object). Because searching this bit vector would be very inefficient for large arrays, we allocate each large array in a (contiguous) set of new blocks and perform a `poolfree` for the array simply by freeing all the blocks. The hash table used for poolchecks below allows us to identify quickly when a particular address is a large array.

By far the most important operations, in terms of performance impact, are the pool bounds checks, which are used either during the array indexing or during cast operations. Given a memory address, this check verifies that the address is contained within the memory of the pool and has the correct alignment for the pool's data type. To make the check efficient, we request memory from the system in blocks of size of 2^k bytes for some fixed k . We maintain a hash set holding the starting addresses of all current blocks. Given an address to check, we compute the block holding the address by masking the low k bits and check if the block is in the hash set. If it is, then we check for the alignment criterion. A check, therefore, involves a mask, two loads, a hash lookup, and an alignment check.

We can eliminate many hash lookups by exploiting the high spatial locality exhibited by many memory references, especially array references. We use a one-element cache to remember the block address of the last successful hash lookup and alignment check. On each check, we first compare with this cached value; if successful, we can avoid the hash lookup. For example, for an array accessed sequentially, we only need a hash look up for one in every $2^k / (\text{element-size})$ array accesses.

For uninitialized pointers, we are able to avoid a software check in many cases. Modern operating systems reserve a set of addresses for the kernel, e.g., Linux reserves the high GB of each process on a 32-bit machine. A user-level program accessing that address range would cause a hardware trap. We therefore set the *Uninit* value to the base of reserved addresses, (and replace the constant '0' in

any pointer-type expression with the same value), so that the hardware does the run-time check for us for free. This technique is unusable for kernel modules and also for references that may access a structure type with size greater than the reserved range (which is extremely rare). For these, we have to retain explicit software checks at run-time.

4.7 Sound Static Analyses Enabled By SAFECode

The semantic guarantees provided by SAFECode can be used to write other sound static analyses. In this section we first show that a static array bounds checking technique, which relies on a call graph, can be used soundly for non-type-safe programs in our environment. We then illustrate how our soundness guarantees about alias analysis can benefit other static analysis tools, using an existing software verification tool as an example.

4.7.1 Static Array Bounds Checking in SAFECode

We use an interprocedural array bounds checking algorithm, described later in chapter 9, to eliminate some runtime array bounds checks. The algorithm uses the call graph but not points-to-graph because it does not track values through loads/stores. It propagates affine constraints on integer variables from callers to callees (for incoming integer arguments and global scalars) and from callees to callers (for integer return values and global scalars). The algorithm then perform a symbolic bounds check for each index expression using integer programming. Since SAFECode semantics guarantee the correctness of the call graph, this optimization is safe (just like it would be safe for a type-safe language). To our knowledge, SAFECode is the first system for ordinary C programs (including explicit memory deallocation) where such an optimization can be performed safely.

4.7.2 Static Analyses in ESP

As final example, we briefly describe one software validation tool, ESP [18], that relies on alias analysis to give guarantees about programs and could benefit from the guarantees provided by our system. Although we explain this in terms of ESP, other software validation tools could make use of our guarantees in a similar fashion.

```

void KernelEntryPoint(int **o) {
    int **q, *r;
    char arr[15];
1: r = malloc(...);
2: ... //some computation using r
3: free(r);
    if (o != NULL)
4:     q = o;
    else {
5:     q = malloc(..);
6:     *q = ... /* *q is initialized with some safe value */
    }
7: *r = ... /* dangling pointer error, this can overwrite *q */
8: if (o != NULL)
    Probe(o); /* checks that *o is a valid pointer */
9: **q = data1; /* Dereference arbitrary pointer */
}

```

Figure 4.12: ESP Example – Value flow analysis with memory errors

ESP relies on *value flow analysis* [24], a static analysis used to identify the set of pointer expressions that refer to the memory locations that hold a certain value of interest, such as a lock. These sets are called *value alias sets* and computed by a data-flow analysis (*value flow simulation*) and transfer functions using May alias information provided by a flow-insensitive, unification-based context sensitive pointer analysis. This approach has been used to verify various properties in software, e.g., the Probe security property [24], which requires that any pointer passed into the kernel from user space is checked (“probed”) before being dereferenced by the kernel.

Consider applying ESP to verify the code fragment in Figure 4.12, which is a version of the kernel code fragment used in [24] modified to introduce a dangling pointer reference. In the function, `KernelEntryPoint`, the pointer `o` is passed in from a user routine and its target needs to be *probed* before being dereferenced by the kernel. Because of line 4, ESP tracks `q` and `o` as value aliases if `o != NULL`. The newly allocated memory when `o == NULL` is initialized to be *safe*. Lines 7 contains a memory error (a dangling pointer dereference). Since the system memory allocation could allocate previously freed memory of `r` for the allocation of `q`, this dangling pointer dereference could actually overwrite `*q`. This violates the results of the May-alias analysis that `q` and `r` are not aliased to each other. In line 8, the target of pointer `o` is probed. ESP thus transitions both the value aliases, `o` and `q`, to the *safe* state. Dereferencing `*q` is hence detected as safe by ESP. However, in reality, `*q` could now point to any location in memory and can be dereferenced by the program, violating the

Probe security property. Enforcing the assumed aliasing properties is essential for the soundness of the tool.

In our system, the same example would allocate q and r in two different pools as they are not aliased. This makes sure that dangling pointer error in r is not allowed to trample the memory of q . While we do not detect the actual error, we make sure that the aliasing property is not invalidated.

The above is an example of a flow-sensitive program analysis that uses an external flow-insensitive alias analysis and can be easily made sound using our approach. For a more general flow-sensitive analysis that reasons about loads/stores, we must modify the semantics of malloc (and free) in the analysis so that the address returned by malloc may be “aliased” to any previously freed objects in the same alias set. This is a straightforward (and local) change within the implementation of a dataflow analysis.

4.8 Experiments

Benchmark	Lines of code	Execution times (secs)					
		native	LLVM (base)	PA	PA + non array checks	SAFECode	CCured
Olden							
bh	2053	1.449	1.357	1.338	1.361	1.403	1.923
bisort	707	11.740	11.530	11.531	11.531	11.531	11.358
em3d	557	13.960	11.29	14.245	14.245	14.248	20.812
health	725	1.909	1.936	1.296	1.296	1.299	1.710
mst	617	11.259	12.920	12.837	12.837	12.96	16.956
perimeter	395	2.033	0.048	0.051	.051	0.051	2.544
power	763	1.253	0.887	0.934	0.934	0.918	1.408
treeadd	385	5.426	5.457	5.425	5.425	5.425	14.784
tsp	561	1.277	1.270	1.250	1.250	1.250	1.578
voronoi	111	Rejected because of cast from integer to pointer					
System							
fingerd	338	6.410	6.555	6.617	6.617	6.753	-
ftpd	26653	1.210	1.185	1.160	1.160	1.190	-
ghttpd	837	3.723	3.507	3.761	3.780	3.766	-
PtrDist							
anagram	647	12.778	16.084	16.915	17.953	19.742	-
ks	782	3.554	4.429	4.501	4.501	4.981	-
yacr2	3982	3.795	3.991	4.398	4.398	5.204	-

Table 4.1: Benchmarks (telnetd in text) - Execution Times

We present an experimental evaluation of SAFECode for several ordinary C programs and a few operating system daemons. These experiments have three goals:

Benchmark	Slowdown ratios			
	SAFECode /LLVM	SAFECode /PA	SAFECode /native	CCured /native
Olden				
bh	1.03	1.05	0.97	1.31
bisort	1.00	1.02	0.98	0.97
em3d	1.27	1.00	1.02	1.49
health	0.67	1.00	0.68	.90
mst	1.00	1.01	1.15	1.51
perimeter	1.04	1.00	.025	1.25
power	1.03	0.98	0.73	1.12
treadd	0.99	1.00	1.00	2.72
tsp	0.98	1.00	0.98	1.23
voronoi	Rejected because of cast from integer to pointer			
System				
fingerd	1.03	1.02	1.05	-
ftpd	1.00	1.03	0.98	-
ghttpd	1.07	0.99	1.00	-
PtrDist				
anagram	1.23	1.05	1.54	-
ks	1.12	1.11	1.40	-
yacr2	1.30	1.18	1.37	-

Table 4.2: Benchmarks - Slowdown Ratios

- To measure the net overhead and different components of overhead incurred by our run-time checks.
- To evaluate the benefit of using sound static analyses enabled by SAFECode to eliminate various kinds of runtime checks.
- To compare the overhead of our approach to that of CCured.

4.8.1 Run-time Overheads

We evaluated our system using 9 programs from the Olden suite of benchmarks [10], 3 programs from PtrDist, and 4 system codes – bsd-fingerd-0.17, ftpd-BSD-0.3.2, ghttpd-1.4, and netkit-telnet-0.17 daemon. The benchmarks and their characteristics are listed in Table 4.1. We compile each program to the LLVM compiler IR, perform our analyses and transformations, then compile LLVM back to C and compile the resulting code using GCC 3.4.2 at -O3 level of optimization. For the benchmarks we used a large problem size to obtain reliable measurements. For ftpd and fingerd, we ran the server and the client on the same machine to avoid network overhead and measured their response times for client requests. We successfully applied SAFECode to netkit-telnetd

but this is an interactive program and we did not notice any perceptible difference in the response times. We do not report detailed timings for this code here.

The “native” and “LLVM (base)” columns in the table represents execution times when compiled directly with GCC -O3 and with the base LLVM compiler using the LLVM C back-end followed by GCC -O3. Using LLVM (base) times as our baseline allows us to isolate the overheads added by SAFECODE. The “PA”, “PA + non-array checks”, and “SAFECODE” columns show the execution times with just pool allocation, SAFECODE without array indexing checks, and SAFECODE with all the run-time checks respectively.

We give the slowdown ratios in Table 4.2. The column “SAFECODE/PA” (the ratio of SAFECODE time to pool allocation time) shows that the run-time checks added by SAFECODE have a relatively small impact on performance over and above pool allocation: less than 10% in all cases except `ks` and `yacr2`, which have 11% and 18% overhead. The latter two overheads are entirely due to pool checks for array references, as seen by comparing the “PA+non-array checks” vs. the “SAFECODE” columns.

Comparing the columns “SAFECODE/LLVM” (ratio of SAFECODE time to LLVM base time) with “SAFECODE/PA,” we see that the pool allocation transformation has a significantly bigger impact on performance than the run-time checks. Four of the programs show significant slowdowns due to PA: `em3d`, `anagram`, `ks` and `yacr2`. We believe that these slowdowns are because our modified pool run-time library has not been tuned at all. We currently use an inefficient bit-vector implementation of free lists. A more recent version of the pool runtime library used in [46] shows no slowdown for these four programs. We aim to merge our extensions with this version in the near future.

We discovered that LLVM uses “loop invariant code motion” to remove an expensive computation out of the timing loop dramatically speeding up the performance of `perimeter`. This suggests that SAFECODE/LLVM ratio is perhaps the only meaningful way to isolate the overheads of SAFECODE approach. `Voronoi` benchmark fails at run-time because our pointer analysis is currently unable to track the region for a pointer that is cast from an int and it is treated as a “manufactured” pointer.

4.8.2 CCured Comparison

The last column in Table 4.2 compares the overhead of SAFECODE with that of CCured, for the Olden benchmarks rewritten by the CCured team. We have not tried to compare our results on other system codes as it involved significant porting effort in writing the CCured wrappers. In all these programs SAFECODE has significantly less overhead than CCured, even though SAFECODE's pool checks are more expensive than the run-time checks inserted by CCured. The lower overhead can be attributed to the broad range of static analysis techniques employed by SAFECODE for eliminating garbage collection (GC) overhead, stack safety checks, and many array bounds checks, and the run-time techniques that eliminate null pointer checks and metadata maintenance overhead. Note, however, that several of our static and run-time techniques for reducing overhead (except GC overhead) could be used with CCured as well. We believe that for end-users, any differences in the overheads of the systems is likely to be less important than the choice between automatic and explicit memory management and the extra wrappers that may need to be written in case of CCured.

4.8.3 Effectiveness of Static Analysis

Table 4.3 shows the effectiveness of our static checks and of segregating memory objects into TK and TU pools. Columns 2 and 3 show the total number of static array accesses and the number that must be checked at run time. The next two columns show the total number of static loads and stores and the number of pointers that need to be checked at run time. The last two columns show the static number of TU and TK pools. We found that our static array safety checks were successful in eliminating some run-time array bounds checks in most programs. Our static pointer safety techniques eliminate all other run-time checks (checks involving TU pools), except in the three programs that have TU pools.

4.9 Related Work

In this section, we compare SAFECODE with two previous approaches that provide soundness guarantees: CCured [52] and Cyclone [29]. A detailed Comparison of SAFECODE with several

Benchmark	Static Counts					
	Total array accesses	Checked array accesses	Total loads / checks	Non-array pointer checks	TU	TK
bh	80	45	708	96	1	3
bisort	2	0	103	0	0	1
em3d	17	14	80	0	0	10
health	3	0	221	0	0	2
mst	4	3	53	0	0	5
perimeter	4	4	233	0	0	1
power	4	4	229	0	0	4
treeadd	2	0	31	0	0	1
tsp	0	0	176	0	0	1
fingerd	13	8	32	11	0	3
ftpd	362	209	1949	285	2	22
telnetd	432	363	1602	0	0	15
anagram	63	47	164	4	1	5
ks	58	52	326	0	0	3
yacr2	302	302	856	0	0	26

Table 4.3: Benchmarks - Effectiveness of Static Checks

other related approaches is provided earlier in Chapter 2.4.

CCured ensures type-safe execution for standard C programs, with some source changes required for compatibility with external libraries. It uses a conservative garbage collector instead of explicit deallocation of heap memory. Compared with our approach, the major advantage of CCured is that it guarantees the absence of dangling pointer references and also performs exact bounds checks on all memory references. In contrast, a key contribution of our work has been to enable sound analysis while still retaining explicit memory management. A second difference is that CCured introduces significant metadata for runtime checks. This metadata is the primary cause of the porting effort required for using CCured on C programs because it can require wrappers around some library functions. SAFECode uses no metadata on individual pointer values and provides better backwards-compatibility than CCured.

There are also minor technical differences between the systems. Our classification of memory into type-consistent and *Unknown* is analogous to the WILD and non-WILD types of CCured, except that we use a pointer analysis to infer the types of memory objects. We allow *Unknown* memory to point to type consistent memory by performing a run-time check as explained in Section 4.1. CCured uses physical subtyping and RTTI to eliminate some run time overhead on pointer casts. Our type inference supports limited forms of physical subtyping (only for upcasts and casts from

void* to other pointer types) but we plan to investigate a more sophisticated version in the future.

Cyclone [29, 35] uses a region-based type system to enforce strict type safety and consequently enforces alias analysis, for a variant of C. Unlike SAFECode and CCured, Cyclone disallows non-type-safe memory accesses (e.g., operations that would produce the equivalent of *Unknown* type or WILD pointers). Cyclone and other region-based languages [8, 27, 9, 12, 62]) have two disadvantages relative to our work: (a) they can require significant programmer annotations to identify regions; and (b) either they provide no mechanism to free or reuse memory within a region (e.g., RTJava) or they allow deallocation of memory within a region only in special cases (e.g., uniqueness annotations to Cyclone [35] or reset region in ML kit for regions [62]). In all the above systems, data structures that must shrink and grow (with non-nested object lifetimes) can be put in regions only when they use a restricted form of aliasing. Often they have to be allocated on the garbage collected heap. In contrast, we infer the pool partitioning automatically with no annotations, and we permit explicit deallocation of individual data items within regions without aliasing restrictions or extra annotations.

4.10 Sound Analysis and Memory Safety: Concluding Discussion

In this chapter, we have described an approach to provide a semantic foundation (a points-to graph, call graph, and type information) for building sound static analyses for nearly arbitrary C programs. The approach can be easily added to any C compiler containing a pointer analysis that meets the specified properties (flow-insensitive, unification-based).

The approach also has some other practical strengths: it is fully automatic and requires no modifications to existing C programs; it allocates and frees memory objects at the same points as the original program (minimizing the need to tune memory consumption); and it supports nearly the full generality of the C language, except for manufactured addresses and some casts from int to pointers. Finally, our experiments show that the run-time overheads of our approach are quite small, generally less than a few percent relative to code with pool allocation alone. We believe these overheads are low enough to be used in production code, especially when security is a significant concern.

We believe that our approach represents an interesting and useful low overhead alternative to

techniques that focus on complete soundness with no dangling pointer errors. However, in some application domains stronger guarantees than given by our current approach are required. We believe that the right long term approach is to offer a choice to the end user between our current approach with alternatives that can detect all memory errors or use garbage collection. Towards this end, we built a framework that includes this work along with our other work that detects bounds errors (see chapter 6) and dangling pointer errors (see chapter 7).

Chapter 5

Formal Proof of Soundness

This chapter gives provides a proof of soundness of our approach for providing memory safety and sound analysis guarantees. First, section 5.1 discusses the invariants that we maintain in our approach. Section 5.2 then states the soundness theorem and gives its proof.

5.1 Invariants for Well Formed Environments

In the rest of this discussion, by environment we mean the pair $(VEnv, L)$ where $VEnv$ is the variable environment and L is the live region map in the heap. For an environment $(VEnv, L)$, we define $\|\tau\|_{(VEnv, L)}$ to be as follows:

$$\begin{aligned} \|\mathbf{int}\|_{(VEnv, L)} &:= \text{Int}_{32} \\ \|\tau * \rho\|_{(VEnv, L)} &:= \{Uninit\} \cup \text{Dom}(L[\rho].RS) \\ \|\text{handle}(\rho, \tau)\|_{(VEnv, L)} &:= \{\text{region}(\rho)\} \\ \|\mathbf{Unknown}\|_{(VEnv, L)} &:= \text{Int}_8 \\ \|\mathbf{char}\|_{(VEnv, L)} &:= \text{Int}_8 \end{aligned}$$

From our earlier assumptions, $\text{sizeof}(\mathbf{int})$ and $\text{sizeof}(\tau * \rho)$ is same: four bytes. This means that $\text{Dom}(L[\rho].RS) \subseteq \text{Int}_{32}$.

Intuitively for a well-formed type τ , $\|\tau\|_{(VEnv, L)}$ represents the set of values that a variable (or object) of that type can hold under that context and environment. We assume that $Uninit \in \text{Int}$. We treat it as zero in our operations. For a pointer variable (or a memory location of pointer type), the values it can hold depend on the already allocated values in the region to which the pointer points to. For a pointer to region ρ only addresses in region ρ (or the uninitialized value) are legal values.

The judgment \vdash_{env} stands for a well formed environment. An environment $(VEnv, L)$ is well formed under a typing context C (denoted by $C(= \Gamma; \Delta) \vdash_{env} (VEnv, L)$) if and only if the following invariants hold.

Inv1 $Dom(\Gamma) = Dom(VEnv)$

All variables in the typing environment are present in the variable environments and vice versa.

Inv2 $Dom(\Delta) = Dom(L)$

All region names in the region type environment are already present in the domain of region maps and vice versa.

Inv3 $\forall x \in Dom(VEnv)$, if $C \vdash x : \tau$ then $VEnv[x] \in \|\tau\|_{(VEnv, L)}$

If a variable has type τ , then it must contain only valid values of type τ . In particular, a pointer variable with region attribute ρ , must always point to an object in that region or it is not initialized.

Inv4 $\forall \rho \in Dom(L)$, if $C \vdash \rho : \tau$ then $\forall v \in Dom(L[\rho].RS)$, $L[\rho].RS[v] \in \|\tau\|_{(VEnv, L)}$.

If region ρ is associated with type τ then each memory location in the region store will only contain values of the correct type.

Inv5 $\forall \rho \in Dom(L)$, $L[\rho].F \subseteq Dom(L[\rho].RS)$

This invariant states that the memory addresses in the free list are a subset of the addresses of the region

Inv6 $\forall \rho_1 \rho_2 \in Dom(L)$, if $\rho_1 \neq \rho_2$ then $Dom((L[\rho_1]).RS) \cap Dom((L[\rho_2]).RS) = \phi$ and $\forall \rho \in Dom(L)$, $Dom(L[\rho].RS) \cap H = \phi$.

A memory address cannot be part of two live regions. Also a memory address cannot be a part of system heap (i.e., unused by a program) and also a part of live region.

5.2 Proof

The proof of soundness is composed of two “invariant preservation” theorems — one for expressions and one for statements of the program. Since we have not included control flow in our formalization, all evaluations of expressions and statements terminate.

Notation: In the rest of this section, \longrightarrow_{expr}^* represents the usual reflexive transitive closure of \longrightarrow_{expr} and \longrightarrow_{stmt}^* represents the usual reflexive transitive closure of \longrightarrow_{stmt} .

In order to prove the “invariant preservation” theorems, we make use of the lemmas listed in Figures 5.1 and 5.2, which are essentially big-step extensions of some of the small step rules given in Figures 4.7, 4.8, and 4.9. The proof of each of the lemmas is by straightforward induction on the number of steps in the derivation of the hypothesis in that lemma. In the Figure 5.1, we give the complete proof for the first lemma, proofs for the rest are similar and straightforward.

We now present three more lemmas, that are useful in proving the invariant preservation theorems.

Lemma 1 Update Lemma1

If $C \vdash_{env} (VEnv, L)$, and $C \vdash \rho : \tau$ and if $v_1 \in Dom(L[\rho].RS)$, and $v_2 \in \|\tau\|_{(VEnv,L)}$ then $C \vdash_{env} (VEnv, update(L, v_1, v_2))$.

This lemma states that given a well formed environment, if we update a memory location in a region of the environment with the appropriate type then the resulting environment continues to be well formed.

Proof: From **Inv2** and $C \vdash \rho : \tau$, we have $\rho \in Dom(L)$.

So $L = L' \cup \{(\rho, R)\}$

We also have $v_1 \in Dom(R.RS)$.

From the definition of update, we have $update(L, v_1, v_2) = L' \cup \{(\rho, \{ R.F ; R.(RS[v_1 \mapsto v_2]) \})\}$

Now all the invariants except **Inv4** trivially hold. **Inv4** holds since $v_2 \in \|\tau\|_{(VEnv,L)}$. q.e.d.

Lemma 2 Update Lemma2

If $C \vdash_{env} (VEnv, L)$, and $C \vdash \rho : Unknown$ and if $[v_1, v_1 + 3] \in Dom(L[\rho].RS)$, and $v_2 \in \|\text{Int}\|_{(VEnv,L)}$ then $C \vdash_{env} (VEnv, update(L, v_1, v_2, 4))$.

- Lemma R2*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, x = E) \longrightarrow_{stmt}^* (VEnv', L', x = E')$.
Proof: By induction on the number of steps in the derivation of $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$.
Base case: zero steps. Trivially true.
Induction Hypothesis : True for 'k' steps in derivation.
For 'k+1' steps, we have
 $(VEnv, L, E) \longrightarrow_{expr} \dots \longrightarrow_{expr} (k \text{ steps}) (VEnv'', L'', E'') \longrightarrow_{expr} (VEnv', L', E')$.
Using induction hypothesis, we have $(VEnv, L, x = E) \longrightarrow_{stmt}^* (VEnv'', L'', x = E'')$.
We also have $(VEnv'', L'', E'') \longrightarrow_{expr} (VEnv', L', E')$
Using R2, we have $(VEnv'', L'', X = E'') \longrightarrow_{stmt} (VEnv', L', x = E')$.
q.e.d.
- Lemma R1*** If $(VEnv, L, S1) \longrightarrow_{stmt}^* (VEnv', L', S1')$ then $(VEnv, L, S1 ; S2) \longrightarrow_{stmt}^* (VEnv', L', S1' ; S2)$
- Lemma R4*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{store}/\text{storec } E, E_2) \longrightarrow_{stmt}^* (VEnv', L', \text{store } E', E_2)$.
- Lemma R5*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{store}/\text{storec } v, E) \longrightarrow_{stmt}^* (VEnv', L', \text{store } v, E')$.
If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{storeU}/\text{storecU } E, E_2, E_3) \longrightarrow_{stmt}^* (VEnv', L', \text{storeU}/\text{storecU } E', E_2, E_3)$.
- Lemma R8*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{storeU}/\text{storecU } v_1, E, E_3) \longrightarrow_{stmt}^* (VEnv', L', \text{storeU}/\text{storecU } v_1, E', E_3)$.
- Lemma R9*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{storeU}/\text{storecU } v_1, v_2, E) \longrightarrow_{stmt}^* (VEnv', L', \text{storeU}/\text{storecU } v_1, v_2, E')$.
- Lemma R12*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{poolfree}(E, E_2) \longrightarrow_{stmt}^* (VEnv', L', \text{poolfree}(E', E_2)$.
- Lemma R13*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{poolfree}(v, E) \longrightarrow_{stmt}^* (VEnv', L', \text{poolfree}(v, E')$.
- Lemma R16*** If $(VEnv, L, S) \longrightarrow_{stmt}^* (VEnv', L', S')$ then $(VEnv, L, \text{pool}\{S\}\text{pop}(\rho) \longrightarrow_{stmt}^* (VEnv', L', \text{pool}\{S'\}\text{pop}(\rho))$.
- Lemma R19*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, E \text{ op } E_2) \longrightarrow_{expr}^* (VEnv', L', E' \text{ op } E_2)$.
- Lemma R20*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, v \text{ op } E) \longrightarrow_{expr}^* (VEnv', L', v \text{ op } E')$.
- Lemma R22*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{load}/\text{loadc } E) \longrightarrow_{expr}^* (VEnv', L', \text{load } E')$.
- Lemma R24*** If $(VEnv, L, E) \longrightarrow_{expr}^* (VEnv', L', E')$ then $(VEnv, L, \text{loadU}/\text{loadcU } E, E_2) \longrightarrow_{expr}^* (VEnv', L', \text{loadU}/\text{loadcU } E', E_2)$.

Figure 5.1: Lemmas for operational semantics

- Lemma R25*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \text{loadU/loadcU } v_1, E) \xrightarrow{*_{expr}} (VEnv', L', \text{loadU/loadcU } v_1, E')$.
- Lemma R29*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \text{castintpointer } E, E_2 \text{ to } \tau) \xrightarrow{*_{expr}} (VEnv', L', \text{castintpointer } E', E_2 \text{ to } \tau)$.
- Lemma R30*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \text{castintpointer } v, E \text{ to } \tau) \xrightarrow{*_{expr}} (VEnv', L', \text{castintpointer } v, E' \text{ to } \tau)$.
- Lemma R32*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \text{poolalloc}(E, E_2)) \xrightarrow{*_{expr}} (VEnv', L', \text{poolalloc}(E', E_2))$.
- Lemma R33*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \text{poolalloc}(v, E)) \xrightarrow{*_{expr}} (VEnv', L', \text{poolalloc}(v, E'))$.
- Lemma R37*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \&(E, E_1)[E_2]) \xrightarrow{*_{expr}} (VEnv', L', \&(E', E_1)[E_2])$.
- Lemma R38*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, \&(v, E)[E_2]) \xrightarrow{*_{expr}} (VEnv', L', \&(v, E')[E_2])$.
- Lemma R39*** If $(VEnv, L, E) \xrightarrow{*_{expr}} (VEnv', L', E')$ then $(VEnv, L, v, \&(v_1)[E]) \xrightarrow{*_{expr}} (VEnv', L', \&(v, v_1)[E'])$.

Figure 5.2: Lemmas for operational semantics

Proof: The proof is straightforward extension of above; we just need to prove that byte function on Int_{32} gives an integer in Int_8 , which is true from the arithmetic properties of integers.

Lemma 3 Getvalue Lemma1

If $C \vdash_{env} (VEnv, L)$, $C \vdash \rho : \tau$ and if $v_1 \in \text{Dom}(L[\rho].RS)$, then $\text{getvalue}(L, v_1) \in \|\tau\|_{(VEnv, L)}$.

Informally, this lemma states that given a well formed environment, if we load from a memory address in a region, the resulting value should have the type of the objects stored in that region.

Proof: We have ρ in $\text{Dom}(L)$ from **Inv2**.

$$\text{So } L \equiv L' \cup \{(\rho, R)\}$$

From the definition of getvalue , we have $\text{getvalue}(L, v_1) = L[\rho].RS[v_1]$. Now from **Inv4** we have $\text{getvalue}(L, v_1) \in \|\tau\|_{(VEnv, L)}$. q.e.d.

Lemma 4 Getvalue Lemma2

If $C \vdash_{env} (VEnv, L)$, $C \vdash \rho : \text{Unknown}$ and if $v_1 \in \text{Dom}(L[\rho].RS)$, then $\text{getvalue}(L, v_1, 4) \in \|\text{Int}\|_{(VEnv, L)}$.

Proof: Proof is straightforward extension of above; we just need to prove that the combine function on Int_8 gives an Int_{32} , which is true from the arithmetic properties of integers.

Lemma 5 (Safe region deallocate Lemma)

If $(\Gamma; \Delta) \vdash \tau$, and $x \notin \text{Dom}(\Gamma)$, and
 $\rho \notin \text{Dom}(\Delta)$, and $(\Gamma[x \mapsto \text{handle}(\rho, \tau)]; \Delta[\rho \mapsto \tau']) \vdash_{env} (VEnv \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, R)\})$
then

$$\|\tau\|_{(VEnv \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, R)\})} = \|\tau\|_{(VEnv, L)}.$$

This lemma states that the set of legal values corresponding to a “well formed type” τ , is independent of a region on which this type is not dependent. Hence that region can be safely deallocated if necessary.

Proof: If τ is of the form `int` or `Unknown` then it is trivially true.

If τ is of the form `handle`(ρ'' , τ'') then $C \vdash \tau$ only if $C \vdash \rho'' : \tau''$ (from SS23) and in turn $\rho'' \in \text{Dom}(\Delta)$ from SS20. Since $\rho \notin \text{Dom}(\Delta)$, $\rho'' \neq \rho$. Therefore, $\|\tau\|_{(VEnv \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, \tau')\})} = \text{region}(\rho'') = \|\tau\|_{(VEnv, L)}$.

The most important case is when τ is a pointer type, i.e. τ is of the form $\tau'' * \rho''$. In this case, using SS20 and (SS22 or SS21), we get $\rho'' \neq \rho$ and hence $\|\tau\|_{(VEnv \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, \tau')\})} = \{\text{Uninit}\} \cup \text{Dom}(L[\rho''].RS) = \|\tau\|_{(VEnv, L)}$.

We now state and prove the invariant preservation theorem for expressions.

Theorem 1 *If $C \vdash e : \tau$ and $C \vdash_{env} (VEnv, L)$ then either $(VEnv, L, e) \xrightarrow{*_{expr}} \text{Error}$ or $(VEnv, L, e) \xrightarrow{*_{expr}} (VEnv', L', v)$ such that $v \in \|\tau\|_{(VEnv', L')}$ and $C \vdash_{env} (VEnv', L')$*

This theorem states that given a typing context C , if e is a well typed expression typing to τ then evaluation of e in a well formed environment either fails because of a run-time check failure or gives a value of the appropriate type along with another well formed environment.

Proof: The proof of this theorem is by induction on the structure of typing derivation of $C \vdash e : \tau$.

Based on the last rule used in the typing derivation of e , we have the following (exhaustive) list of cases:

- SS0
 e must be of the form x and $x \in \text{Dom}(\Gamma)$.

Since $C \vdash_{env} (\text{VEnv}, L)$, from **Inv1** we get $x \in \text{Dom}(\text{VEnv})$.

Now consider (VEnv, L, x) with $x \in \text{Dom}(\text{VEnv})$. Rule **R18** applies. Hence, $(\text{VEnv}, L, x) \longrightarrow_{expr} (\text{VEnv}, L, v)$ where v is the image of x in VEnv . q.e.d.

- SS1

e must be of the form n .

Trivially, $(\text{VEnv}, L, e) \longrightarrow_{expr}^* (\text{VEnv}, L, n)$. q.e.d.

- SS2

e must be of the form $e_1 \text{ op } e_2$ with $C \vdash e_1 : \text{int}$ and $C \vdash e_2 : \text{int}$.

Using induction hypothesis, we have $(\text{VEnv}, L, e_1) \longrightarrow_{expr}^* \text{Error}$ or $(\text{VEnv}'', L'', v_1)$ with $v_1 \in \|\text{int}\|_{(\text{VEnv}'', L'')}$ and $C \vdash_{env} (\text{VEnv}'', L'')$.

If $(\text{VEnv}, L, e_1) \longrightarrow_{expr}^* \text{Error}$ then q.e.d.

If not, using induction hypothesis again, we have $(\text{VEnv}'', L'', e_2) \longrightarrow_{expr} \text{Error}$ or (VEnv', L', v_2) with $v_2 \in \|\text{int}\|_{(\text{VEnv}', L')}$ and $C \vdash_{env} (\text{VEnv}', L')$.

If $(\text{VEnv}, L, e_2) \longrightarrow_{expr}^* \text{Error}$ then q.e.d.

If not,

- Using lemma **R19***, we have $(\text{VEnv}, L, e_1 \text{ op } e_2) \longrightarrow_{expr}^* (\text{VEnv}'', L'', v_1 \text{ op } e_2)$
- Using lemma **R20***, we have $(\text{VEnv}'', L'', v_1 \text{ op } e_2) \longrightarrow_{expr}^* (\text{VEnv}', L', v_1 \text{ op } v_2)$
- Since $v_1, v_2 \in \text{Int}$, using **R21** we have $(\text{VEnv}', L', v_1 \text{ op}_{Int} v_2)$. with $v_1 \text{ op}_{Int} v_2 \in \text{Int}$ and $(C \vdash_{env} (\text{VEnv}', L'))$. q.e.d.

- SS3

e must be of the form $Uninit$.

Trivially $(\text{VEnv}, L, Uninit) \longrightarrow_{expr}^* (\text{VEnv}, L, Uninit)$ and since $Uninit \in \|\tau\|_{(\text{VEnv}, L)}$ where $\tau \neq \text{handle}(\rho', \tau')$, q.e.d.

- SS4

e must be of the form $\text{load } e'$ and there exists ρ such that $C \vdash \rho : \tau$ and $C \vdash e' : \tau * \rho$ and $\tau \notin \{\text{Unknown}, \text{char}\}$.

Now using induction hypothesis we have either $(VEnv, L, e') \longrightarrow_{expr}^* \text{Error}$ or $(VEnv'', L'', v')$ such that $v' \in (\{Uninit\} \cup \text{Dom}(L''[\rho].RS))$ and $(VEnv'', L'')$ is well formed environment.

If $v' = Uninit$ then by **R23**, $(VEnv'', L'', \text{load } v') \longrightarrow_{expr} \text{Error}$ and q.e.d.

If $v' \in \text{Dom}(L''[\rho].RS)$ then from the load rule **R23**, we get $(VEnv'', L'', \text{load } v') \longrightarrow_{expr} (VEnv'', L'', \text{getvalue}(L'', v'))$.

From the “**getvalue**” lemma we get $\text{getvalue}(L'', v') \in \|\tau\|_{(VEnv'', L'')}$. Now using **R22*** we have the result. q.e.d.

Informally, the proof step says that in case of load from type consistent memory, the address points to a correct object in the region or its uninitialized value, hence it progresses to an error or gives a value of the correct type.

- SS4char

Similar to above.

- SS5

e must be of the form $\text{loadU } x, e_2$, τ must be **int** with $C \vdash e_2 : \tau' * \rho$ and $C \vdash \tau : \text{Unknown}$ and $C \vdash x : \text{handle}(\rho, \text{Unknown})$.

Using induction hypothesis, we have either $(VEnv, L, x) \longrightarrow_{expr}^* \text{Error}$ or $(VEnv'', L'', v_1)$ with $v_1 = \text{region}(\rho)$ and $C \vdash_{env} (VEnv'', L'')$.

If not Error, using lemma **R24***, we get $(VEnv, L, \text{loadU } e_1, e_2) \longrightarrow_{expr}^* (VEnv'', L'', \text{loadU } \text{region}(\rho), e_2)$.

Again from induction hypothesis on e_2 we have $(VEnv'', L'', e_2) \longrightarrow_{expr}^* \text{Error}$ or $(VEnv', L', v_2)$ with $v_2 \in \{Uninit\} \cup \text{Dom}(L[\rho].RS)$ and $C \vdash_{env} (VEnv', L')$.

If $(VEnv'', L'', e_2) \longrightarrow_{expr}^* \text{Error}$ then q.e.d.

If not, using lemma **R25***, we get $(VEnv'', L'', \text{load } \text{region}(\rho), e_2) \longrightarrow_{expr}^* (VEnv', L', \text{load } \text{region}(\rho), v_2)$ and $v_2 \in \{Uninit\} \cup \text{Dom}(L[\rho].RS)$. Now R26 applies and since we check that $v_2 + 3 \in \text{Dom}(L[\rho].RS)$, we get Error or $(VEnv', L', \text{getvalue}(L, v_1, 4))$.

From “**Getvalue**” lemma2, getvalue just retrieves the value from the location, and doesn't change any of the invariants of $(VEnv', L')$. q.e.d.

- SS5char

Similar to SS4char.

- SS6

e has to be of form $\text{poolalloc}(x, e2)$ and τ of the form $\tau' * \rho$ and $C \vdash \rho : \tau'$ and $C \vdash x : \text{handle}(\rho, \tau')$ and $C \vdash e2 : \text{int}$

Using induction hypothesis, $(\text{VEnv}, L, x) \longrightarrow_{\text{expr}} \text{Error}$ or (VEnv'', L'', v) s.t. $v = \text{region}(\rho)$. If not error, $(\text{VEnv}'', L'', e2) \longrightarrow_{\text{expr}} (\text{VEnv}''', L''', n)$. If n is one then either rule **R34** or rule **R35** applies.

If rule **R34** applies, then the value returned is a value from the free list, and from invariant 4 for well-formed environments, we get that the value returned is of the correct type. Since we just removed an element from the freelist, the **Inv5** still holds and we continue to have a well formed environment.

If **R35** applies, then we add an element to the set of addresses of this region and since the new address is from system heap H , **Inv6** continues to hold. **Inv4** holds in the new environment since we have initialized it with $Uninit$ value. **Inv5** holds since the free list has not changed. Hence the well formedness of the environment remains intact.

The case where n is not one is similar by using the array allocation rule. q.e.d.

- SS7

e has to be of the form $\text{castintpointer } x, e''$ to $\tau' * \rho$ with τ of the form $\tau' * \rho$ with $C \vdash \rho : \tau'$, $C \vdash x : \text{handle}(\rho, \tau')$ and $C \vdash e'' : \text{int}$

Using induction hypothesis, $(\text{VEnv}, L, x) \longrightarrow_{\text{expr}}^* \text{Error}$ or (VEnv'', L'', v') such that $v' = \text{region}(\rho)$.

If not error, using lemma **R30*** we have $(\text{VEnv}, L, \text{castintpointer } e', e'' \text{ to } \tau' * \rho) \longrightarrow_{\text{expr}}^* (\text{VEnv}'', L'', \text{castintpointer } \text{region}(\rho), e'' \text{ to } \tau' * \rho)$.

Using induction hypothesis again $(\text{VEnv}'', L'', e'') \longrightarrow_{\text{expr}}^* \text{Error}$ or (VEnv', L', v'') such that $v'' \in \text{Int}$. If not error, using lemma **R31*** we have $(\text{VEnv}'', L'', \text{castintpointer } \text{region}(\rho), e'')$

to $\tau' * \rho$) \longrightarrow_{expr}^* (VEnv'', L'', castintpointer region(ρ), v_2 to $\tau' * \rho$) with $v_2 \in \text{Int}$.

From rule **R31** from the operational semantics, we get (VEnv'', L'', castintpointer region(ρ), v_2 to $\tau' * \rho$) \longrightarrow_{expr} Error or (VEnv'', L'', v_2) s.t. $v_2 \in \text{Dom}(L''[\rho].\text{RS})$ or in other words $v_2 \in \|\tau' * \rho\|_{(VEnv'', L'')}$. q.e.d.

- SS8

This is straightforward application of the induction hypothesis and rule **R28**. None of the invariants change since the invariants for pointer types depend on the region attribute and not the actual type (see the discussion of SS8 in section 4.2.2 to understand how we can support casts between arbitrary pointer types).

- SS9

All array accesses are checked for the pool boundaries and alignment. e is of the form $x, \&e2[e3]$ with $C \vdash \rho : \tau$, $C \vdash x : \text{handle}(\rho, \tau)$, $C \vdash e2 : \tau * \rho$, $C \vdash e3 : \text{int}$.

Using induction hypothesis we get, (VEnv, L, x) \longrightarrow_{expr}^* Error or (VEnv'', L'', region(ρ)) with $C \vdash_{env}$ (VEnv'', L'').

If not error, then using lemma **R37*** we get (VEnv, L, $x, \&e2[e3]$) \longrightarrow_{expr}^* (VEnv'', L'', region(ρ), $\&e2[e3]$).

Using induction hypothesis again we get, (VEnv'', L'', $e2$) \longrightarrow_{expr}^* Error or (VEnv''', L''', v_2) with $v_2 \in \|\tau * \rho\|_{(VEnv''', L''')}$ and $C \vdash_{env}$ (VEnv''', L''').

If not error, then using lemma **R38*** we get (VEnv'', L'', region(ρ), $\&e2[e3]$) \longrightarrow_{expr}^* (VEnv''', L''', region(ρ), $\&v_2[e3]$).

Using induction hypothesis again we get, (VEnv''', L''', $e3$) \longrightarrow_{expr}^* Error or (VEnv', L', n) with $C \vdash_{env}$ (VEnv', L').

If not error, then using lemma **R39*** we get (VEnv''', L''', region(ρ), $\&v_2[e3]$) \longrightarrow_{expr}^* (VEnv', L', region(ρ), $\&v_2[m]$).

Now **R40** applies. If $(v_2 + m * \text{sizeof}(\tau)) \notin \text{Dom}(L'[\rho])$ then Error else (VEnv', L', $v_2 + m * \text{sizeof}(\tau)$) with $(v_2 + m * \text{sizeof}(\tau)) \in \|\tau * \rho\|_{(VEnv', L')}$ and $C \vdash_{env}$ (VEnv', L'). q.e.d.

- SS10

This is straightforward application of the induction hypothesis and rule **R28**.

Theorem 2 *If $C \vdash S$ and $C \vdash_{env} (VEnv, L)$ then either $(VEnv, L, S) \longrightarrow_{stmt}^* Error$ or $(VEnv, L, S) \longrightarrow_{stmt}^* (VEnv', L', \epsilon)$ and $C \vdash_{env} (VEnv', L')$.*

Here \longrightarrow_{stmt}^* represents reflexive transitive closure of \longrightarrow_{stmt} . This theorem states that given a typing environment C , if statement S is well typed in that typing environment then evaluation of S either fails because of a run-time check failure or terminates along with another well formed environment. To put it differently, it does not get stuck because of type violation (e.g. trying to access non-existent memory location).

Proof: The proof of this theorem is by induction on the structure of typing derivation of $C \vdash S$.

Based on the last rule used in the typing derivation of S , we have the following (exhaustive) list of cases:

- SS11

S is of the form ϵ .

Trivial case.

- SS12

S is of the form $S1; S2$ with $C \vdash S1$ and $C \vdash S2$.

Using induction hypothesis, $(VEnv, L, S1) \longrightarrow_{stmt}^* Error$ or $(VEnv'', L'', \epsilon)$.

If not Error, Using R1* $(VEnv, L, S1 ; S2) \longrightarrow_{stmt}^* (VEnv'', L'', S2)$

Using induction hypothesis again, we have $(VEnv'', L'', S2) \longrightarrow_{stmt}^* Error$ or $(VEnv'', L'', \epsilon)$.

Using transitivity of \longrightarrow_{stmt}^* q.e.d.

- SS13

S is of the form $x = e$ with $C \vdash x : \tau$ and $C \vdash e : \tau$.

Using Theorem 1 on $C \vdash e : \tau$, we have $(VEnv, L, e) \longrightarrow_{expr}^* Error$ or $(VEnv'', L'', v)$ with $v \in \llbracket \tau * \rho \rrbracket_{(VEnv'', L'')}$ and $C \vdash_{env} (VEnv'', L'')$.

Using lemma **R2***, we get $(\text{VEnv}, L, x = e) \longrightarrow_{\text{stmt}}^* (\text{VEnv}'', L'', x = v)$. with $v \in \|\tau\|_{(\text{VEnv}'', L'')}$.

Using R3 we get $(\text{VEnv}'', L'', x = v) \longrightarrow_{\text{stmt}}^* (\text{VEnv}''[x \mapsto v], L'', \epsilon)$.

Let $\text{VEnv}' = \text{VEnv}''[x \mapsto v]$. Now we need to prove that $C \vdash_{\text{env}} (\text{VEnv}', L'')$. **Inv1, Inv2, Inv4, Inv5, Inv6** can be trivially proved from $C \vdash_{\text{env}} (\text{VEnv}'', L'')$.

We have $C \vdash x : \tau$ and $v \in \|\tau\|_{(\text{VEnv}'', L'')}$. So $v \in \|\tau\|_{(\text{VEnv}', L')}$ since VEnv' differs from VEnv'' only in the mapping of x . We also have $\text{VEnv}'[x] = v$. So **Inv3** continues to hold. Hence $C \vdash_{\text{env}} (\text{VEnv}', L'')$.

- SS14

S is of the form $C \vdash \text{store } e_2, e_1$, with $C \vdash \rho : \tau$, $C \vdash e_1 : \tau * \rho$, and $C \vdash e_2 : \tau$.

Using Theorem 1, we have $(\text{VEnv}, L, e_2) \longrightarrow_{\text{expr}}^* \text{Error}$ or $(\text{VEnv}'', L'', v_2)$ with $v_2 \in \|\tau\|_{(\text{VEnv}'', L'')}$ and $C \vdash_{\text{env}} (\text{VEnv}'', L'')$.

If not error, Using **R4***, we have $(\text{VEnv}, L, \text{store } e_2, e_1) \longrightarrow_{\text{stmt}}^* (\text{VEnv}'', L'', \text{store } v_2, e_1)$.

First from **Inv2** we have $\rho \in \text{Dom}(L)$.

Now using theorem 1, we have $(\text{VEnv}'', L'', e_1) \longrightarrow_{\text{expr}}^* \text{Error}$ or (VEnv', L', v_1) with $v_1 \in \|\tau * \rho\|_{(\text{VEnv}', L')}$ and $C \vdash_{\text{env}} (\text{VEnv}', L')$.

If not error, Using **R5***, we have $(\text{VEnv}'', L'', \text{store } v_2, e_1) \longrightarrow_{\text{stmt}}^* (\text{VEnv}', L', \text{store } v_2, v_1)$.

If v_1 is *Uninit* then using **R6** we get $(\text{VEnv}', L', \text{store } v_2, v_1) \longrightarrow_{\text{stmt}} \text{Error}$ and q.e.d.

If not, $(\text{VEnv}', L', \text{store } v_2, v_1) \longrightarrow_{\text{stmt}} (\text{VEnv}', \text{update}(L', v_1, v_2), \epsilon)$ from **R6**.

Using the **Update lemma1**, $C \vdash_{\text{env}} (\text{VEnv}', \text{update}(L', v_1, v_2))$.

Similarly we can prove for the SS14char case.

- SS15 and SS15char

same as above

- SS16

S is of the form $\text{poolfree}(x, e_2)$ and $C \vdash \rho : \tau$ and $C \vdash x : \text{handle}(\rho, \tau)$ and $C \vdash e_2 : \tau * \rho$.

Using **Theorem1**, we get $(\text{VEnv}, L, x) (\text{VEnv}'', L'', v_1)$ s.t. $v_1 \in \|\text{handle}(\rho, \tau)\|_{(\text{VEnv}'', L'')}$ and $C \vdash_{env} (\text{VEnv}'', L'')$.

Now using **Theorem 1** again, we get $(\text{VEnv}'', L'', e_2) \xrightarrow{*}_{expr} \text{Error}$ or (VEnv', L', v_2) with $v_2 \in \|\tau * \rho\|_{(\text{VEnv}', L')}$ and $C \vdash_{env} (\text{VEnv}', L')$.

Using **R12*** and **R13***, we get $(\text{VEnv}, L, \text{poolfree}(x, e_2)) \xrightarrow{*}_{stmt} (\text{VEnv}', L', \text{poolfree}(v_1, v_2))$.

From the definition of $\|\tau\|$, $v_1 = \text{region}(\rho)$ and

$v_2 \in \{\text{Uninit}\} \cup \text{Dom}(L'[\rho].\text{RS})$.

If $(v_2 == \text{Uninit})$ then from **R14**, $(\text{VEnv}', L', \text{poolfree}(v_1, v_2)) \xrightarrow{*}_{stmt} \text{Error}$.

If not, from **Inv2** $\rho \in \text{Dom}(L')$ and **R14** applies. Let $L' = L'' \cup \{(\rho, \{F; RS\})\}$, $\rho \notin \text{Dom}(L'')$.

So $(\text{VEnv}', L'' \cup \{(\rho, \{F; RS\})\}, \text{poolfree}(\text{region}(\rho), v_2)) \xrightarrow{stmt} (\text{VEnv}', L'' \cup \{(\rho, \{v_2F; RS\})\}, \epsilon)$.

We just need to prove that $C \vdash_{env} (\text{VEnv}', L'' \cup \{(\rho, \{v_2F; RS\})\})$.

We already have $C \vdash_{env} (\text{VEnv}', L')$. So for $(\text{VEnv}', L'' \cup \{(\rho, \{v_2F; RS\})\})$ **Inv1**, **Inv2**, **Inv3** trivially hold as they are the same for (VEnv', L') . **Inv4** holds since $\forall \rho L'[\rho].\text{RS}$ is unmodified. **Inv5** holds since $v_2 \in \text{Dom}(L'[\rho].\text{RS})$. **Inv6** holds since $\forall \rho L'[\rho].\text{RS}$ and H is unmodified. Hence $C \vdash_{env} (\text{VEnv}', L'' \cup \{(\rho, \{v_2F; RS\})\})$. q.e.d.

- SS17

S is of the form $\text{poolinit}(\rho, \tau) \times \{ S' \}$ with $C(= \Gamma, \Delta) \vdash \tau$ and $\Gamma[x \mapsto \text{handle}(\rho, \tau)], \Delta[\rho \mapsto \tau] \vdash S'$ with $x \notin \text{Dom}(\Gamma)$ and $\rho \notin \text{Dom}(\Delta)$.

Rule **R15** applies if $\rho \notin \text{Dom}(L)$. We already have $\rho \notin \text{Dom}(\Delta)$ and from **Inv2** we have $\rho \notin \text{Dom}(L)$. So **R15** applies. Hence, $(\text{VEnv}, L, \text{poolinit}(\rho, \tau) \times \{ S' \}) \xrightarrow{stmt} (\text{VEnv} \cup \{(x, \text{region}(\rho))\}, L \cup \{(\rho, \{\phi; \phi\})\}, \text{pool} \{ S' \} \text{pop}(\rho))$.

Let $\text{VEnv}'' = \text{VEnv} \cup \{(x, \text{region}(\rho))\}$ and $L'' = L \cup \{(\rho, \{\phi; \phi\})\}$.

Let $C'(\Gamma', \Delta') = \Gamma[x \mapsto \text{handle}(\rho, \tau)], \Delta[\rho \mapsto \tau]$. We first need to prove that $C' \vdash_{env} (\text{VEnv}'', L'')$.

- **Inv1**

From **Inv1** of (VEnv, L) , we get $\text{Dom}(\Gamma) = \text{Dom}(\text{VEnv})$.

$$\begin{aligned}
\text{Dom}(\Gamma') &= \text{Dom}(\Gamma) \cup \{ x \} \\
&= \text{Dom}(\text{VEnv}) \cup \{ x \} \\
&= \text{Dom}(\text{VEnv}''')
\end{aligned}$$

So **Inv1** holds.

– **Inv2**

From **Inv2** of (VEnv, L) , we get $\text{Dom}(\Delta) = \text{Dom}(L)$.

$$\begin{aligned}
\text{Dom}(\Delta') &= \text{Dom}(\Delta) \cup \{ \rho \} \\
&= \text{Dom}(L) \cup \{ \rho \} \\
&= \text{Dom}(L''')
\end{aligned}$$

So **Inv2** holds.

– **Inv3**

From **Inv3** of (VEnv, L) we get $\forall y \in \text{Dom}(\text{VEnv})$, if $C \vdash y : \tau'$ then $\text{VEnv}[y] \in \|\tau'\|_{(\text{VEnv}, L)}$

Since $\text{Dom}(\text{VEnv}') = \text{Dom}(\text{VEnv}) \cup \{ x \}$. We just need to prove the invariant for x . We have $C' \vdash x : \text{handle}(\rho, \tau)$ and $\text{VEnv}'''[x] = \text{region}(\rho)$, and so the invariant **Inv3** continues to hold.

– **Inv4**

From **Inv4** of (VEnv, L) we get $\forall \rho' \in \text{Dom}(L)$, if $C \vdash \rho' : \tau'$ then $\forall v \in \text{Dom}((L[\rho']).\text{RS})$, $L[\rho'].\text{RS}[v] \in \|\tau'\|_{(\text{VEnv}, L)}$.

$\forall \rho' \in \text{Dom}(L''')$,

- * if $\rho' \in \text{Dom}(L)$, then **Inv4** continues to hold because none of the previous regions changed.
- * if $\rho' = \rho$, then since $\text{Dom}(L'''[\rho].\text{RS}) = \emptyset$ the invariant holds trivially.

Hence **Inv4** holds.

– **Inv5**

From **Inv5** of (VEnv, L) we get $\forall \rho' \in \text{Dom}(L)$, $\text{Dom}((L[\rho']).\text{F}) \subseteq \text{Dom}((L[\rho']).\text{RS})$.

$\forall \rho' \in \text{Dom}(L''')$,

- * if $\rho' \in \text{Dom}(L)$, then **Inv5** continues to hold because none of the regions in L are changed.

- * if $\rho' = \rho$, then since $\text{Dom}(L''[\rho].\text{RS}) = \text{Dom}(L''[\rho].\text{F}) = \phi$, the invariant holds trivially.

Hence **Inv5** holds.

– **Inv6**

From **Inv6** of (VEnv, L) we get $\forall \rho_1 \rho_2 \in \text{Dom}(L)$, if $\rho_1 \neq \rho_2$ then $\text{Dom}((L[\rho_1]).\text{RS}) \cap \text{Dom}((L[\rho_2]).\text{RS}) = \phi$.

$\forall \rho_1 \rho_2 \in \text{Dom}(L'')$,

- * if $\rho_1, \rho_2 \in \text{Dom}(L)$, then **Inv6** continues to hold because none of the regions in L changed.
- * if either of ρ_1 or $\rho_2 = \rho$, then since $\text{Dom}(L''[\rho].\text{RS}) = \phi$, the invariant holds trivially.

Moreover, H does not change so the **Inv6** holds.

Hence $C' \vdash_{env} (\text{VEnv}'', L'')$.

Now using induction hypothesis, we get $(\text{VEnv}'', L'', S) \xrightarrow{*_{stmt}} (\text{VEnv}''', L''', \epsilon)$ and $C' \vdash_{env} (\text{VEnv}''', L''')$.

Using lemma **R16*** we get $(\text{VEnv}'', L'', \text{pool } \{ S \} \text{pop}(\rho)) \xrightarrow{*_{stmt}} (\text{VEnv}''', L''', \text{pool } \{ \epsilon \} \text{pop}(\rho))$.

Now $C' \vdash_{env} (\text{VEnv}''', L''')$.

So from **Inv1** we have $x \in \text{Dom}(\text{VEnv}''')$.

From **Inv3** we have $\text{VEnv}'''[x] = \text{region}(\rho)$.

From **Inv2** we have ρ in $\text{Dom}(L''')$.

Let $\text{VEnv}''' = \text{VEnv}' \cup \{(x, \text{region}(\rho))\}$ and $L''' = L' \cup \{(\rho, R)\}$.

Now rule **R17** applies.

$(\text{VEnv}''', L''', \text{pool } \{ \epsilon \} \text{pop}(\rho)) \xrightarrow{stmt} (\text{VEnv}', L', \epsilon)$.

Let H''' be the set of unused system addresses before this operation. and H' be the set of unused system addresses after. Then $H' = H''' \cup \text{Dom}(L'''[\rho].\text{RS})$.

We need to prove that $C \vdash_{env} (\text{VEnv}', L')$.

Note that $\text{Dom}(\text{VEnv}''') = \text{Dom}(\text{VEnv}') \cup \{x\}$ where $x \notin \text{Dom}(\text{VEnv}')$.

– **Inv1**

From **Inv1** of $(VEnv'', L'')$, we get $Dom(\Gamma') = Dom(VEnv'')$.

So $Dom(\Gamma) \cup \{x\} = Dom(VEnv') \cup \{x\}$. but $x \notin Dom(\Gamma)$ and $x \notin Dom(VEnv')$. So $Dom(\Gamma) = Dom(VEnv')$. q.e.d.

– **Inv2**

From **Inv2** of $(VEnv'', L'')$, we get $Dom(\Delta') = Dom(L'')$.

So $Dom(\Delta) \cup \{\rho\} = Dom(L') \cup \{\rho\}$. but $\rho \notin Dom(\Delta)$ and $\rho \notin Dom(L')$. So $Dom(\Delta) = Dom(L')$. q.e.d.

– **Inv3**

From **Inv3** of $(VEnv'', L'')$, we get $\forall y \in Dom(VEnv'')$, if $C' \vdash y : \tau$ then $VEnv''[y] \in \|\tau\|_{(VEnv''', L''')}$.

$\forall y \in Dom(VEnv')$, if $C \vdash y : \tau$ then we have $y \neq x$ and $y \in Dom(VEnv'')$.

$VEnv''[y] \in \|\tau\|_{(VEnv''', L''')}$. Since $y \neq x$, $VEnv''[y] = VEnv'[y]$. From **Well formed type lemma** and **Safe region deallocate lemma** we have $\|\tau\|_{(VEnv''', L''')} = \|\tau\|_{(VEnv', L')}$. q.e.d.

– **Inv4**

From **Inv4** of $(VEnv'', L'')$ we get $\forall \rho' \in Dom(L'')$, if $C' \vdash \rho' : \tau'$ then $\forall v \in Dom((L''[\rho']).RS)$, $L''[\rho'].RS[v] \in \|\tau'\|_{(VEnv''', L''')}$.

$\forall \rho' \in Dom(L')$, if $C \vdash \rho' : \tau'$ then we know that

$\rho' \neq \rho$ and $\rho' \in Dom(L'')$.

So we have, $\forall v \in Dom((L''[\rho']).RS)$, $L''[\rho'].RS[v] \in \|\tau'\|_{(VEnv''', L''')}$.

But $\rho' \neq \rho$ so $\forall v \in Dom((L'[\rho']).RS)$, $L'[\rho'].RS[v] \in \|\tau'\|_{(VEnv''', L''')}$.

Now using **Well formed type lemma** and **Safe region deallocate lemma** we have

$\|\tau'\|_{(VEnv''', L''')} = \|\tau'\|_{(VEnv', L')}$.

q.e.d.

– **Inv5**

From **Inv5** of $(VEnv'', L'')$ we get $\forall \rho' \in Dom(L'')$, $Dom((L''[\rho']).F) \subseteq Dom((L''[\rho']).RS)$.

This trivially holds for $(VEnv', L')$ since we have just taken element out of the map for L'' .

– **Inv6**

From **Inv6** of (VEnv''', L''') we get $\forall \rho_1 \rho_2 \in \text{Dom}(L''')$, if $\rho_1 \neq \rho_2$ then $\text{Dom}((L''')[\rho_1]).\text{RS} \cap \text{Dom}(L''')[\rho_2].\text{RS} = \phi$.

This trivially holds for (VEnv', L') since we have just taken an element out of the map for L''' .

Also, $\forall \rho' \in \text{Dom}(L')$, $\rho' \neq \rho$ and $\text{Dom}(L'[\rho']).\text{RS} = \text{Dom}(L''')[\rho'].\text{RS}$. From **Inv6** of (VEnv''', L''') $\text{Dom}(L''')[\rho'].\text{RS} \cap H''' = \phi$ and $\text{Dom}(L''')[\rho'].\text{RS} \cap \text{Dom}(L''')[\rho].\text{RS} = \phi$.

So, $\text{Dom}(L'[\rho']).\text{RS} \cap (H''' \cup \text{Dom}(L''')[\rho].\text{RS}) = \phi$.

Hence **Inv6** holds.

q.e.d.

Chapter 6

Backwards-Compatible Bounds Checking for C/C++

The SAFECode approach discussed in chapters 4 and 5, as explained earlier, may not detect some array bound violations. In this chapter, we present a novel backwards-compatible bounds checking technique for C that can be used in conjunction with the earlier solution.

The fundamental difficulty of bounds checking in C and C++ is the need to track at runtime, the intended target object of each pointer value (called the *intended referent* by Jones and Kelly [37]). Unlike safe languages like Java, pointer arithmetic in C and C++ allows a pointer to be computed into the middle of an array or string object and used later to further index into the object. Since such intermediate pointers can be saved into arbitrary data structures in memory and passed via function calls, checking the later indexing operations requires tracking the intended referent of the pointer through in-memory data structures and function calls. The compiler must transform the program to perform this tracking and this has proved to be a very difficult problem.

More specifically, there are three broad classes of solutions:

- *Use an expanded pointer representation (“fat pointers”) to record information about the intended referent with each pointer:* This approach allows efficient look-up of the pointer but the non-standard pointer representation is incompatible with external unchecked code, e.g. precompiled libraries. The difficulties of solving this problem in existing legacy code makes this approach largely impractical by itself. The challenges involved are described in more detail in Section 6.4.
- *Store the metadata separately from the pointer but use a map (e.g., a hash table) from pointers*

to metadata: This solution reduces but does not eliminate the compatibility problems of fat pointers, because checked pointers possibly modified by an external library must have their metadata updated at a library call. Furthermore, this adds a potentially high cost for searching the maps for the referent on loads and stores through pointers.

- *Store only the address ranges of live objects and ensure that intermediate pointer arithmetic never crosses out of the original object into another valid object* [37]: This approach, attributed to Jones and Kelly, stores the address ranges in a global table (organized as a splay tree) and looks up the table (or the splay tree) for the intended referent before every pointer arithmetic operation. This eliminates the incompatibilities caused by associating metadata with pointers themselves but current solutions based on this approach have even higher overhead than the previous two approaches. Jones and Kelly [37] report overheads of 5x-6x for most programs. Ruwase and Lam [57] extend the Jones and Kelly approach to support a larger class of C programs but report slowdowns of a factor of 11x-12x, if enforcing bounds for all objects, and of 1.6x-2x for several significant programs even if only enforcing bounds for strings. These overheads are far too high for use in “production code” (i.e., finished code deployed to end-users), which is important if bounds checks are to be used as a security mechanism (not just for debugging). For brevity, we refer to these two approaches as JK and JKRL in this work.

Note that compile-time checking of array bounds violations via static analysis is not sufficient by itself because it is usually only successful at proving correctness of a fraction (usually small) of array and pointer references [7, 22, 25, 26, 70]. Therefore, such static checking techniques are primarily useful to reduce the number of run-time checks.

An acceptable solution for production code would be one that has no compatibility problems (like the Jones-Kelly approach and its extension), but has overhead low enough for production use. A state-of-the-art static checking algorithm can and should be used to reduce the overhead but we view that as a way to reduce overhead by a constant fraction, for any of the run-time techniques. *The discussion above shows that none of the three current run-time checking approaches come close to providing such an acceptable solution, with or without static checking.*

In this chapter, we describe a method that dramatically reduces the run-time overhead of Jones

and Kelly’s “referent table” method with the Ruwase-Lam extension, to the point that we believe it can be used in production code (and static checking and other static optimizations could reduce the overhead even further). We propose two key improvements to the approach:

1. We exploit Automatic Pool Allocation (discussed in Chapter 3.4 to greatly reduce the cost of the referent lookups by partitioning the global splay tree into many small trees, while ensuring that the tree to search is known at compile-time. The transformation also safely eliminates many scalar objects from the splay trees, making the trees even smaller.
2. We exploit a common feature of modern operating systems to eliminate explicit run-time checks on loads and stores, which are a major source of additional overhead in the Ruwase-Lam extension. This technique also eliminates a practical complication of Jones and Kelly, namely, the need for one byte of padding on objects and function parameters, which compromises compatibility with external libraries.

We also describe a few compile-time optimizations that reduce the sizes of the splay trees, by a large extent in some cases, or reduce the number of referent lookups. As discussed in Section 6.1.4, our approach preserves compatibility with external libraries (the main benefits of the JK and JKRL methods) and detects all errors detected by those methods except for references that use pointers cast from integers.

Section 6.1 briefly describes the Jones-Kelly algorithm with the Ruwase-Lam extension and then describes how we maintain and query the referent object maps on a per-pool basis. It also describes three optimizations to reduce the cost of referent object queries. Section 6.3 describes our experimental evaluation and results. Section 6.4 compares our work with previous work on array bounds checking. Section 6.5 concludes with a summary.

6.1 Runtime Checking with Efficient Referent Lookup

6.1.1 The Jones-Kelly Algorithm and Ruwase-Lam Extension

Jones and Kelly rely on, and strictly enforce, three properties of ANSI C in their approach: (1) Every pointer value at run-time is derived from the address of a unique object, which may be a

declared variable or memory returned by a single heap allocation, and must only be used to access that object. Jones and Kelly refer to this as the *intended referent* of a pointer. (2) Any arithmetic on a pointer value must ensure that the source and result pointers point into the same object or at most one byte past the end of the object (the latter value may be used for comparisons, e.g., in loop termination, but not for loads and stores). (3) Because of the potential for type-converting pointer casts, it is not feasible in general to distinguish distinct arrays within a single allocated object defined above, e.g., two array fields in a `struct` type, and the Jones-Kelly technique does not attempt to do so.

Jones and Kelly maintain a table describing all allocated objects in the program and update this table on `malloc/free` operations and on function entry/exit. To avoid recording the intended referent for each pointer (this is the key to backwards compatibility), they check property (2) strictly on every pointer arithmetic operation, which ensures that a computed pointer value always points within the range of its intended referent. Therefore, the intended referent can be found by searching the table of allocated objects.

More specifically, they insert the following checks (ignoring any later optimization) on each arithmetic operation involving a pointer value:

JK1. check the source pointer is not the invalid value (-2);

JK2. find the referent object for the source pointer value using the table;

JK3. check that the result pointer value is within the bounds of this referent object plus the extra byte. If the result pointer exceeds the bounds, the result -2 is returned to mark the pointer value as invalid.

JK4. Finally, on any load or store, perform checks [JK1-JK3] but JK3 checks the source pointer itself.

Assuming any dereference of the invalid value (-2) is disallowed by the operating system, the last run-time check (JK4) before loads and stores is strictly not necessary for bounds checking. It is, however, a useful check to detect some (but not all) dereferences of pointers to freed memory and pointer cast errors. The most expensive part of these checks is step (JK2), finding the referent object by searching the table. Jones-Kelley use a data structure called a splay tree to record the

valid object ranges, which must be disjoint. Given a pointer value, they search this tree to find an object whose range contains that value.

If no valid range is found for a given pointer value, the pointer must have been derived from an object allocated by uninstrumented part of the program, e.g., an external library, or by pointer arithmetic in such a part of the program (since no legal pointer can ever be used to compute an illegal one). Such pointers values cannot be checked and therefore, step (JK3) is skipped, i.e., any array bound violations may not be detected.

One complication in Jones-Kelley's work is that, because a computed pointer may point to the byte after the end of its referent object, the compiler must insert padding of one-byte (or more) between any two objects to distinguish a pointer to the "extra" byte of the first object from a pointer to the second object. They modify the compiler and the `malloc` library to add this extra byte on all allocated objects. Objects can also be passed as function parameters, however, and inserting padding between two adjacent parameters could cause the memory layout of parameters to differ in checked and unchecked code. To avoid this potential incompatibility, they do not pad parameters to any function call if the call could invoke an unchecked function and also they do not pad formal parameters in any function that may be called from unchecked code. In the presence of indirect calls via function pointers, the compiler must be conservative about identifying such functions.

A more serious difficulty observed by Ruwase and Lam is that rule (2) above is violated by many C programs (60% of the programs in their experiments), and hence is too strict for practical use. The key problem is that some programs may compute illegal intermediate values via pointer arithmetic but never use them. For example, in the sequence `{q = p+12; r = q-8; N = *r;}`, the value `q` may be out-of-bounds while `r` is within bounds for the same object as `*p`. Jones and Kelly would reject such a program at `q = p+12` because the correct referent cannot be identified later (`q` may point into an arbitrary neighboring object).

Ruwase and Lam extend the JK algorithm essentially by tracking the intended referent of pointers explicitly but only in the case where a pointer moves out of bounds of its intended referent. For every such out-of-bounds pointer, they allocate an object called the OOB (Out-Of-Bounds) object to hold some metadata for the pointer. The pointer itself is modified to point to the OOB

object, and the addresses of live OOB objects are also entered into a hash table. This hash table is checked only before accessing the OOB object to ensure that it is a valid OOB object address. The OOB object includes the actual pointer value itself and the address of the intended referent (saved when the pointer first goes out of bounds). Any arithmetic on the pointer is performed on the value in the OOB object. If the pointer value comes back within bounds, the original pointer is restored to its current value and the OOB object is deallocated.

The extra operations required in the Ruwase-Lam extension are: (1) to allocate and initialize an OOB object when a pointer first goes out-of-bounds; (2) on any pointer arithmetic operation, if the pointer value does not have a valid referent *and* cannot be identified as an unchecked object, search the OOB hash table to see if it points to an OOB object, and if so, perform the operation on the value in the OOB object; (3) when an object is deallocated (implicitly at the end of a program scope or explicitly via `free` operation), scan the OOB object hash table to deallocate any OOB objects corresponding to the referent object that is being deallocated.

The first two operations add extra overhead only for out-of-bounds pointers, which would have caused the program to halt with a run-time error in the JK scheme. The third operation is required even in the case of strictly correct program behavior allowed by J-K. Perhaps more importantly, step JK4 of Jones-Kelley, is now necessary for bounds checking since dereferencing OOB objects is disallowed. In particular, if we wish to combine this approach with other techniques for detecting *all* dereferences to freed memory ([71, 20]) or all pointer cast errors ([52, 21]), we would still need to perform JK4 (or a variant which checks that OOB objects are never dereferenced).

6.1.2 Our Approach

Our approach is based on the Jones-Kelley algorithm with the RL extension but with two key improvements that greatly reduce the run-time overhead in practice and makes the approach useful in production level systems. In fact, the improvements are dramatic enough that we are even able to use our system for checking all array operations (not just strings) and still achieve much lower overheads than the JK or RL approaches (even compared with the RL approach applied only to strings). The two improvements are: (1) Exploiting Automatic Pool Allocation [46] for much faster searches for referent objects; and (2) An extra level of indirection in the RL approach for OOB

```

f() {
  A = malloc(...)
  ...
  while(..) {
    ...
    A[i] = ...
  }
}

f() {
  PoolDescriptor PD
  A = poolalloc(&PD,...)
  ...
  while(..) {
    ...
    Atmp = getreferent(&PD, A);
    boundscheck(Atmp, A+i);
  }
}

```

Figure 6.1: Sample code before and after bounds checking instrumentation

pointers that eliminates the need for run-time checks on most loads and stores.

The Jones-Kelley approach, and in turn Ruwase-Lam extension, rely on one splay data structure for the entire heap. Every memory object, except for a few stack objects whose address is not taken, is entered in this big data structure. This data structure is looked up for almost every access to memory or pointer arithmetic operation. For a program with large number of memory objects, the size of the data structure could be very large, making the lookups quite expensive.

The main idea behind our first improvement is to exploit the partitioning of memory created by Automatic Pool Allocation to reduce the size of the splay tree data structures used for each search operation. Instead of using one large splay tree for the entire program, we maintain one splay tree per pool. The size of each individual splay tree is likely to be much smaller than the combined one. Since the complexity of searching the splay tree for uniform accesses is amortized $O(\log_2 n)$ (and better for non-uniform accesses), the lookup for each pointer access is likely to be much faster than in the JK or RL approaches.

A key property that makes this approach feasible is that the pool descriptor for each pointer is known at compile-time. Without this, we would have to maintain a run-time mapping from pointers to pools, which would introduce a significant extra cost as well as the same compatibility problems as previous techniques that maintain metadata on pointers.

Algorithm

The steps taken by the compiler in our approach are as follows:

1. First, pool-allocate the program. Let `Pools` be the map computed by the transformation

giving the pool descriptor for each pointer variable.

2. For every pointer arithmetic operation in the original program, $p = q + c$, insert a run-time check to test that p and q have the same referent. We use the function `getreferent(PoolDescriptor *PD, void *p)` to look up the intended referent of a pointer, p . The pool descriptor, PD , identifies which splay tree to lookup. For the instruction $p = q + c$, we compute p , then invoke `getreferent(Pools[q], q)`, and finally check that p has the same referent as q using the function call `boundscheck(Referent *r, void *p)`.
3. The correct pool descriptor for a pointer q may not be known either if the value q is obtained from an integer-to-pointer cast or from unchecked code (e.g, as a result of a call to an external function). The latter case is discussed in Section 6.1.4. The two cases can be distinguished via the flags on the target points-to graph node: the former case results in a **U** (Unknown) flag while the latter results in a missing **C** (complete) flag, i.e., the node is marked incomplete. In the former case, the pointer may actually point to an object allocated in the main program, i.e., the object has a valid entry in the splay tree of some pool but we do not know which pool at compile-time. We do not check pointer arithmetic on such pointers. This is weaker than Jones-Kelly as it might allow bound violations on such pointers to go undetected.

Handling Non-Heap Data

The original pool allocation transformation only created pools to hold heap-allocated data. We would like to create partitions of globals and stack objects as well, to avoid using large combined splay trees for those objects. The pointer analysis underlying pool allocation includes points-to graph nodes for all memory objects, including global and stack objects. We have already extended pool allocation so that it assigns pool descriptors to all global and stack objects as well, *without changing how the objects are allocated* (see chapter 4.4). Pool allocation already created pool descriptors for points-to graph nodes that include heap objects as well as global or stack objects. We only had to modify it to also create “dummy” pool descriptors for nodes that included only global or stack objects. The transformation automatically ensures that the objects are created in the appropriate function (e.g., in `main` if the node includes any globals). We call these “dummy” pool descriptors because no heap allocation actually occurs using them: they simply provide a

logical handle to a compiler-chosen subset of memory objects.

For the current work, we have to record each object in the splay tree for the corresponding pool. We do this in `main` for global objects and in the appropriate function for stack-allocated variables (many local variables are promoted to registers and do not need to be stack-allocated or recorded). The bounds checks for operations on pointers to such pools are unchanged.

6.1.3 Handling Out-Of-Bounds Pointers

The Ruwase-Lam extension to handle OOB pointers requires expensive checks on all loads/stores in the program (before any elimination of redundant checks). In this work, we propose a novel approach to handle out of bounds values in user-level programs without requiring checks on any individual loads or stores.

Whenever any pointer arithmetic computes an address outside of the intended referent, we create a new OOB object and enter it into a hash-table recording the OOB object address (just like Ruwase-Lam). We use a separate OOB hash-table per pool, for reasons described below. The key difference is that, instead of returning the address of the newly created OOB object and recording that in the out-of-bounds pointer variable, we return an address from a part of the address space of the program reserved for the kernel (e.g., addresses greater than `0xbfffffff` in standard Linux implementations on 32-bit machines). Any access to this address by a user level programs will cause a hardware trap¹. Within each pool, we maintain a second hash table, mapping the returned value and the OOB object. Note that we can reuse the high address space for different pools and so we have a gigabyte of address space (on 32 bit linux systems) for each pool for mapping the OOB objects.

A load/store using out of bounds values will immediately result in a hardware trap and we can safely abort the program. However, all pointer arithmetic on such values needs to be done on the actual out of bounds value. So on every pointer arithmetic, we first check if the source pointer lies in the high gigabyte and then lookup the OOB hash map of the pool to get the corresponding OOB object. This OOB object contains the actual out of bounds value. We perform the pointer arithmetic on the actual out of bounds value. If the result after arithmetic goes back in to the

¹If no such reserved range is available, e.g. we are doing bounds-checking for kernel modules, then we will need to insert checks on individual loads and stores just like the Ruwase-Lam extension.

bounds of the referent then we return that result. If the result after arithmetic is still out of bounds, we create a new OOB object and store the result in the new OOB. We then map this new OOB to an unused value in the high gigabyte, store the value along with the OOB object in the OOB hash map for the pool and return the value. Note that just like Ruwase-Lam, we need to change all pointer comparisons to take in to account the new out of bound values.

Step 2 in our approach is now modified as follows:

For every pointer arithmetic operation in the original program, $p = q + c$, we first check if q is a value in the high gigabyte. This is an inexpensive check and involves one comparison. There are two possibilities.

- Case 1: q is not in the high giga byte.

Here we do the bounds check as before but with one key difference. If the result p is out of bounds of the referent of q , then instead of flagging it as an error, we create a new OOB object to store the out of bounds value just like Ruwase-Lam extension. Now we map this OOB object to a value in the high address space and assign this high address space value to p .

- Case 2: q is a value in the high address space.

We do the following new check (from the discussion above): We first get the corresponding OOB object for that address using the hash map in the pool. We then retrieve the actual out of bounds value from the OOB object and do the arithmetic. If the result is within the bounds of the referent then we assign the result to p and proceed. If the result is still outside the bounds of the referent, then we create a new OOB object just like in Case 1.

6.1.4 Compatibility and Error Detection with External Libraries

Automatic Pool Allocation transformation may modify function interfaces and function calls to add pool descriptors. As in the case of other SAFECode work (see chapter 4.5.6) the transformation and our bounds checking algorithm can be implemented to work correctly and fully automatically with uninstrumented external code (e.g., external libraries), although a few out-of-bound accesses may not be detected. First, to preserve compatibility, calls to external functions are left unmodified. Second, if an internal function may be called from external code, we must ensure that the

external code calls the original function, not the pool-allocated version. This ensures backwards-compatibility but makes it possible to miss bounds errors in the corresponding function. In most cases, we can directly transform the program to pass in the original function and not the pool-allocated version (this change can be made at compile-time if it passes the function name but may have to be done at run-time if it passes the function pointer in a scalar variable). In the general case (which we have not encountered so far), the function pointer may be embedded inside another data structure. Even for most such functions, the compiler can automatically generate a “`varargs`” wrapper designed to distinguish transformed internal calls from external calls. When this is not possible, we must leave the callback function (and all internal calls to it), completely unmodified.

Except in call-back functions, bounds checks can still be performed within the available program for *all* heap-allocated objects (internal or external). Like JK, we intercept all direct calls to `malloc` and record the objects in a separate global splay tree. For pointer arithmetic on a pointer to an incomplete node, we check both the splay tree of the recorded pool for that node and the global splay tree. All heap objects must be in one of those trees, allowing us to detect bounds violations on all such objects.

Internal global and stack objects will be recorded in the splay tree for the pool and hence arithmetic on pointers to them can be checked. We cannot check any static or stack objects allocated in external code since we do not know the size of the objects. The JK and JKRL techniques have the same limitation.

6.1.5 Errors in Calling Standard Library Functions and System Calls

More powerful error checking is possible for uses of recognized standard library functions and system calls. Many bugs triggered inside such functions are due to incorrect usage of library interfaces and not bugs within the library itself. We can guard against these interface bugs by generating wrappers for each potentially unsafe library routine; the wrappers first check the necessary preconditions on buffers passed to the library call and then invoke the actual library call. For example, for a library call like `memcpy(void *s1, const void *s2, size_t n)`, we can generate a wrapper that checks (1) `n > 0`, (2) the object pointed to by `s2` has at least `n` more bytes starting from `s2` and (2) the object pointed to by `s3` has at least `n` more bytes starting from `s3`. These checks can be done using

the same `getreferent` and `boundscheck` functions as before.

Note that the wrappers referred to here are not for compatibility between checked code and library code, and are only needed if extra bug detection is desired. The wrappers are independent of the program and only need to be written once per the library. We have already written the wrappers for many of the standard C library functions.

6.1.6 Optimizations

There are a number of ways to further reduce the overheads of our run-time checks. We briefly describe three optimizations that we have implemented. The first optimization below is specific to our approach because it requires a key property of pool allocation. The other two are orthogonal to the approach for finding referents and can also be used with the Jones-Kelly or Ruwase-Lam approaches.

First, we observe that a very large number of single-element objects (which may be scalars or single-element arrays) are entered into the splay trees in all three approaches. Since a pointer to any such object can be cast and then indexed as a pointer to an array (e.g., an array of bytes), references to all such objects (even scalars) must be checked for bounds violations. While many local scalars of integer or floating point type are promoted to registers, many other local and all global scalars may still stay memory-resident. Entering all such scalars into the search trees is extremely wasteful since few programs ever index into such scalars, legally or illegally. We propose a technique to avoid entering single-element objects into search trees while still detecting bounds violations for such objects.

To achieve this goal, two challenges must be solved: (1) to identify single-element object allocations, and (2) to detect bounds violations even if such objects are not in the splay trees. For the former, we observe that most pools even in C and C++ programs are *type-homogeneous* [46], i.e., all objects in the pool are of a single type or are arrays of that type. For non-type-homogeneous pools, the pool element type is simply a byte. Furthermore, all objects in such a pool are aligned on a boundary that is an exact multiple of the element size. The size of the element type is already recorded in each pool at pool creation time. This means that the run-time can detect allocations of scalars or single-element arrays: these are objects whose size is exactly the size of the pool element

type. We simply do not enter such objects into the splay tree in the pool.

For the latter problem, the specific issue is that a referent look-up using a valid pool descriptor will not find the referent object in the splay tree. This can only happen for two reasons: (i) the object was a one-element object, or (ii) the object was an unchecked object or a non-existent object but the pointer being dereferenced was assigned the same pool during pool allocation. The latter can happen, for example, with code of the form:

```
T* p = some_cond? malloc(..) : external_func(..);
```

Here, the pointer `p` is assigned a valid pool because of the possible `malloc`, but if it points to an object returned by the external function `external_func`, the referent lookup will not find a valid referent. The same situation arises if the pointer `p` were assigned an illegal value, e.g., from an uninitialized pointer or by casting an integer. To distinguish the first case from the second, we simply use the pool metadata to check if the object is part of the pool. This check is exactly the `poolcheck` operation discussed in chapter 4. Thus, we can successfully identify and omit single element arrays from the splay trees and yet detect when they are indexed illegally.

The next two optimizations are far simpler and not specific to our approach. They both exploit the fact that it is very common for a loop nest or recursion to access very few arrays (often one or two) repeatedly. Since all accesses to the same array have the same referent, we can exploit this locality by using a small lookup cache before each splay tree. We use a two-element cache to record the last two distinct referents accessed in each pool. When an access finds the referent in the cache, it reduces overhead because it avoids the cost of searching the splay tree to find the referent (we found this to be more expensive even if the search succeeds at the root) and also the cost of rotating the root node when successive references to the same pool access distinct arrays. It increases the overhead on a cache miss, however, because all cache elements must be compared before searching the splay tree. We experimented with the cache size and found that a two-element cache provided a good balance between these tradeoffs and improved performance very significantly over no cache or a one-element cache.

The third optimization attempts to achieve the same effect via a compile-time optimization, viz., loop-invariant code motion (LICM) of the referent lookup. We find that the two-element cache is important even with this optimization because LICM sometimes fails, e.g., within recursion, if

the loop nest is spread across multiple functions, or the referent lookup does not dominate all loop exits. Implementing this optimization is easy because the referent lookup is a pure function: the same pointer argument always returns the same referent object (or none). Therefore, the lookup is loop-invariant if and only if the pointer is loop-invariant.

6.2 Compiler Implementation

We have implemented our approach using the LLVM compiler infrastructure [45]. LLVM already includes the implementation of Automatic Pool Allocation, using a context-sensitive pointer analysis called Data Structure Analysis (DSA). We implemented the compiler instrumentation as an additional pass after pool allocation. We also run a standard set of scalar optimizations needed to clean up the output of pool allocation [46]. Because DSA and pool allocation are interprocedural passes, this entire sequence of passes is run at link-time so that they can be applied to as complete a program as possible, excluding libraries available only in binary form. Doing cross-module transformations at link-time is standard in commercial compilers today because it preserves the benefits of separate compilation.

Our implementation includes three optimizations described earlier: leaving out single-element objects from the splay tree in each pool, the two-element caching to reduce searches of the splay tree, and moving loop-invariant referent lookups out of loops. We have also implemented an aggressive interprocedural static array bounds checking algorithm, which can optionally be used to eliminate a subset of run-time checks (see chapter 9).

We compile each application source file to the LLVM compiler IR with standard intra-module optimizations, link the LLVM IR files into a single LLVM module, perform our analyses and insert run-time checks, then translate LLVM back to ANSI C and compile the resulting code using GCC 3.4.4 at -O3 level of optimization. The final code is linked with any external (pre-compiled) libraries.

In terms of compilation time, DSA and Automatic Pool Allocation are both very fast, requiring less than 3 seconds combined for programs up to 130K lines of code that we have tested. This time is in fact a small fraction of the time taken by gcc or g++ at -O3 for the same programs) [46]. The additional compiler techniques for bounds checking described and implemented in this work add negligible additional compile time.

6.3 Experiments

We present an experimental evaluation of our bounds checking technique with the following goals:

- To measure the net overhead incurred by our approach.
- To isolate the effect of using multiple distinct splay trees and the associated optimizations, which is our key technical improvement over the Ruwase-Lam (and so Jones-Kelley) approaches.
- To evaluate the effectiveness of our approach in detecting known vulnerabilities. For this purpose, we use Zitser’s suite of programs modeling vulnerabilities found in real-world software [76].

It is also interesting to confirm the backwards-compatibility of our approach. In our experience so far, we have required *no* changes to any of the programs we have evaluated, i.e., our compiler works on these programs “out-of-the-box” (all the wrappers required for correct usage of standard library calls are already generated and need not be written again). This is similar to Jones-Kelly and Ruwase-Lam but significantly better than other previous techniques that use metadata on pointers, applied to the same programs, discussed in Section 6.3.3.

6.3.1 Overheads

We have evaluated the run-time overheads of our approach using the Olden [10] suite of benchmarks, and the unix daemons, ghttpd, bsd-fingerd, and wu-ftpd-2.6.2. We use the Olden benchmarks because they are pointer-intensive programs that have been used in a few previous studies of memory error detection tools [71, 52, 73]. We compare our overheads with these and other reported overheads in Section 6.3.3. The benchmarks and their characteristics are listed in Table 6.1. The programs are compiled via LLVM and GCC, as described in the previous section. For the benchmarks we used a large input size to obtain reliable measurements. For the daemon programs we ran the server and the client on the same machine to avoid network overhead and measured the response times for client requests.

The “LLVM (base)” column in the table represents execution time when the program is compiled to the LLVM IR with all standard LLVM optimizations (including the standard optimizations

used to clean up after pool allocation, but not pool allocation itself), translated back to C code, and the resultant code is compiled directly with GCC -O3. The “PA” column shows the time when we run the above passes as well as the pool allocator but without any run-time checks. Notice that in a few cases, pool allocation speeds up the program slightly but doesn’t significantly degrade the performance in any of these cases. We use the LLVM(base) column as the baseline for our experiments in calculating the net overhead of our bounds checking approach because we believe that gives the most meaningful comparisons to previous techniques. Since Automatic Pool Allocation can be used as a separate optimization, the PA column could be used as a baseline instead of LLVM(base), but the two are close enough for the benchmarks in the table that we do not expect this choice to affect our conclusions.

The “BoundsCheck” column shows the execution times with bounds checking. Here, we have turned on the three optimizations that we have discussed in Section 6.1.6: caching on top of the splay tree, loop invariant code motion, and not storing single-element objects in the splay tree. The “Slowdown” ratio shows the net overhead of our approach relative to the base LLVM. In almost half of the benchmarks, we found that overheads are within 3% of the base LLVM. Only two programs (em3d, health) have overheads greater than 25%.

In order to isolate the benefits of smaller splay data structures, we conducted another experiment. The pool allocator pass provides an option to merge all the pools in the program into one single global pool. This pool uses the same memory allocation algorithm as before but puts all tracked objects into a single splay tree. This allowed us to isolate the effect of using multiple splay trees instead of the single splay tree used by **JK** and **JKRL**. Note that we cannot use optimization one (leaving singleton objects out of the splay tree) because after merging pools, type information for the pool is lost and we cannot identify singleton object allocations. The other two optimizations, caching splay tree results and LICM for referent lookups, are used, which is again appropriate because they can also be used with the previous approaches. Columns “PA with one pool” and “PA with one pool + bounds checking” show the execution times of this single-global-pool program without and with our run-time checks, and the last column shows the ratio of the two. The benchmark `health` used up all system memory and started thrashing. The main reason is we could not eliminate singleton objects from the splay tree, making the single global splay tree

Name	LOC	Base LVM	PA	Bounds-Check	Our slow-down ratio	PA with one pool	PA with one pool + bounds checks	One pool ratio
bh	2053	9.146	9.156	9.138	1.00	9.175	10.062	1.10
bisort	707	12.982	12.454	12.443	0.96	12.425	14.172	1.14
em3d	557	6.753	6.785	11.388	1.69	6.803	11.419	1.68
health	725	14.305	13.822	19.902	1.39	13.618	-	-
mst	617	12.952	12.017	15.137	1.17	12.203	28.925	2.37
perimeter	395	2.963	2.601	2.587	0.87	2.547	6.306	2.48
power	763	2.943	2.920	2.928	0.99	2.925	2.931	1.00
treeadd	385	17.704	17.729	17.310	0.98	17.706	21.063	1.19
tsp	561	7.086	6.989	7.219	1.02	6.978	8.897	1.27
AVG					1.12			
Applications								
fingerd	336	2.379	2.384	2.475	1.04	2.510	2.607	1.04
ghttpd	837	11.405	9.423	9.466	0.83	11.737	12.182	1.03
ftpd	23033	1.551	1.539	1.542	0.99	1.551	1.546	1.00

Table 6.1: Benchmarks and Run-time Overheads. The One-Pool Ratio compared with Our Slow-down Ratio isolates the benefit of partitioning the splay-tree.

much larger than the combined splay trees in the original code. Comparing the last column with the column labelled “Our Slowdown Ratio” shows that in at least three cases (health, mst, and perimeter) the overheads when using multiple search data structures is dramatically better (more than 100%) than using a single data structure for the entire heap. The benefits are also significant in tsp and bisort. The remaining programs show little difference in overheads for the two cases.

6.3.2 Effectiveness in Detecting Known Attacks

We used Zitser’s suite of programs modeling real-world vulnerabilities [76] to evaluate the effectiveness of our approach in detecting buffer overrun violations in real software. The suite consists of 14 model programs, each program containing a real world vulnerability reported in bugtraq. 7 of these vulnerabilities were in `sendmail`, 3 were in `wu-ftpd`, and 4 were in `bind`. This suite has been used previously to compare dynamic buffer overflow detection approaches [75].

The results of our experiments are reported in Figure 6.3.2. We are able to detect all the vulnerabilities in all three programs out of the box. In each case, the illegal memory reference was detected and the program was halted with a run-time error. The four bugs in `bind` are not triggered in the main program but in a library routine (e.g. due to passing a negative argument to

Program	No. of vulnerabilities	No. of vulnerabilities detected	No. of vulnerabilities detected with std. lib. check
sendmail	7	7	7
bind	4	0	4
wu-ftpd	3	3	3

Figure 6.2: Effectiveness of our approach in detecting known attacks/vulnerabilities

memcpy). These bugs are automatically detected by our approach using the wrappers described earlier because they are due to incorrect usage of the library functions (and not bugs within the library).

6.3.3 Performance Comparison with Previous Approaches

Finally, we briefly compare the overheads observed in our work with those reported in other related work, to the extent possible. We can make direct comparisons in cases where there are published results for Olden suite of benchmarks. When such numbers are not available, only a rough comparison is possible and only in cases where the differences are obviously large. Also, note that some previous techniques including [71, 54] detect a wider range of bugs than we do in the current work. Where possible, we try to compare the overheads they incur due to bounds checking alone.

The two previous approaches with no compatibility problems, JK and JKRL, have both reported far higher overheads than ours, as noted earlier. Jones and Kelly say that in practice, most programs showed overheads of 5x-6x. Ruwase and Lam report slowdowns up to a factor of 11x-12x if enforcing bounds for all objects, and up to a factor of 1.6x-2x for several significant programs even if only enforcing bounds for strings. Their overheads are even higher than those of Jones and Kelly because of the additional cost of checking all loads and stores and also the cost of checking for OOB objects that may have to be deallocated as they go out of bounds. While two of our optimizations (the two-element cache and LICM for loop-invariant referent lookups) might reduce these reported overheads, it seems unlikely that they would come close to our reported overheads. Our overheads are dramatically lower than these previous techniques due to the combination of three factors: using multiple splay trees (whose benefit was shown earlier), not requiring checks on loads and stores, and the additional optimizations.

Xu. et al [71] have proposed to use metadata for pointer variables that is held in a separate

data structure that mirrors the program data in terms of connectivity. They use the metadata to identify both spatial errors (array bounds, uninitialized pointers) and temporal errors (dangling pointer errors). Their average overhead for Olden benchmarks for just the spatial errors is 1.63 while ours is far less at 1.12. Moreover, their approach incurs some difficulties with backwards compatibility, as described in Section 6.4.

CCured [52] divides the pointers of the program into safe, seq pointers (for arrays) and wild (potentially unsafe) pointers at compile-time. At run-time CCured checks that seq pointers never go out of bounds and wild pointers do not clobber the memory of other objects. While CCured checking for WILD pointers is more extensive than ours, in the case of Olden benchmarks, they did not encounter any wild pointers [52]. It is important to note, however, that CCured uses garbage collection for dynamic memory management and the overhead due to garbage collection is unknown. The reported average overhead for Olden is 1.28, which is only slightly higher than our observed overheads. However, they needed to change 1287 lines of code in total to achieve these results while our technique works out of the box.

Yong et al [73] describe a technique to identify many illegal write references and free operations via pointers, by identifying a set of pointers that might be unsafe using a pointer-analysis and tagging the memory corresponding to the objects those pointers may point to. They use a shadow memory with 1 tag bit per byte of memory, setting this tag bit on allocations and clearing them on deallocations. They check these tag bits on every write or free of a potentially unsafe pointer, allowing them to detect a number of potential security attacks and errors such as accesses to a freed memory that has not been reallocated. They report an average overhead of 1.37x for the Olden benchmarks (the fraction of overhead due to array references is unknown). Unlike our work and the previous papers described above, they do not perform any checks on read operations and read operations are far more frequent than writes.

6.4 Related Work

We focus our comparisons on techniques for run-time bounds checking, and any optimizations directly related to those techniques. We do not discuss existing compile-time techniques for bounds checking here (including our own), because these techniques are complementary and can be used

to eliminate some run-time checks in any of the approaches discussed here.

There are a number of debugging tools like `purify` and `valgrind` that use binary instrumentation to detect a wide range of memory referencing errors. Using binary instrumentation allows these tools to add arbitrary metadata to pointers without the compatibility problems of other approaches. These tools, however, incur very high run-time overheads, e.g., often greater than a factor of 10x for `purify` and `valgrind`. Also, in case of `purify` it does not catch some pointer arithmetic violations if the arithmetic yields a pointer to a valid region [37].

A number of other approaches target debugging but work at the source level. These include Loginov’s work on runtime type checking [50], Kendall’s `bcc` [39], and Steffens’ `rtcc` [60]. All of these approaches focus on debugging and usually performance is not a serious consideration. For instance, the reported overheads for Loginov’s work are up to 900%.

Some tools including SafeC [4] and Cyclone [35] use an augmented pointer representation that includes the object base and size of the legal target object for every pointer value. Such “fat pointers” require significant changes to programs to allow the use of external libraries, typically introducing wrappers around library calls to convert pointer representations. Furthermore, writing such wrappers may be impractical for indirect function calls and for functions that access global variables or other pointers in memory. Unlike the remaining techniques below, fat pointers have the major advantage of no cost in finding the metadata for each pointer value.

To reduce the compatibility problems caused by fat pointers, several recent systems store pointer metadata separately from the pointer variables themselves, at the cost of significantly greater overhead for finding the metadata associated with each pointer. This approach was used by Patil and Fisher [54], CCured [52], and Xu et al. [71]. Separating the metadata eliminates the potential for program failures mentioned above and reduces the need for wrappers on library calls. This technique does not require wrappers for pointers passed *to* library functions or pointer values explicitly returned by such functions. Wrappers are still needed for checked pointers that may be modified indirectly as a side-effect of a library call because the metadata before the call would be invalid if the call overwrites the pointer. Such wrappers are likely to be needed less often but if needed, may be impractical to write for the same reasons as with fat pointers, described above. The work of Xu et al. is also more restrictive than ours because they restrict pointer casts between

structures of incompatible types. Finally, and most important from a practical viewpoint, all these techniques have significantly higher overhead than ours, as discussed in more detail in Section 6.3.3.

As noted earlier, the compatibility problems of both fat pointers and pointers with separately stored metadata occur because the metadata is associated with the pointer itself and not the object that is the target of a pointer. The work of Jones and Kelly [37] and Ruwase and Lam [57] associate metadata with objects instead of pointers, which greatly reduces the compatibility problem. However, the overheads of these two approaches are quite high. As the comparison in Section 6.3.3 shows, our approach is able to reduce these overheads greatly; sufficient, we believe, for the technique to be used in production code.

6.5 Run-time Bounds Checking: Concluding Discussion

We have described a collection of techniques that dramatically reduce the overhead of an attractive, fully automatic approach for run-time bounds checking of arrays and strings in C and C++ programs. Our techniques are essentially based on a fine-grain partitioning of memory. They bring the average overhead of run-time checks down to only 12% for a set of benchmarks we have evaluated. Thus, we believe we have achieved the twin goals that have not been simultaneously achieved so far: overhead low enough for production use, and fully automatic checking, i.e., not requiring manual effort to circumvent compatibility problems or to assist the compiler's checking techniques.

Chapter 7

Efficient Detection of All Dangling Pointer Uses

A major limitation of the SAFECode work described thus far is that it allows references of dangling pointers to freed memory to go undetected. Detecting these dangling pointer errors is important from a security stand point since they can also be exploited in much the same way as buffer overruns to compromise system security [74]. In fact, many exploits that take advantage of a subclass of these errors (double free vulnerabilities) in server programs have been reported in bugtraq (e.g., CVS server double free exploit [23], MIT Kerberos 5 double free exploit [2], MySQL double free vulnerability [1]). Efficient detection of all such errors in servers during deployment, rather than just during development, is crucial for security.

Unfortunately, detecting dangling pointer errors in programs has proven to be an extremely difficult problem. Detecting such errors statically in any precise manner is undecidable. Detecting them efficiently at run-time while still allowing safe reuse of memory can be very expensive and we do not know of any practical solution that has overheads low enough for use in production code.

A number of approaches (including [4, 33, 37, 51, 56, 54, 58, 71, 73]) have been proposed that use some combination of static and run-time techniques to detect several kinds of memory errors, including buffer overflow errors and some dangling pointer errors. All of these techniques either have prohibitively high run-time overheads (2x - 100x) or memory overheads (or both) and are unsuitable for production software. Purify [33] and Valgrind [58], two of the most widely used tools for debugging memory access errors, often have overheads in excess of 1000% and can sometimes be too slow even for debugging long-running programs. Moreover, most of these approaches (except

FisherPatil [54], Xu et al [71] and Electric Fence [56]) employ only heuristics to detect dangling pointer errors and do not provide any guarantees about absence of such errors. FisherPatil and Xu et al, detect all dangling pointer errors but perform software run-time checks on *all individual* loads and stores, incurring overheads up to 300% and also causing substantial increases in virtual and physical memory consumption (1.6x-4x). Electric Fence uses page protection mechanisms to detect all dangling pointer errors but does so at the expense of several fold increase in virtual *and* physical memory consumption of the applications.

In this chapter, we present a new technique that can detect dangling pointers in server code with very low overheads; low enough that we believe it can be used in production code though it is also useful for debugging. Our approach builds on the naive idea (previously used in Electric Fence [56], PageHeap [51]) of using a new virtual and physical page for each allocation of the program. Upon deallocation, we change the permissions on the individual virtual pages and rely on the memory management unit (MMU) to detect all dangling pointer accesses. This naive idea has two problems that make it impractical for any use other than debugging: increased address space usage (one virtual page for each allocation) and increased physical page usage (one page for each allocation). Our technique is based on two key insights that alleviate both these problems. Our first insight is based on the observation that even when using a new virtual page for each allocation we can still use the underlying physical page using a different virtual page that maps to that physical page. Our approach exploits this idea by using a new virtual page for each allocation of the program but mapping it to the same physical page as the original program (thus using the same amount of physical memory as the original program). Upon deallocation, we can change the permissions on the individual virtual pages but still use the underlying physical memory via different virtual pages. We rely on the memory management unit (MMU) just like in the naive idea to detect all dangling pointer accesses *without* any software checks. If the goal is to guarantee absence of undetected dangling pointer dereferences, then this basic scheme will not allow us to reuse a virtual page ever again for any other allocation in the program. Our second insight is that we can build on a previously developed compiler transformation called Automatic Pool Allocation [46] to alleviate the problem of address space exhaustion. The transformation essentially partitions the memory used by the program into pools (sub heaps) and is able to infer when a partition or a pool is no

longer accessible (using a standard compiler analysis known as escape analysis that is much simpler, but can be less precise, than that required for static detection of dangling pointer references). We leverage this information, to safely reuse address space belonging to a pool, when the memory corresponding to a pool becomes inaccessible.

As our experimental results indicate, our approach works extremely well for server programs. This is because most server programs seem to follow a simple memory allocation and usage paradigm: they have low or moderate frequency of allocations and deallocations but do have many memory accesses. Our approach fits well with this paradigm — we move all the run-time overheads to allocation and deallocation points (since we require extra system call per allocation and deallocation), and we do not perform any checks on individual memory accesses themselves.

Our approach has several practical strengths. First, we do not use fat pointers or meta-data for individual pointers. Use of such meta-data complicates interfacing with existing libraries and requires significant effort to port programs to work with libraries. Second, if reuse of address space is not important ¹, particularly during debugging, our technique can be directly applied on the binaries and does not require source code; we just need to intercept all calls to malloc and free from the program. Finally, we do not change the cache behavior of the program; so carefully memory-tuned applications can benefit from our approach without having to retune to a new memory management scheme.

There are two main limitations to our approach. First, since we use a system call on every memory allocation, applications that do in fact perform a lot of allocations and deallocations will have a big performance penalty (our approach can still be used for debugging such applications). However, we expect many security critical server software to not exhibit this behavior. Second, since each allocation has a new virtual page, our approach has more TLB (“translation lookaside buffer”) misses than the original program. We are currently investigating simple architectural improvements that can mitigate both of these problems by changing the TLB structure.

We briefly summarize the contributions of this work:

- We propose a new technique that can effectively detect *all* dangling pointer errors by making use of the MMU, while still using the same physical memory as the original program.

¹As shown later in Section 7.1.4, on 64-bit systems programs will run for at least 9 hours before running out of virtual pages

- We propose the use of previously developed compiler transformation called Automatic Pool Allocation to reduce the problem of address space exhaustion.
- We evaluate our approach on five unix utilities, five daemons and an allocation intensive benchmark suite. Our overheads on unix utilities are less than 15% and on server applications are less than 4%. However, our overheads on allocation intensive benchmark suite are much worse (up to 11x slowdown).

The rest of this chapter is organized as follows. Section 7.1 contains a detailed description of our overall approach and our implementation. Section 7.2 gives experimental evaluation of our approach. Section 7.3 discusses related work and Section 7.4 concludes with possible future directions of this work.

In the rest of this chapter, by dangling pointer errors we mean use of pointers to freed heap memory, where use of a pointer is a read, write or free operation on that pointer.

7.1 Detecting Dangling Pointers : Our Approach

7.1.1 Overview of the Approach

The memory management unit (MMU) in most modern processors performs a run-time check on every memory access by looking at the permission bits of each page. Debugging tools like Electric Fence [56] and PageHeap [51] exploit this check to detect dangling pointer accesses at run-time. Since the MMU does checks only at page level, these tools allocate only one object per (virtual and physical) page and change the permissions at a free operation to protect the page. Any dangling pointer access will then result in a hardware trap, which can be caught. However, allocating only one object per *physical* page would quickly exhaust physical memory. Furthermore, changing the allocation pattern this way would potentially lead to poor cache performance in physically indexed caches. Our first insight addresses this drawback:

Insight1: *Mapping multiple virtual pages to the same physical page enables us to set the permissions on each individual virtual page separately while still allowing use and reuse of the entire physical page via different virtual pages.*

With this approach, the consumption of physical memory would be (nearly) identical to the original program, and the multiple objects could be contiguous within the page, preserving spatial locality in physically indexed caches. Moreover, as we show later, with a minor increase in memory allocation (one word per allocation) this scheme can be implemented *without* requiring any changes to the underlying memory allocator. In a practical system, this is likely to be significant, since programs that are already tuned to an existing memory allocation strategy do not need to be retuned again.

Virtual memory pages, however, still cannot be reused to ensure that any access to a previously freed object is detected arbitrarily far in the future. As noted previously in chapter 3.4, the Automatic Pool Allocation transformation provides bounds on the lifetimes of pools containing heap objects, such that *there are no live pointers to a pool* after the `pooldestroy` operation on the pool. This yields:

Insight2: *For a program transformed using Automatic Pool Allocation, it is safe to release all the virtual pages assigned to a pool at the `pooldestroy` operation on the pool.*

The remaining limitation of this approach is that long lived pools (e.g., pools reachable via a global pointer, or pools created and destroyed in the `main` function for other reasons) effectively live throughout the execution and their virtual pages cannot be reused. In section 7.1.4, we discuss three strategies to avoid this problem in practice.

Overall, we now have the ability to reuse physical pages as well as the original program does and reuse virtual memory pages partially. Nevertheless, there are several key implementation and performance issues that must be considered to make this approach practical for realistic production software. These issues are addressed as we describe the details of the approach below.

7.1.2 Page Mapping for Detecting Dangling Pointer Errors

The primary mechanism we use for detecting dangling pointer references (i.e., a load, store, or free to a previously freed object) is to use a distinct virtual page or sequence of pages for every dynamically allocated object. When an object is freed, the protected bit is set for the page or pages using the `mprotect` system call so that any subsequent reference to the page causes an access violation, which is handled by our run-time system.

We assume in this subsection that a standard heap allocator is used. The basic approach works as follows, assuming `malloc` and `free` are the interface for the underlying system allocator.

Allocation: An allocation request is passed to `malloc` with the size incremented by `sizeof(addr_t)` bytes; the extra bytes at the start of the object will be used to record an address for book-keeping purposes. Let a be the address returned by `malloc`, $\text{Page}(a) = a \& \sim (2^p - 1)$ and $\text{Offset}(a) = a \& (2^p - 1)$, where p is \log_2 of the VM page size. The latter two denote the start address of the page containing a and the offset of a relative to the start of the page. We then invoke a system call to assign a fresh virtual page (or pages) that share the same physical memory as the page(s) containing a . On Linux, we do this using `mremap(old_address, old_size, new_size, flags)`, with `old_size = 0`, which returns a new page-aligned block of virtual memory at least as large as `new_size`.² If the new page address is P_{new} , we return $P_{new}(a) + \text{Offset}(a) + \text{sizeof}(\text{addr}_t)$ to the caller.

Note that the underlying allocator still believes that the allocated object was at address a , whereas the caller sees the object on a different page but at the same location within the page. We refer to virtual page $\text{Page}(a)$ as the *canonical page* (the one assumed by the allocator) and the actual virtual page P_{new} as the *shadow page* for the object. We have to record the original page $\text{Page}(a)$ for the object to support deallocation; we do this in the extra `sizeof(addr_t)` bytes we allocated above. Note that `malloc` implementations usually add a header recording the size of the object just before the object itself so we are effectively extending that header to also record the value of $\text{Page}(a)$.

The net result of this approach is that multiple objects can live in a single physical page and the underlying allocator believes they live in a single virtual page (the canonical page). The program, however, is given a distinct virtual page (a shadow page) for each object in the physical page.

Deallocation: On a deallocation request for address f , we first look up the canonical page for this object which was recorded at $f - \text{sizeof}(\text{addr}_t)$. Note that this read operation will cause a run-time error if the object has already been freed because the virtual page containing this memory would have been protected as explained next. We read the size of the object using the metadata recorded by `malloc`, use this size to compute how many pages the object spanned, and use the

²This behavior is undocumented in the man page but is described here [64]. On systems where this feature is not available, we can use `mmap` with an in-memory file system.

system call `mprotect` to set the protection status of the page(s) containing f to the state `PROT_NONE`. This will cause any future read or write of the page to trap. If the canonical page is `Page_C^f`, we invoke `free(Page_C^f + Offset(f))` to free the object. The allocator marks the object within the canonical page as free, allowing that range of virtual memory (and therefore the underlying physical memory) to be reused for future allocations. The shadow page(s) containing the object at f cannot be reused, at least with the approach as described so far.

7.1.3 Reusing Virtual Pages via Automatic Pool Allocation

In Automatic Pool Allocation, each pool created by the program at run-time is essentially a distinct heap, managed internally using some allocation algorithm. We can use the remapping approach described above *within each pool* created by the program at run-time. The key benefit is that, at a `pool_destroy`, we can release all (shadow and canonical) virtual memory pages of the pool to be reused by future allocations. Note that physical pages will continue to be reused just as in the original program, i.e., the physical memory consumption remains the same as in the original program (except for minor differences potentially caused by using the pool allocator on top of the original heap [46]).

The only significant change in the Allocation and Deallocation operations described above is for reusing virtual pages. This is slightly tricky because we need to reuse virtual pages that might have been aliased with other virtual pages previously. One simple solution would be to use the `unmap` system call to release previous virtual-to-physical mappings for all pages in a pool after a pool destroy. `unmap` would work for both canonical and shadow pages because these are obtained from the Operating System (OS) via `mmap` and `mremap` respectively. Canonical pages are obtained in contiguous blocks from the underlying system (via `mmap`) and the blocks can be unmapped efficiently. The shadow pages, however, are potentially scattered around in the heap, and in the worst case may require a separate `unmap` operation for every individual object allocated from the pool (in addition to the earlier `mprotect` call when the object was freed). This could be expensive.

We avoid the explicit `munmap` calls by maintaining a free list of virtual pages shared across pools and adding all pool pages to this free list at a `pool_destroy`. We modified the underlying pool allocator to obtain (canonical) pages from this free list, if available. If this free list is empty, we

use *mmap* to obtain fresh pages from the system as before. For each allocated object, the shadow page(s) is(are) obtained using *mremap* as before to ensure a distinct virtual memory page.

A `pool_free` operation works just like the Deallocation case described previously, and invokes the underlying `pool_free` on the canonical page. A `pool_destroy` operation simply returns all canonical *and* shadow pages in the pool to the shared free list of pages.

7.1.4 Avoiding Costs of Long-lived Pools

As noted earlier in Section 7.1.1, the main limitation of our approach as described so far is that virtual pages in pools with lifetimes spanning the entire execution will never be reused. Our examination of several Unix daemons in Section 7.2.3 shows that this problem arises in very few cases.

If it occurs in a specific program, it would impose two costs in practice: (1) A long-running program may eventually run out of virtual memory. (2) Small operating system resources (the page table entry) are tied up for each non-reusable virtual page. The page cannot be unmapped to prevent reuse of the virtual addresses. We propose three solutions to avoid these problems in practice.

The simplest solution is to start reusing virtual pages when we run out of virtual addresses or at some regular (but large) interval. A simple calculation shows that on a 64-bit Linux system (and assuming a maximum of 2^{47} bytes of virtual memory for a user program), even an extreme program that allocates a new 4K-page-size object *every microsecond* with no reuse of these pages, can operate for 9 hours before running out of virtual pages ($2^{47}/(2^{12} \times 10^6 \times 86,400)$). In practice, even with no reuse at all, we expect typical servers to be able to operate for days before running out. The small probability of not detecting a dangling pointer in such situations appears unimportant. In practice, therefore, the second cost above (tying up OS resources) appears to pose a tighter constraint than the first. For this reason, real-world applications would likely choose to reuse memory after a shorter (but still infrequent) interval.

An alternative approach is to run a conservative garbage collector (GC) at the same infrequent intervals (based on the same criteria above) to release the tied-up virtual addresses. This is much simpler and less expensive than using garbage-collection for overall memory management for two

Benchmark	LOC	Execution time in Secs					Slowdown ratios	
		native	LLVM (base)	PA	PA + dummy syscalls	Our ap- proach	Ratio 1	Ratio 2
Utilities								
enscript	8514	1.135	1.077	1.177	1.143	1.238	1.15	1.09
jwhois	10702	0.539	0.534	0.539	0.535	0.534	1.00	0.99
patch	11669	0.174	0.176	0.177	0.179	0.179	1.02	1.03
gzip	8163	4.509	5.419	5.01	4.91	4.943	0.91	1.10
Servers								
ghttpd	6036	4.385	4.507	4.398	4.461	4.486	1.00	1.02
ftpd	23033	2.236	2.293	2.267	2.318	2.291	1.00	1.02
fingerd	1733	1.238	1.285	1.277	1.278	1.285	1.00	1.04
tftpd	880	2.246	2.281	2.289	2.293	2.287	1.00	1.02

Table 7.1: Runtime overheads of our approach. Ratio 1 is the ratio of execution time of our approach with respect to base LLVM, Ratio 2 is the ratio of execution time of our approach to native code. (Two other applications, *telnetd* and *less*, are discussed in text)

reasons. Most importantly, since the actual physical memory consumption is not an issue and GC only needs to ameliorate the two problems above, we can run garbage collection quite infrequently (e.g., once every few hours) and when there is a light load on the server. Second, we only need to use GC to collect memory from the long-lived pools, which are known to the pool allocation transformation. We already have a very simple facility in our system for each run-time pool descriptor to record exactly which currently live pools point to it, i.e., a “dynamic pool points-to graph” [47]. By knowing which pools need to be collected, the collector can use this information to traverse only a subset of the heap.

If the first solution is not acceptable for some reason and conservative GC is not available or unattractive, a third alternative is that the programmer could modify the application so that fewer heap objects are reachable from global pointers. This is similar to (but a subset of) the tuning needed to reduce memory consumption of applications that use GC for memory management, since the latter also requires resetting local and global pointer variables to null as data structures are freed.

Overall, we believe that with one or more of these techniques, the potential costs of lack of reuse in globally live pools is likely to be unimportant in real-world systems.

7.1.5 Implementation

We have implemented the techniques described in this chapter in the pool allocation run-time system developed as a part of SAFECode.

We used the LLVM compiler infrastructure [45] to apply Automatic Pool Allocation to C programs, using the existing version of this transformation with no changes. We modified the pool allocator run-time in minor ways to implement the techniques described earlier. We modified `pooldestroy` to return all pages to a shared free list of pages. We also modified `poolfree` so it did not return unused blocks to this free list. We modified `poolalloc` to try to obtain fresh pages from the free list first when it needs fresh pages (and falling back on `mmap` as before, if the free list is empty). Finally, we wrapped the calls to `poolalloc` and `poolfree` to remap objects from canonical to shadow pages and vice-versa, as described earlier.

7.2 Experimental Evaluation

We present an experimental evaluation of our approach for unix utilities, server applications and few allocation intensive applications. The goal of these experiments is to measure the net run-time overhead of our approach, a breakdown of these overheads contributed by different factors, and the potential for unbounded growth in the virtual address space usage incurred in our approach. Note that our physical memory consumption is almost exactly the same as the original program (except for minor differences when using the pool allocation runtime library) and we do not evaluate the physical memory consumption experimentally.

7.2.1 Run-time Overheads for System Software

We evaluated our approach using few commonly used unix utilities and five server codes – `ghttpd-1.4`, `wu-ftpd-2.6`, `bsd-finger-0.17`, `netkit-telnet-0.17`, and `netkit-tftp-0.17`. The characteristics of these applications are listed in Table 7.1. We compiled each program to the LLVM compiler intermediate representation (IR), performed our analyses and transformations, then compiled LLVM IR back to C and compiled the resulting code using GCC 3.4.2 at `-O3` level of optimization. We performed our experiments on a (32-bit) Intel Xeon with Linux as the operating system. For each

of the server applications, we generated a list of client requests and measured the response time for the requests. We ran the server and the client on the same machine to eliminate network overhead. We conducted each experiment five times and reported the median of the measured times. In case of utilities, we chose a large input size to get reliable measurements. We successfully applied our approach to two interactive applications `netkit-telnetd` and unix utility `less` and did not notice any perceptible difference in the response time. We do not report detailed timings for these two applications.

The “native” and “LLVM (base)” columns in the table represent execution times when compiled directly with GCC -O3 and with the base LLVM compiler (without pool allocation or any of our `mmap` system calls) using the LLVM C back-end. LLVM uses a different set of optimizations from GCC so there is a minor difference in the two execution times. Using LLVM (base) times as our baseline allows us to isolate the affect of the overheads added by our approach. The “native” column shows that the LLVM (base) code quality is comparable to GCC and reasonable enough to use as a baseline. The “PA” column shows the time when we only run the pool allocator and do not use our virtual memory technique, i.e., it shows the effect of pool allocation alone on execution time.

As noted earlier, overheads in our approach could be due to two reasons: (1) use of a system call on every allocation (`mremap`) and deallocation (`mprotect`) (2) TLB miss penalty because we use far more virtual pages than the original program. The “PA + dummy syscalls” column shows the execution time when we do a dummy `mremap` system call on every allocation and a dummy `mprotect` system call on deallocation. This allows us to isolate the overheads due to system calls from that of TLB misses. The “Our approach” column gives the total execution time with our approach.

Ratio 1 gives the ratio of execution time of our approach to that of LLVM (base). Ratio 2 gives the ratio of execution time of our approach to that of native code.

As we can see from column **Ratio1**, our overheads are negligible for most applications. Only one application, `enscript`, has a 15% overhead. These overheads are much better than the any one of the previous approaches for detecting dangling pointers [56, 71, 54]. `enscript` does many allocations and deallocations. In fact, when used with electric fence, `enscript` runs out of physical

Benchmark	Execution time (Secs)		Slowdown ratios	
	Our approach	Valgrind	Our slowdown	Valgrind slowdown
enscript	1.238	29.931	1.15	26.37
jwhois	0.534	1.336	1.00	2.48
patch	0.179	1.461	1.02	8.40
gzip	4.943	94.483	0.91	20.95

Table 7.2: Comparison with Valgrind. Our slowdown ratio is Ratio 1 from Table 7.1

memory. From the “PA + dummy syscalls” column, we can see that the overhead in `enscript` due to system calls is about 6%. We attribute the remaining component of the overhead (around 9%) to TLB miss penalty. Automatic Pool Allocation transformation can sometimes speedup applications (e.g., `gzip`) because of better cache performance [46].

Overall, we believe our low overheads are due to the patterns of memory allocation and use that servers seem to obey: there are relatively few allocations/deallocations (keeping our system call overheads low) but potentially many uses of the allocated memory (we do not incur overheads due to hardware checks).

7.2.2 Comparison with Valgrind

We compared the overheads of our approach with Valgrind [58] (a widely used open source debugging tool) on the four Unix utilities. The servers are spawned off the `xinetd` process for every client request and we are unable to run them under Valgrind. The results are shown in Table 7.2. Valgrind attempts to detect both spatial errors and few dangling pointer errors. We cannot isolate Valgrind overhead for temporal errors, so the comparison is only meant to give a rough indication of the magnitude of difference in overhead. It is worth noting that Valgrind uses a heuristic to detect dangling pointer errors (see Section 7.3 for more discussion) and does not guarantee the absence of undetected dangling pointer accesses. The overheads for Valgrind range from 148% to 2537%, which is orders-of-magnitude worse than ours. In contrast, our approach detects only dangling pointer errors,; our overheads are negligible for three of the applications and 15% for one application.

7.2.3 Address Space Wastage due to Long-lived Pools

We studied the usage (and wastage) of virtual address space incurred by our technique by tracing three of the server programs (`ftpd` , `telnetd` , `ghttpd`) using `gdb`. We found that a common programming model used by these servers is to fork a new process to service each new connection. Although we did not trace `fingerd` and `tftp`, it is clear from the comments in the source code that these codes follow exactly the same programming model; in fact, in case of `tftpd` every command from the client (e.g, get filename) forks off a new process. This model of forking a new process to service requests fits well with our approach. Any wastage in address space in one connection is not carried over to the other connections handled by the server. We expect each individual connection to be of short duration even though all the servers themselves are long running.

We then measured the usage of virtual address space within each individual connection for the three servers.

`Ghttpd` is a webserver designed for small memory foot print and performs only one dynamic allocation per connection. Consequently, there is no virtual memory wastage when we use our approach.

In case of `ftpd` , we found that in a few cases the pool allocation transformation helps in reuse of address space. For example, the `fb_real_path` function in `ftpd` , which resolves sym links, first creates a pool, allocates some memory out of the pool, does some computation, frees the memory, and finally destroys the pool. Any virtual addresses used by this pool are reusable after the pool destroy. However, there are other allocations in `ftpd` that are out of global pools and since global pools get destroyed only at program exit, they do not benefit from the pool allocation transformation. We found that for each command given by the ftp client (e.g, get filename) there are 5-6 allocations from global pools, which tie up that many virtual pages. Thus virtual memory usage increases at the rate of 5-6 pages per command. Although this problem could be alleviated using the techniques described in Section 7.1.4, this problem is unlikely to be important for `ftpd` because the process is killed at end of a user connection.

`Telnetd` performs 45 small allocations (and deallocations) before giving control to the shell in each session (process). It does not do any more (de)allocations and just waits for the session to end. Using our approach, we just use 45 virtual pages for each session. In all these cases, we

	native	LLVM (base)	PA + dummy sys call	Our approach	Ratio 3
bh	15.090	9.723	11.035	12.127	1.25
bisort	2.803	2.641	4.740	8.495	3.22
em3d	9.774	6.830	7.366	7.801	1.14
health	0.319	0.305	2.355	3.429	11.24
mst	0.285	0.166	1.040	1.582	9.53
perimeter	0.187	0.210	1.428	2.188	10.42
power	5.698	2.903	2.959	3.168	1.09
treeadd	0.277	0.293	0.455	1.0777	3.68
tsp	1.753	1.637	3.647	6.749	4.12

Table 7.3: Overheads for allocation intensive Olden benchmarks, Ratio 3 is the slow down of our approach with respect to LLVM base.

guarantee the absence of any undetected dangling pointer accesses.

7.2.4 Overheads for Allocation Intensive Applications

We measured the run-time overheads of our approach when applied to allocation intensive Olden benchmarks. These benchmarks have high frequency of allocations and represent the worst case scenario for our approach. While the overheads for three of the Olden benchmarks were less than 25%, the overheads for the remaining six are high (slowdowns from 3.22 to 11.24). As can be noted from several programs, including `bisort`, `health`, and `mst`, the overheads can be attributed to both the system call overheads and TLB misses. Our approach can be used for such allocation (and deallocation) intensive applications during debugging.

7.3 Related Work

Detecting memory errors in C/C++ programs is a well researched area. A number of previous techniques focus only on spatial memory errors (buffer overflow errors, uninitialized pointer uses, arbitrary type cast errors, etc). As explained in Section 3.1, detecting spatial errors is complementary to our approach. In this section, we compare our approach to only those techniques that detect or eliminate dangling pointer errors.

One way to eliminate dangling pointer errors is to use automatic memory management (garbage collection) instead of explicit memory allocation and deallocation. Where appropriate, that solution

is simple and complete, but it can significantly impact the memory consumption of the program and perhaps lead to unacceptable pause times. For C and C++ programs that choose to retain explicit frees in the program, an alternative solution is required. We focus here on comparing those approaches that detect dangling pointers in the presence of explicit deallocation.

7.3.1 Techniques that Rely on Heuristics to Detect Dangling Pointer

References

A number of systems have been proposed that use heuristic run-time techniques to detect heap errors, including some dangling pointer errors [73, 58, 37, 33]. These techniques do not provide any guarantees about the absence of such errors. The limitation is indeed significant: in fact, these techniques can detect dangling memory errors only as long as the freed memory is not reused for other allocations in the program. Furthermore, all of them rely on heuristics to delay reuse of freed memory, which can increase the *physical* memory consumption.

7.3.2 Techniques that Guarantee Absence of Dangling Pointer References

SafeC [4] is one of the earliest systems to detect (with high probability) all memory errors including all dangling pointer errors in C programs. SafeC creates a unique capability (a 32-bit value) for each memory allocation and puts it in a Global Capability Store (GCS). It also stores this capability with the meta-data of the returned pointer. This meta-data gets propagated with pointer copying, arithmetic. Before every access via a pointer, the pointer's capability is checked for membership in the GCS. A `free` operation removes the capability from the global capability store and all dangling pointer accesses are detected. FisherPatil [54] and Xu et. al [71] propose improvements to the basic scheme by eliminating the need for fat-pointers and storing the meta-data separately from the pointer for better backwards compatibility. To be able to track meta-data they disallow arbitrary casts in the program, including casts from pointers to integers and back. Their overheads for detecting only the temporal errors on allocation intensive Olden benchmarks are much less than ours – about 56% on average (they do not report overheads for system software).

However, the GCS can consume significant memory: they report increases in (physical and virtual) memory consumption of factors of 1.6x - 4x for different benchmarks sets [71]). For servers

in particular, we believe that such significant increases in memory consumption would be a serious limitation.

Our approach, on the other hand, provides better backwards compatibility: we allow arbitrary casts including casts from pointers to integers and back. Furthermore, our approach uses the memory management unit to do a hardware runtime check and does not incur any per access penalty. Our overheads in our experiments on system software, with low allocation frequency, are negligible in most cases and less than 15% in all the cases. However, for programs that perform frequent memory allocations and deallocations like the Olden benchmarks, our overheads are significantly worse (up to 11x slowdown). It would be an interesting experiment to see if a combination of these two techniques can work better for general programs.

7.3.3 Techniques that Check using MMU

As mentioned earlier, Electric Fence [56] and PageHeap [51] are debugging tools that make use of the memory management unit (MMU) to detect dangling pointer errors (and some buffer overflow errors) without inserting any software checks on individual loads/stores. However, both the tools allocate only one memory object per virtual and physical page, and do not attempt to share a physical page through different virtual pages. This means that even small allocations use up a page of actual physical memory. This results in several fold increase in memory consumption of the applications. In turn, the applications exhibit very bad cache behavior increasing the overheads of the tools. These overheads effectively limit the usefulness of the tools to debugging environments. In contrast, we share and reuse physical memory, and use automatic pool allocation to reuse virtual addresses.

7.4 Detecting Dangling Pointer References: Concluding Discussion

In this chapter, we have presented a novel technique to detect uses of pointers to freed memory that relies on two simple insights: (1) Using a new virtual page for every allocation but mapping it to the same physical page as the original allocator. (2) Using a compiler transformation called automatic

pool allocation to mitigate the problem of virtual address space exhaustion. We evaluated our approach on several Unix utilities and servers and we showed that our approach incurs very low overhead for all these cases – less than 4% for the server codes and less than 15% for the utilities. *These overheads are low enough to be acceptable even in production code* (although our techniques could be very effective for debugging as well). We believe this is the first time such a result has been demonstrated for the run-time detection of dangling pointer errors.

For C or C++ programs that have frequent allocations and deallocations, two main performance problems remain — the system call overhead for allocations and deallocations, and the TLB miss overhead. As an extension to this work, we plan to investigate simple OS and architectural enhancements that can reduce both these kinds of overheads and make our approach applicable to these other kinds of software.

Chapter 8

SAFECode for Embedded Systems

In application domains like embedded systems any memory safety solution that imposes performance overheads is unattractive. For this reason, the SAFECode approach described thus far though has low overheads, may not be directly suitable for embedded systems. In this chapter, we present a new mode of SAFECode by including some restrictions on the language usage that enable complete static checking for memory safety¹. We believe that our restrictions are not too onerous in the context of embedded systems and show that for a fairly large subset of embedded benchmarks, memory safety can be guaranteed without run-time checks or garbage collection.

In the next few sections, we describe the language restrictions that we impose for static checking and the (new) compiler techniques that enable static checking for the restricted language.

8.1 Language Restrictions

We define a subset of C which can be completely statically checked for memory safety. The key restrictions are (a) no casts to a pointer type from any other non-equivalent type; (b) a union must not contain types that are illegal to cast to each other; (c) local variables must be initialized before being referenced; (d) no structure type could be larger than the size of the reserved address range (or if it were, references to such an object would incur null pointer checks); (e) the address of a stack variable must not be stored in the heap or in a global variable, or returned from a function; and (f) a collection of rules requiring that array index expressions must have an affine relationship with the corresponding extent expression (we rejected programs that violated this rule, instead

¹The work presented in this chapter is joint work with Sumant Kowshik, Chris Lattner

of introducing runtime checks); and (g) all functions of the program must be available, except a collection of trusted library routines with enforced preconditions and assumed postconditions. Programs that violated these rules are simply rejected.

Rules (a-d) above are easy to check with simple type checking and dataflow analysis. Rules (e-g) and our static safety checking techniques required more analysis and transformations. More specifically, we exploited three new compiler techniques and two new runtime techniques to achieve memory safety statically. These are described briefly below.

8.2 Uninitialized Pointers

References to uninitialized scalar variables are identified through a simple dataflow analysis (Rule c). For pointer fields in aggregate types, we follow the SAFECode strategy described earlier in Chapter 4.6.2: we initialize them to the base of reserved address and exploit hardware checks to perform the uninitialized pointer checking.

8.3 Stack Safety

Here, we follow a variant of the strategy used in the SAFECode approach for stack safety. In the original SAFECode, described in chapter 4.4, all stack objects that may potentially escape the function in which they are allocated, are actually moved to the heap. SAFECode in embedded mode simply reject programs for which it is unable to prove stack safety (i.e, the stack object does not escape the allocated function).

8.4 Dangling Pointers to Freed Memory

Here again, we follow the same approach as outlined in chapter 4. We use automatic pool allocation and then rely on the type homogeneity of pools to ensure that any dangling pointer reference can only access objects of the same type. Just like earlier, we restrict that memory from a pool may not be released until the life time of the pool ends.

Restricting memory from being released could eliminate some reuse of memory between two different pools (“cross-reuse”), and so increase memory consumption. We therefore perform an

additional interprocedural flow analysis to identify cases when this can happen and report them to the programmer. Such cases can be profiled to identify if the memory increase is significant (we expect this is quite unlikely because most pools are either short-lived or have significant internal reuse), and to restructure the program if necessary to avoid the increase. We found that very few pools actually have potential cross-reuse, at least across a wide range of simple embedded benchmarks.

Note that our language restrictions ensure that there are no non type homogeneous (**TU**) pools. Specifically, we reject programs that have these **TU** pools. We found in our experiments that most embedded programs do not have **TU** pools or if they do can be easily modified to not have **TU** pools.

Benchmark	Lines of Code	Lines of Code Modified for type safety	Lines of Code Modified for array safety	Heap and Pointer safety	Stack safety	Array safety
control						
Pendulum	300(Average)	0	0	Yes	Yes	Yes
Pendubot	1300(Average)	0	31	Yes	Yes	Yes
TinyOS apps	300(Average)	0	0	Yes	Yes	Yes
automotive						
basicmath	579	1	3	Yes	Yes	Yes
bitcount	17	5	0	Yes	Yes	Yes
qsort	156	0	1	Yes	Yes	Yes
susan	2122	1	0	Yes	Yes	No
office						
stringsearch	3215	0	3	Yes	Yes	Yes
security						
sha	269	0	1	Yes	Yes	Yes
blowfish	1502	1	5	Yes	Yes	Yes
rijndael	1773	3	6	Yes	No	Yes
network						
dijkstra	348	0	0	Yes	Yes	No
telecomm						
CRC 32	282	0	1	Yes	Yes	Yes
adpcm codes	741	0	0	Yes	Yes	No
FFT	469	0	0	Yes	Yes	No
gsm	6038	0	0	Yes	Yes	No
multimedia						
g721	1622	11	0	Yes	Yes	No
mpeg(decode)	9839	0	0	Yes	Yes	No
epic	3524	7	0	Yes	Yes	No
rasta	7373	25	0	Yes	Yes	No
Totals: 20	41769	70	53	20	19	11

Table 8.1: Benchmarks, code sizes, and analysis results

8.5 Array Safety and String and I/O Libraries

We developed an interprocedural constraint propagation algorithm (described in detail in Chapter 9), that symbolically performs array bounds analysis on the input program. The algorithm propagates affine constraints on integer variables from callers to callees (for incoming integer arguments and global scalars) and from callees to callers (for integer return values and global scalars). It then perform a symbolic bounds check for each index expression using integer programming (we use the Omega Library from Maryland [38]) to symbolically prove the safety of the array access. The algorithm uses the result of the call graph to find callers of each function and callees of each call site.

8.6 Summary of Results

We implemented the compiler for this restricted language as different mode in the SAFECODE framework. We evaluated the applicability and the memory overhead of our system on a diverse collection of embedded programs from two widely used benchmark suites, MiBench [30] and MediaBench [48], as well as some control and sensor applications, shown in Table 8.1 and Table 8.2. Our results show that we are able to ensure the safety of pointers and dynamic memory usage *in all these programs* without incurring any run-time overhead. This is due to a combination of our technique for preventing null pointer dereferences using a run-time hardware check and the pool allocation technique for ensuring safe dereferencing of dangling pointers, both of which work successfully for all the programs. Our compiler analysis identifies specific data structures in three of these programs where our memory management strategy could lead to some potential increase in memory consumption and we found that in all the three cases, the actual increase is not significant. The static technique for checking the lifetimes of stack-allocated data works successfully for 19 of the 20 codes tested. The static array bounds checking analysis, however, is completely successful for only 11 out of the 20 codes. The other codes would require some run-time software checks. Overall, these results show that with the exception of array bounds checks, the set of compiler techniques in this work are able to achieve memory safety without garbage collection and without run-time software checks, for a language that is essentially a “type-safe” subset of C, including

programs with complex pointer-based data structures and extensive heap usage.

Benchmark	Execution Time (secs)			Memory Usage (bytes)				
	Orig time	heap safety time	exec ratio	Orig mem usage	pool alloc mem usage	mem ratio 1	pool alloc + safety restriction	mem ratio 2
automotive								
basicmath	1.667	1.672	1.00	16384	16384	1	16384	1
bitcount	0.710	0.727	1.02	16384	16384	1	16384	1
qsort	0.405	0.404	1.00	24576	24576	1	24576	1
susan	0.670	0.675	1.01	253952	253952	1	253952	1
office								
stringsearch	0.024	0.024	1.00	16384	16384	1	16384	1
security								
sha	0.145	0.138	0.95	24576	24756	1	24576	1
blowfish	0.713	0.722	1.01	24576	24756	1	24576	1
rijndael	0.340	0.366	1.07	24576	24576	1	24576	1
network								
dijkstra	0.340	0.349	1.02	32768	32768	1	32768	1
telecomm								
CRC 32	1.463	1.53	1.04	16384	16384	1	16384	1
adpcm codes	1.255	1.252	1.00	0	0	-	0	-
FFT	0.495	0.478	0.96	540672	540672	1	540672	1
gsm	1.979	1.959	0.98	24576	24576	1	24576	1
multimedia								
g721	0.354	0.355	1.00	24576	24576	1	24576	1
mpeg(decode)	0.331	0.320	0.97	385024	401408	1.04	401408	1
epic	0.126	0.128	1.01	671744	681616	1.01	779920	1.14
rasta	0.124	0.125	1.01	147456	212992	1.44	212992	1

Table 8.2: Execution time and memory usage for heap safety approach. exec ratio is the ratio of execution time after pool allocation to the original time. (A ratio of 2 means the program runs twice as long as the original). mem ratio 1 is the ratio of the memory usage of program after pool allocation to that of the original program. mem ratio 2 is the ratio of the memory usages of pool allocated program with our safety restriction to that of just the poolallocated program

8.7 SAFECODE for Embedded Systems: Concluding Discussion

In this chapter we have discussed a different mode of SAFECODE that places semantic restrictions on the language usage and then statically proves memory safety of programs in the restricted language. Using several benchmarks, we showed that our restrictions are not too onerous for embedded system software. To our knowledge, ours is the first system that achieves 100% static checking for memory safety for any non-trivial subset of C.

Chapter 9

Static Array Bounds Check Analysis

In general, array operations are one of the most expensive to check for memory safety at run time. In this chapter, we present an interprocedural array bounds checking algorithm that can be used in SAFECODE to reduce some of the run-time checks for arrays. The algorithm uses the call graph but not points-to-graph and does not track values through loads/stores. It could not be safely applied without the guarantees provided by SAFECODE regarding the call graph.

For ensuring safety of array accesses statically, the compiler must prove (symbolically) that the index expressions in an array reference lie within the corresponding array bounds on all possible execution paths. For each index expression, this can be formulated as an integer constraint system with equalities, inequalities, and logical operators used to represent the computation and control-flow statements of the program. Unfortunately, satisfiability of integer constraints with multiplication of symbolic variables is undecidable. A broad, decidable class of symbolic integer constraints is *Presburger arithmetic*, which allows addition, subtraction, multiplication by constants, and the logical connectives \vee , \wedge , \neg , \exists and \forall . (For example, the Omega library [38] provides an efficient implementation that has been used for solving such integer programming problems in compiler research.) By exploiting static analysis based on Presburger arithmetic we can prove safety of array accesses whose index expression is provably affine in terms of the size of the array.

Checking for safe array usage requires the following three steps:

- generating constraints from each procedure
- interprocedural propagation of constraints
- verifying whether each array access is safe

```

Constraints ::=  AffineConstraint // affine equality or inequality interms of program variables
               |  Constraints && Constraints // conjunction of two constraints
               |  Constraints || Constraints //disjunction of two constraints.

//Helper Functions
isSatisfiable(Constraints c) : returns true if c is satisfiable, false if not. //uses Omega
ControlDependentConditions(var v) : returns a list of Conditions on which v is control dependent.
def(v) : definition of the variable v
VarsUsed(Variable v) : returns a list of variables used in the def of v
CollectConstraintsAtAllCallSitesOfThisFunction() : merges constraints from all call sites of this function
CollectConstraintsOnReturnValue(Function f) : returns the constraints on return value of f
                                               in terms of its arguments (if affine).
stepfn(v) : if v is an induction variable in a loop, it returns the step value of the loop.

// The following tries to check if array access A[i] is safe
AnalyzeArrayAccess(A, i)
begin
  Constraints c = generateConstraints(A);
  c = c && generateConstraints(i); //We merge the constraints of A and i
  c = c && ((A.size < 0) || (i >= A.size))
  if (isSatisfiable(c)) return false //Access is unsafe
  else return true //Access is safe
end

//The following generates constraints for SSA variable v
generateConstraints(Variable v)
begin
  Constraints c;
  if v is in cache return constraints from cache;
  //First check if the definition of v is an affine expression
  if def(v) is affine arithmeticOperation
    c = def(v); //generate constraints for simple arithmetic operations
    c = c && generateConstraints(VarsUsed(def(v)));
  //check def(v) is a malloc or alloca of an array
  else if def(v) is (non char) array allocation of size d
    c = (v.size = d)
  else if def(v) is char array allocation of size d
    c = (v.size = d - 1) //This is because of A5
  else if def(v) is formal argument
    c = CollectConstraintsAtAllCallSitesOfThisFunction();
  else if def(v) is return value of a function call
    c = CollectConstraintsOnReturnValue(callee(def(v)));
    c = c && generateConstraints(VarsUsed(args(def(v))))
  else if def(v) is a PHI(x1,x2) and def(v) is induction variable of a loop
    and x2 is coming through backward edge of the loop
    if stepfn(def(v)) is positive  c = c && ((v >= x1) || (v <= x2))
    else if stepfn(def(v)) is negative c = c && ((v <= x1) || (v >= x2))
  //We now add the control dependent conditions and their definitions
  ConditionList = ControlDependentConditions(v);
  foreach (Condition k in ConditionList)
    c = c && k && generateConstraints(VarsUsed(k));
  store constraints of v as c in cache
  return c;
end

```

Figure 9.1: Algorithm for Array bounds checking

9.1 Generating the Constraints

We generate a set of constraints for each array access in a program, including only those constraints that affect the array access. The algorithm is described in Figure 9.1. We use a flow-insensitive algorithm that exploits the SSA representation in LLVM. The LLVM instruction set distinguishes registers (which are in SSA form) and memory, and allocates to registers all local scalar variables (including pointer variables) whose address is not taken. Global variables, heap-allocated data, and address-taken local variables are kept in memory, and are not in SSA form. All instruction operands are SSA registers and memory locations are accessed only via load and store instructions. To get constraints for an array access, $A[i][j]$, we traverse def-use chains backwards from the definitions of the SSA variables holding $\&A$, i , and j respectively. These constraints are simply inequalities on integer SSA variables that can be inferred from the program statements. For most program statements, generating the constraints is straightforward. For example, from a simple statement like $i = (x + z) * 5$, we would generate an affine constraint $i = 5x + 5z$ (our examples use C syntax, although such a statement would internally require three LLVM instructions and two temporaries, not shown here). No constraints are generated for any non-affine expression including a load instruction. Not generating a constraint for a variable makes it unconstrained and the safety checker treats the array access as unsafe, unless the variable is irrelevant. We recursively traverse the def-use chains for each variable (e.g., x and z), stopping only if we encounter a non-affine operation, a formal argument, a return value from a call, or an instruction whose constraints have already been computed and cached. We cache the final constraints on each instruction we traverse so that they can be reused.

In order to speed up our analysis we provide a set of trusted routines with pre-defined constraints listed in Figure 9.1. Finally, to ensure that string routines will not read beyond the size of the array, we always initialize the last character in any array of characters to be null and make sure that the last character is not modified by excluding it in the array size expression in our static analysis.

We explain the basics of our approach with the help of the example in Figure 9.2.

For array access $A[i]$ in Figure 9.2, the constraints we generate are a conjunction of the following:

Library Call	Return Value Constraints	Safety Pre-conditions
<code>n = read(fd, buf, count)</code>	<code>n <= count</code>	<code>buf.size >= count</code>
<code>n = puts(s)</code>	-	-
<code>p = memcpy(p1, p2, n)</code>	<code>p.size = p1.size</code>	<code>p1.size >= n</code>
<code>fp = fopen(p,m)</code>	-	-
<code>n = getc(s)</code>	-	-
<code>n = strlen(s)</code>	<code>n <= s.size</code>	-
<code>p = strcpy(s1,s2)</code>	<code>p.size = s1.size</code>	<code>s1.size >= s2.size</code>
<code>p = strdup(s)</code>	<code>p.size = s.size</code>	-
<code>p = strncpy(s1, s2, n)</code>	<code>p.size = s1.size</code>	<code>s1.size >= n</code>

Table 9.1: Some Trusted Library Routines with Implied Constraints and Preconditions

```

char A[51];          // last character is set to null
...
k = read(fd, A, 50); // requires A.size >= 50; implies k <= 50
if (k > 0) {
    len = strlen(A); // implies len <= A.size
    for (i=0; i < len; i++)
        if (A[i] == '-')
            break;
    ...           // use A and i
}

```

Figure 9.2: Array Usage Example

- `(A.size = 51-1)` generated using the def-use edges from the array declaration with the last character excluded from the size for character arrays as explained earlier.
- `(len <= A.size && k <= 50)` generated using the def-use edges and return value constraints on library functions `strlen` and `read`.
- `(i < len && k > 0)` generated from the control dependence graph using the `ControlDependentConditions` procedure in the algorithm.

For an SSA ϕ node, $x_3 = \phi(x_1, x_2)$, we check if it represents an induction variable, using an existing induction variable analysis [5, 6]. If the ϕ node is not an induction variable, we simply ignore the constraints on the ϕ node since this cannot lead to affine constraints. If the ϕ is an induction variable, we know it merges values from a back edge and a forward edge. If the step

function of the variable is positive, we add the constraint $((x_3 \geq x_1) \vee (x_3 \leq x_2))$, where x_1 comes from a forward edge and x_2 comes from a backward edge. If it is negative, we add the constraint $(x_3 \leq x_1) \vee (x_3 \geq x_2)$. An induction variable with an unknown step function cannot produce affine constraints and will simply be ignored.

Overall, induction variable recognition allows us to generate useful constraints about index variables (e.g., $i \geq 0$), and (together with the renaming of variables in SSA form) avoids generating inconsistent equalities like $i = i + 1$ for both induction variables and ordinary variables.

The complete set of constraints we generate for the example reference are `(A.size = 50 && len <= A.size && k <= 50 && i < len && k > 0 && i >= 0)`.

9.2 Interprocedural Propagation

In many cases, size expressions for an array or constraints on variables used in index expressions must be propagated interprocedurally. For this purpose, we have developed an algorithm for interprocedural constraint propagation. The interprocedural algorithm consists of two passes on the call graph. First, a bottom-up pass gets constraints on return values in terms of procedure arguments. A top down pass then merges constraints on arguments coming in to a procedure from different call sites and then tries to prove safety for all array accesses and safety preconditions in that procedure. Our current implementation cannot prove the safety of accesses which depend on recursive functions. Our experiments have shown that array accesses that depend on recursive functions are very rarely provably affine.

The algorithm has a worst case exponential time complexity. In practice, however, we have found that, for most applications, a simple heuristic like collecting all the constraints for each of the different arrays passed to the procedure and then merging and simplifying them removes many redundant constraints and greatly increases efficiency.

Our static checking algorithm assumes that there will be no overflows and underflows in the integer arithmetic involved in index or array size computations. Statically verifying this is extremely hard and furthermore, once verified, does not allow us to perform many traditional compiler optimizations that reorder computations (unless we also verify that there is no overflow/underflow for any possible reordering of the computations). Fortunately, many modern processors, though not

all, have the ability to raise exceptions on overflow or underflow on arithmetic operations. In the embedded world, most processors derived from the MIPS instruction set (e.g., MIPS32, MIPS64, R4300i) have such an ability. Among the general purpose processors, DEC Alpha, VAX, and IBM/S 360 descendants provide such a feature. The x86 and sparc architectures do not automatically raise hardware exceptions upon overflow/underflow but set some flags in condition code register. However, they do provide special “trap on overflow” instructions like *INTV* in x86 and *tcc* in sparc, which when inserted after an arithmetic operation would check the overflow/underflow flag and raise an exception if the flag is set. To guarantee safety on these processors, we would need to insert the corresponding “trap on overflow” instruction after every arithmetic operation that affects an array access.

9.3 Proving Array Safety

We use the Omega integer set library [38] to test each array index expression for safety. Once we generate constraints for an array reference, we add conditions representing array bounds violations for the reference (such as `(i < 0 || i >= A.size)` in the earlier example). We then use the Omega library to check if the resulting constraint system is satisfiable. If the system is not satisfiable (as we have here), the constraints have been proven inconsistent and the array access is safe. Otherwise, the access is potentially unsafe and we retain the run-time check.

9.4 Checking Safety Preconditions

To verify the preconditions for each trusted library call (e.g., the call to `read` above), we simply need to check if the negation of the precondition (`(A.size >= 50)`) along with known constraints on variables in the argument expressions (`buf.size` and `count`) result in an inconsistent system. Here, `(A.size < 50 && A.size = 50)` is trivially inconsistent. In this manner, we generate and check the preconditions for every trusted library call used by the program. If a safety precondition fails, we have to insert a run-time check before the trusted call (or include the source of the trusted call in our analysis).

Chapter 10

Conclusion

This thesis has described SAFECODE, a system that guarantees memory safety, provides error detection capabilities, and provides a semantic foundation (a points-to graph, call graph, and type information) for building sound static analyses for nearly arbitrary C programs.

SAFECODE approach has several practical strengths: it is fully automatic and requires no modifications to existing C programs; it allocates and frees memory objects at the same points as the original program (minimizing the need to tune memory consumption); and it supports nearly the full generality of the C language, except for manufactured addresses and some casts from int to pointers.

SAFECODE operates in three modes:

- In the first mode, SAFECODE imposes semantic restrictions on the language usage that enable static checking for memory safety without any performance overhead. This mode is useful for embedded systems where the restrictions are not too onerous.
- In the second mode, SAFECODE provides memory safety guarantees and sound analysis guarantees but does not detect few errors. In particular, it doesn't detect dangling pointer errors. This mode of SAFECODE has very low overhead and can be used for all production systems.
- In the final mode, SAFECODE provides memory safety guarantees, sound analysis guarantees, and complete error checking. This mode is useful for development, debugging, and testing. If the overhead is less, this mode can even be used for production code.

In this thesis, we showed that the analysis guarantees provided by SAFECODE enable many

sophisticated static checking algorithms. We briefly discussed how one such system (ESP) can take advantage of SAFECode guarantees.

We evaluated SAFECode on two sets of benchmarks and server applications and showed that the overhead incurred by SAFECode is less than that of other existing approaches that guarantee memory or type safety.

We believe SAFECode is an interesting, useful, and practical alternative to existing techniques for providing memory safety and sound analysis guarantees.

10.1 Future Work

There are many possible future directions for the SAFECode work. We discuss some of them below.

10.1.1 Applying SAFECode to Operating System Components

In this thesis, we focussed on techniques that provide safety guarantees to applications. We found that applying the same techniques to operating system components (e.g. Linux kernel components) is non trivial. This is because of two main reasons: (1) Linux kernel uses a memory management scheme that is different from ordinary applications (2) some of the kernel components use inline assembly that is unanalyzable in the current framework.

The first issue arises, as Linux kernel uses a pool based memory management scheme instead of the normal malloc/free. However, the pools in Linux kernel, in general, do not satisfy the property of automatically generated pools that SAFECode relies on. In particular, we found that it is often the case that there are multiple pools that correspond to the same alias node unlike the automatically generated pools. Moreover, the pool(s) for a given pointer is embedded in the source and somehow needs to be extracted out to do the necessary SAFECode analysis. In an ongoing work, we are working on extracting the pool information automatically from the kernel code. We are also developing new techniques that can reconcile the difference in properties of the manually written kernel pools and the pools that would have been inferred by the Automatic Pool Allocation scheme.

The second problem could be solved by either (1) converting the inline assembly to external C functions and making conservative assumptions about what they modify or (2) using a virtual

instruction set and porting Linux kernel to that instruction set. For example, LLVA [17] work ports the Linux kernel to a typed virtual instruction set based on LLVM's intermediate representation. In an ongoing work, we are considering the use of LLVA to analyze the Linux kernel.

10.1.2 More Precise Type Inference

SAFECode inserts several run-time checks on pointers read from **TU** pools. The number of **TU** pools in the program depend on two factors: (1) use of type unsafe features of C (2) imprecision in the pointer analysis that infers the type for the pools. While we cannot reduce the number of **TU** pools due to (1), it is certainly possible to reduce the number of pools due to (2). For example, C programmers commonly use casts to `void*` to implement the subtyping in object oriented style of programming. Some of these casts will result in **TU** pools since the pointer analysis that we use currently recognizes only a limited form of upcasts. Other approaches like CCured [52] have recognized both upcasts and downcasts and have eliminated many unnecessary run-time checks. We plan to extend the pointer analysis used in SAFECode to recognize downcasts like CCured to further improve the performance.

10.1.3 Non-unification based Pointer Analysis

One of the limitations of the SAFECode approach is that it supports only unification based pointer analysis. As explained in chapter 4.5.3, extensions to non-unification based pointer analysis are possible and require meta-data on individual pointer values.

References

- [1] MySQL double free heap corruption vulnerability. <http://www.securityfocus.com/bid/6718/info>, Jan 2003.
- [2] MITKRB5-SA: double free vulnerabilities. <http://seclists.org/lists/bugtraq/2004/sep/0015.html>, Aug 2004.
- [3] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1995.
- [4] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1994.
- [5] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computation*, pages 242–249, 1994.
- [6] Johnie Birch. Using the chains of recurrences algebra for data dependence testing and induction variable substitution. Master’s thesis, Computer Science Dept., Florida State University, Tallahassee, FL, 2002.
- [7] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2000.
- [8] Greg Bollella and James Gosling. The real-time specification for Java. *IEEE Computer*, 33(6):47–54, 2000.

- [9] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2003.
- [10] Martin Christopher Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.
- [11] Shuo Chen. *Design for Security: Measurement, Analysis and Mitigation Techniques*. Ph.d. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
- [12] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2004.
- [13] Jeremy Condit, Mathew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, June 2003.
- [14] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.
- [15] Crispian Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pontguard: Protecting pointers from buffer overflow vulnerabilities.
- [16] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.
- [17] John Criswell, Brent Monroe, and Vikram Adve. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)*, 2006.

- [18] Manuvir Das, Sorin Lerner, and Mark Siegle. Esp: Path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, Jun 2002.
- [19] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Snowbird, UT, June 2001.
- [20] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, Philadelphia, USA, June 2006.
- [21] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECODE: Enforcing alias analysis for weakly typed languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2006.
- [22] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, February 2005.
- [23] Igor Dobrovitski. Exploit for cvs double free() for linux pserver. <http://seclists.org/lists/bugtraq/2003/feb/0042.html>, Feb 2003.
- [24] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *Proc. of ACM SIGSOFT international symposium on Software testing and analysis*, 2004.
- [25] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Ccssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, June 2003.
- [26] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, New York, NY, USA, 2003.

- [27] David Gay and Alexander Aiken. Memory management with explicit regions. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 313–323, Montreal, Canada, 1998.
- [28] Eric Grevstad. Cpu-based security: The nx bit. <http://hardware.earthweb.com/chips/article.php/3358421>.
- [29] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2002.
- [30] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.
- [31] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proc. 28th Int'l Conf. on Software Engineering (ICSE)*, Shanghai, 2006.
- [32] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, Jun 2002.
- [33] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX*, 1992.
- [34] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.
- [35] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in Cyclone. In *Proc. of the 4th international symposium on Memory management (ISMM)*, 2004.
- [36] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

- [37] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.
- [38] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Computer Science Dept., U. Maryland, College Park, April 1996.
- [39] Samuel C. Kendall. Bcc: Runtime checking for c programs. In *In Proceedings of the USENIX*, 1983.
- [40] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proc. Int’l Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Grenoble, Oct 2002.
- [41] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Comp. Sci. Dept., Univ. of Illinois, Urbana, IL, May 2005.
- [42] Chris Lattner and Vikram Adve. Automatic pool allocation: Compile-time control of data structure layout in the heap. Tech. Report UIUCDCS-R-2004-2465, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.
- [43] Chris Lattner and Vikram Adve. Automatic Pool Allocation for Disjoint Data Structures. In *MSP*, Berlin, Germany, Jun 2002.
- [44] Chris Lattner and Vikram Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.
- [45] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int’l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.
- [46] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

- [47] Chris Lattner and Vikram Adve. Transparent Pointer Compression for Linked Data Structures. In *MSP*, Chicago, IL, Jun 2005.
- [48] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
- [49] X. Leroy. Exploiting type systems and static analyses for smart card security. In *Cassis International Workshop*, Marseille, March 2004.
- [50] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2001.
- [51] Microsoft. How to use Pageheap.exe in Windows XP and Windows 2000. <http://support.microsoft.com/?kbid=286470> .
- [52] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Language and Systems*, 27(3):477–526, 2005.
- [53] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *POPL*, London, January 2002.
- [54] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software–Practice and Experience*, 27(1):87–110, 1997.
- [55] Harish Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.
- [56] Bruce Perens. Electric fence malloc debugger. <http://perens.com/FreeSoftware/ElectricFence/> .
- [57] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium*, February 2004.
- [58] J. Seward. Valgrind, an open-source memory debugger for x86-gnu/linux.

- [59] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM symposium on Principles of programming languages (POPL)*, 1996.
- [60] Joseph L. Steffen. Adding run-time checking to the portable c compiler. In *Software: Practice and Experience*, 1992.
- [61] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–768, July 1998.
- [62] Mads Tofte, Lars Birkedal, Martin Elsmann, Niels Hallenberg, Tommy Højfeldt Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, 1997.
- [63] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, pages 132(2):109–176, February 1997.
- [64] Linus Torvalds. mremap feature discussion, see <http://lkm1.org/lkm1/2004/1/12/265>.
- [65] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symp.*, pages 3–17, San Diego, CA, February 2000.
- [66] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.
- [67] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Comp. Sci.*, vol. 2071, 2001.
- [68] R . Wojtczuk. Defeating solar designer nonexecutable stack patch. <http://www.securityfocus.com/archive/1/8470>, Jan 1999.
- [69] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.

- [70] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.
- [71] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proc. 12th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 117–126, 2004.
- [72] Suan Yong. *Runtime monitoring of C programs for security and correctness*. Ph.d. Thesis, Department of Computer Science, University of Wisconsin-Madison, 2004.
- [73] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Foundations of Software Engineering*, 2003.
- [74] Yves Younan. An overview of common programming security vulnerabilities and possible solutions. Master’s thesis, Vrije Universiteit Brussel, 2003.
- [75] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic buffer overflow detection. In *BUGS : Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
- [76] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT symposium on Foundations of software engineering*, 2004.

Author's Biography

Dinakar Dhurjati received his Bachelors degree from the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) - Delhi. He attended University of Illinois at Urbana-Champaign for his graduate studies and earned his Ph.D. degree in Computer Science. His current research interests are in the areas of software reliability and security. Post graduation, he is planning to join DoCoMo USA Labs as a researcher.